ИМС Propeller
Руководство по применению

Версия 1.1

ГАРАНТИЯ

Parallax Inc гарантирует отсутствие в своих продуктах дефектов в материалах и исполнении сроком на 90 дней с момента получения продукта. Если Вы обнаружите дефект, то Parallax Inc, по своему выбору, восстановит или заменит товар, либо возместит затраченные на покупку средства. Перед возвращением продукта в Parallax, запросите номер Разрешения на Возвращение Товара (RMA). Запишите номер RMA на внешней стороне упаковки, используемой для возврата товара в компанию Parallax. Вместе с возвращенными товарами, пожалуйста, укажите следующее: ваше полное имя, номер телефона, адрес отправления и описание проблемы. Parallax возвратит ваш продукт, или его замену, используя тот же самый метод доставки, который использовался для доставки товара в Parallax.

14-ДНЕВНАЯ ГАРАНТИЯ С ВОЗВРАТОМ ДЕНЕГ

Если, в течение 14 дней после получения товара, Вы обнаружите, что он не удовлетворяет Ваши потребности, Вы можете возвратить его с возмещением потраченной на покупку суммы. Parallax Inc возвратит покупную цену продукта, исключая стоимость доставки и погрузочно-разгрузочных работ. Эта гарантия теряет силу, если в продукт были внесены изменения или повреждения. См. выше главу «Гарантия» с инструкцией по возвращению продукта в Parallax.

АВТОРСКИЕ ПРАВА И ТОРГОВЫЕ МАРКИ

ISBN 9-781928-982470

1.1.0-09.03.05-HKTP

ОТКАЗ ОТ ОТВЕТСТВЕННОСТИ

Компания Parallax Inc. не несет ответственности за специальные, непредвиденные, или косвенные убытки, следующие из любого нарушения гарантийных обязательств, или согласно любой теории права, включая потерянную прибыль, время простоя, престиж фирмы, повреждение или замену оборудования или собственности, или любых затрат восстановления, перепрограммирования, или репродуцирования любых данных, сохраненных в или используемых с продуктами Parallax. Parallax Inc. также не ответственна за любой личный ущерб, включая угрозу жизни и здоровью, в результате использования любого из наших продуктов. Вы несете полную ответственность за применение микроконтроллера Propeller, независимо от того, насколько это может быть опасным для жизни.

ИНТЕРНЕТ-ФОРУМЫ

Мы поддерживаем активные и доступные через сеть форумы обсуждения для людей, заинтересованных продуктами Parallax. Эти форумы доступны на http://www.parallax.com через меню "Support" → "Discussion Forums". Ниже приведены форумы, расположенные на нашем веб-сайте:

- Propeller chip - этот форум специально предназначен для наших пользователей, использующих микросхемы и продукты Propeller.
- BASIC Stamp - этот форум широко используется инженерами, людьми, увлечёнными своим хобби и студентами, которые делятся своими проектами с использованием BASIC Stamp и задают вопросы.
- Stamps in Class® – форум создан для педагогов и студентов; подписчики обсуждают использование программы Stamps in Class в их курсах. Форум обеспечивает возможность студентам и педагогам задать вопросы и получить ответы.
- HYDRA – для любителей системы разработки видеоигр на базе ИМС Propeller.
- Parallax Educators - частный форум исключительно для педагогов и тех, кто вносит свой вклад в развитие Stamps in Class. Parallax создал эту группу, чтобы обеспечить обратную связь с нашими курсами и дать возможность педагогам разработать и выпустить Руководства Преподавателя.
- Robotics - созданный для обсуждения роботов Parallax, этот форум предназначен для осуществления открытого диалога между энтузиастами робототехники. Темы форума включают трансляцию, исходный текст, развитие и ручные обновления. Здесь обсуждаются роботы Boe-Bot®, Toddler®, SumoBot®, HexCrawler и QuadCrawler.
- SX Microcontrollers and SX-Key - обсуждение программирования микроконтроллера SX с ассемблером Parallax, инструментальных средств SX - Key® и компиляторов BASIC и C третьих производителей.
- Javelin Stamp - Обсуждение применений и разработок с использованием Javelin Stamp, модуля Parallax, который запрограммирован с использованием подмножества языка программирования Sun Microsystems' Java®.

ОПЕЧАТКИ

Нами прилагаются большие усилия для обеспечения верности наших текстов, однако ошибки все же еще могут существовать. Если Вы обнаружите ошибку - пожалуйста, сообщите нам по электронной почте: editor@parallax.com. Мы постоянно стремимся улучшить все наши образовательные материалы и документацию, и часто проверяем наши тексты. Лист опечаток со списком известных ошибок и исправлений для каждого документа обычно размещается на нашем вебсайте, www.parallax.com. Пожалуйста, обращайтесь к веб-страницам с файлами документации для конкретных изделий для проверки на наличие файлов опечаток.

ПОДДЕРЖИВАЕМЫЕ АППАРАТНЫЕ СРЕДСТВА, ВСТРОЕННОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Это руководство действительно для следующих версий аппаратных средств, программного и встроенного обеспечения:

Аппаратное	Программное	Встроенное
P8X32A-D40 P8X32A-Q44 P8X32A-M44	Propeller Tool v1.2.5	P8X32A v1.0

БЛАГОДАРНОСТИ

Автор: Jeff Martin. Формат и Редактирование: Stephanie Lindsay.
Дизайн обложки: Jen Jacobs; Техническая Графика: Rich Allred; с большой благодарностью каждому из Parallax Inc.

Перевод: Александр Касьяненко.

Содержание

ВВЕДЕНИЕ 15

ГЛАВА 1 : ПРЕДСТАВЛЯЕМ ИМС PROPELLER. 17

Общие понятия. 17
Типы корпусов 18
Описание выводов 19
Технические характеристики 20
Внешние соединения 21
Начальная загрузка 22
Исполнение приложения 22
Режим Останова 23
Блок схема 24
Разделяемые ресурсы 26
Системный генератор 26
Процессоры (*Cogs*) 26
Переключатель (*Hub*) 29
Линии В/В 30
Системный Счетчик 32
Регистр CLK 33
Биты защиты 35
Основная память 36
Основное ОЗУ 37
Основное ПЗУ 37
Знакогенератор 37
Таблицы Log и Anti-Log 40
Таблица Sin 40
Загрузчик и интерпретатор Spin 40

ГЛАВА 2 : РАБОТА С ПРОГРАММОЙ PROPELLER TOOL 43

Общие положения 43
Организация Экрана 45
Пункты меню 55
Меню Файл (File) 55
Меню Редактировать (Edit) 56
Меню Выполнить (Run) 58
Меню Помощь (Help) 59
Диалог Найти/Заменить 61
Вид Объекта (Object View) 64
Информация об объекте (Object Info) 67
Таблица символов 70
Режимы просмотра, Отметки и Номера строк 74
Режимы просмотра 74
Отметки 77

Нумерация Строк 78
Режимы редактирования 79
Режимы Вставка и Замена 79
Режим Выравнивание 80
Выделение и перемещение блока 83
Отступы и Выступы 84
Одиночные Строки (Single Lines) 85
Несколько строк (Multiple Lines) 86
Индикаторы Блок-Групп 89
Сочетания клавиш (Shortcut Keys) 90
Перечень по функциям 90
Перечень по клавишам 95

ГЛАВА 3 : ПРОГРАММИРОВАНИЕ ИМС PROPELLER 101

Общие положения 101
Языки ИМС Propeller (Spin и Propeller Ассемблер) 102
Объекты Propeller 102
Кратко: Введение 109
Упражнение 1: Output.spin – Наш первый объект 109
Разница между загрузкой в ОЗУ и ЭСППЗУ 110
Кратко: Упр. 1 114
Процессоры (*Cogs*) 115
Упражнение 2: Output.spin - Константы 116
Указатели Блоков 117
Упражнение 3: Output.spin - Комментарии 118
Кратко: Упр. 2 и 3 121
Упражнение 4: Output.spin – Параметры, Вызовы и Конечные Циклы 122
Упражнение5: Output.spin – Параллельное Выполнение 125
Кратко: Упр. 4 и 5 128
Упражнение 6: Output.spin и Blinker1.spin – Используем наш объект 129
Вид объекта 131
Верхний Объектный Файл 132
Какие объекты были откомпилированы? 134
Кратко: Упр. 6 135
Объекты и Процессоры 136
Упражнение 7: Output.spin – Совершенствуем далее 136
Кратко: Упр. 7 143
Упражнение 8: Blinker2.spin – Много объектов, много процессоров 144
Окно Информации об Объекте 149
Время жизни объекта 151
Кратко: Упр. 8 152
Упражнение 9: Установки генератора 153
Упражнение 10: Временные соотношения 155

Кратко: Упр. 9 и 10 ... 160
Упражнение 11: Библиотечные Объекты ... 161
Рабочие и Библиотечные папки ... 164
Упражнение 12: Целые и вещественные числа ... 167
Псевдо-вещественные числа ... 167
Формат с плавающей запятой ... 168
Контекстно-зависимая информация компиляции ... 171
Кратко: Упр.11 и 12 ... 172

ГЛАВА 4 : СПРАВОЧНИК ПО ЯЗЫКУ SPIN ... 173

Структура Объектов Propeller ... 174
Перечень элементов языка Propeller Spin по категориям ... 176
Указатели блоков ... 176
Конфигурация ... 176
Управление Процессорами ... 177
Управление Процессом ... 177
Управление потоками ... 177
Память ... 178
Директивы ... 179
Регистры ... 179
Константы ... 180
Переменные ... 180
Унарные операции ... 180
Бинарные операции ... 181
Символы синтаксиса ... 182
Элементы языка Spin ... 183
Правила Идентификаторов ... 183
Представление величин ... 183
Правила Синтаксиса ... 185
ABORT ... 187
BYTE ... 192
BYTEMOVE ... 199
CASE ... 200
CHIPVER ... 203
CLKFREQ ... 204
_CLKFREQ ... 206
CLKMODE ... 208
_CLKMODE ... 209
CLKSET ... 213
CNT ... 215
COGID ... 217
COGINIT ... 218
COGNEW ... 221

COGSTOP 227
CON 228
CONSTANT 235
Предопределенные Константы 237
CTRA, CTRB 239
DAT 243
DIRA, DIRB 249
FILE 252
FLOAT 253
_FREE 255
FRQA, FRQB 256
IF 257
IFNOT 263
INA, INB 264
LOCKCLR 266
LOCKNEW 268
LOCKRET 271
LOCKSET 272
LONG 274
LONGFILL 281
LONGMOVE 282
LOOKDOWN, LOOKDOWNZ 283
LOOKUP, LOOKUPZ 285
NEXT 287
OBJ 288
Операторы Spin 291
OUTA, OUTB 326
PAR 330
PHSA, PHSB 332
PRI 333
PUB 334
QUIT 338
REBOOT 339
REPEAT 340
RESULT 347
RETURN 349
ROUND 351
SPR 353
_STACK 355
STRCOMP 356
STRING 358
STRSIZE 359
Символы 360

TRUNC ... 363
VAR ... 364
VCFG ... 368
VSCL ... 371
WAITCNT ... 373
WAITPEQ ... 377
WAITPNE ... 379
WAITVID ... 380
WORD ... 382
WORDFILL ... 390
WORDMOVE ... 391
_XINFREQ ... 392

ГЛАВА 5 : СПРАВОЧНИК ПО ЯЗЫКУ АССЕМБЛЕРА ... 394

Структура ассемблера Propeller ... 394
Память Процессора ... 396
Где инструкция берет свои данные? ... 396
Не забывайте символ константы '#' ... 397
Константы должны умещаться в 9-ти битах ... 398
Глобальные и локальные метки ... 398
Перечень элементов Propeller ассемблер по категориям ... 400
Директивы ... 400
Конфигурация ... 400
Управление процессором ... 400
Управление процессами ... 400
Условные операторы ... 400
Управление потоком ... 402
Воздействия ... 402
Доступ к Основной Памяти ... 402
Общие операции ... 402
Регистры ... 404
Константы ... 405
Унарные операторы ... 405
Бинарные операторы ... 406
Элементы языка ассемблер ... 408
Определения синтаксиса ... 408
Сводная таблица инструкций языка Propeller ассемблер ... 412
ABS ... 416
ABSNEG ... 417
ADD ... 417
ADDABS ... 418
ADDS ... 420
ADDSX ... 421

ADDX 423
AND 425
ANDN 426
CALL 427
CLKSET 429
CMP 431
CMPS 432
CMPSUB 433
CMPSX 434
CMPX 438
CNT 440
COGID 441
COGINIT 442
COGSTOP 444
Условия (IF_x) 446
CTRA, CTRB 447
DIRA, DIRB 448
DJNZ 449
Воздействия 450
FRQA, FRQB 451
FIT 453
HUBOP 454
IF_x 455
INA, INB 457
JMP 458
JMPRET 459
LOCKCLR 463
LOCKNEW 465
LOCKRET 466
LOCKSET 467
MAX 468
MAXS 469
MIN 470
MINS 470
MOV 471
MOVD 472
MOVI 473
MOVS 474
MUXC 475
MUXNC 476
MUXNZ 477
MUXZ 478
NEG 479

NEGC ... 480
NEGNC ... 481
NEGNZ ... 482
NEGZ ... 483
NOP ... 483
NR ... 484
Операторы ... 486
OR ... 488
ORG ... 488
OUTA, OUTB ... 490
PAR ... 491
PHSA, PHSB ... 492
RCL ... 494
RCR ... 494
RDBYTE ... 495
RDLONG ... 496
RDWORD ... 498
Регистры ... 499
RES ... 501
RET ... 504
REV ... 504
ROL ... 505
ROR ... 506
SAR ... 507
SHL ... 509
SHR ... 509
SUB ... 510
SUBABS ... 512
SUBS ... 513
SUBSX ... 514
SUBX ... 516
SUMC ... 518
SUMNC ... 519
SUMNZ ... 520
SUMZ ... 521
Символы ... 522
TEST ... 523
TESTN ... 524
TJNZ ... 525
TJZ ... 526
VCFG ... 526
VSCL ... 527
WAITCNT ... 528

WAITPEQ 529
WAITPNE 530
WAITVID 531
WC 532
WR 533
WRBYTE 534
WRLONG 535
WRWORD 535
WZ 536
XOR 537

ПРИЛОЖЕНИЕ A: СПИСОК СЛУЖЕБНЫХ СЛОВ 539

ПРИЛОЖЕНИЕ B: МАТЕМАТИЧЕСКИЕ ПРИМЕРЫ И ТАБЛИЦЫ ФУНКЦИЙ 540

ИНДЕКС 547

Введение

Благодарим Вас за приобретение интегральной микросхемы Propeller (ИМС Propeller). Вы будете разрабатывать свои собственные программы в считанные мгновения!

Микросхемы Propeller – невероятно мощные мультипроцессорные микроконтроллеры, это долгожданный результат более чем восьми лет интенсивной работы Chip Gracey и всей команды разработчиков Parallax.

Эта книга призвана стать справочным руководством по микросхемам Propeller и их собственным языкам программирования: *Spin* и *Propeller* Ассемблер. Для прохождения курса по программированию, а также уточнения деталей работы с *Propeller Tool*, пожалуйста, обращайтесь к файлам онлайн-помощи, которые устанавливаются на Ваш компьютер при инсталляции программы *Propeller Tool*. Удачи!

Несмотря на все наши усилия, невозможно осветить все вопросы в рамках одного справочного руководства. Посетите наш форум Propeller chip (доступный на www.parallax.com, через меню «Support» → «Discussion Forums»). Эта ветка предназначена специально для пользователей ИМС Propeller, здесь Вы можете оставлять свои вопросы, а также просматривать обсуждения, в которых, возможно, уже могут быть на них ответы.

Кроме форума, посетите сайт Propeller Object Exchange (www.obex.parallax.com) для получения свободного доступа к сотням программных Propeller-объектов, написанных как пользователями, так и инженерами Parallax. Кроме случаев их непосредственного использования в Ваших собственных приложениях, сами эти объекты, написанные различными авторами, представляют собой богатый ресурс для изучения различных техник и приемов программирования, используемых очень активным сообществом пользователей ИМС Propeller.

Глава 1: Представляем ИМС Propeller

Эта глава описывает аппаратную часть ИМС Propeller. Для полного понимания и эффективного использования чипа, важно изначально понять его архитектуру. В этой главе описываются детали архитектуры, такие как типы и размеры корпусов, описание выводов и их функции.

Общие понятия

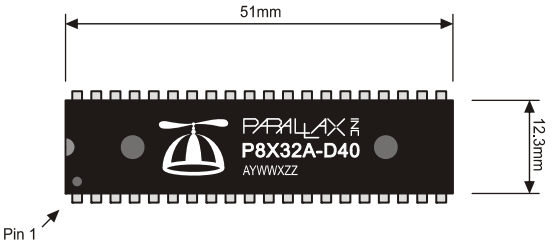
Интегральная микросхема Propeller разработана для обеспечения высокоскоростной обработки данных во встраиваемых системах, совмещая при этом одновременно малое потребление и малые размеры корпуса. Вдобавок к своему быстродействию, ИМС Propeller обеспечивает гибкость и производительность за счет своих восьми процессоров-ядер, так называемых «*Cog*» (от англ. слова «*Cog*» - «зубец шестеренки»). Эти процессоры могут одновременно выполнять независимые либо совместные задачи, обладая в то же время относительно простой архитектурой, которую легко освоить и использовать.

Проектирование систем на основе ИМС Propeller освобождает разработчиков от общих сложностей, присущих программированию встроенных систем. Например:

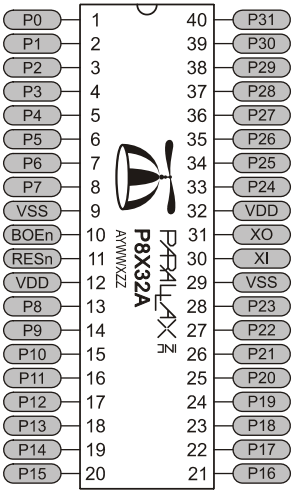
- Карта памяти линейна. Нет необходимости использования схем страничной организации с блоками кода, данных, либо переменных. Это позволяет значительно сократить время разработки.
- Асинхронные процессы обрабатывать проще, чем в устройствах, использующих для этих целей прерывания. Propeller не нуждается в прерываниях; необходимо лишь назначить некоторым из *cog*-ов свои собственные высоко-скоростные задачи, в то время как остальные будут иметь полностью свободные ресурсы. В результате имеем систему с малым временем отклика, которую легче адаптировать.
- Мощный язык ассемблера поддерживает условное выполнение и опциональную запись результата для каждой инструкции. Это позволяет обеспечить четкое временное согласование в критических блоках кода со многими ветвлениями; обработчики событий менее склонны к джиттеру и разработчики тратят меньше времени на подсчет и подгонку количества циклов тут и там.

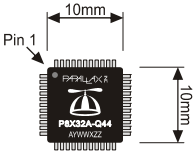
Типы корпусов

ИМС Propeller выпускается в следующих типах корпусов.

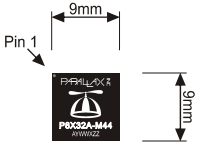


P8X32A-D40 ***40-выв. DIP***

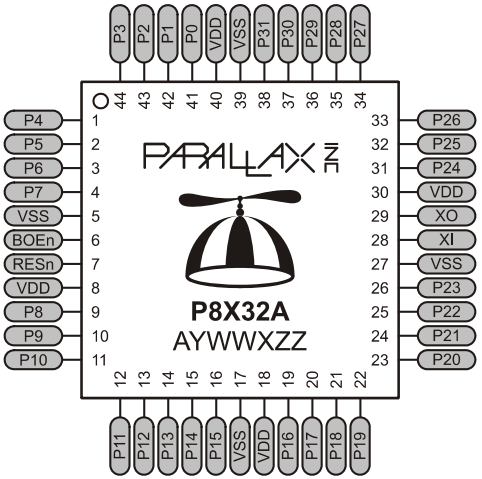




P8X32A-Q44
44-выв. LQFP



P8X32A-M44
44-выв. QFN



Описание выводов

Табл. 1-1: Описание выводов

Имя вывода	Направление	Описание
P0 – P31	I/O	Порт A. Порт ввода/вывода общего назначения. Каждый вывод может выдавать/потреблять ток 40мА при 3.3В DC. Логический пороговый уровень составляет ≈ ½ VDD; 1.65VDC @ 3.3В VDD. Выводы, указанные ниже, имеют специальные функции при включении питания либо сбросе, однако затем они так же являются портами ввода/вывода общего назначения. P28 - I2C SCL, линия для внешней ЭСППЗУ (опционально). P29 - I2C SDA, линия для внешней ЭСППЗУ (опционально). P30 - TX, линия последовательной передачи к хосту. P31 - RX, линия последовательного приема от хоста.
VDD	---	3.3В напряжение питания (2.7 – 3.3VDC).
VSS	---	Земля (общий).
BOEn	I	Brown Out Enable (низкий уровень активный). Должен быть соединен либо к VDD, либо к VSS. При низком уровне на BOEn, вывод RESn становится высокоомным выходом (подтянутым к VDD через 5 KΩ) для целей мониторинга, но приводит к сбросу при установке в ноль. При высоком уровне на BOEn, вывод RESn является CMOS входом с триггером Шмитта.
RESn	I/O	Сброс (низкий уровень активный). При низком уровне происходит сброс ИМС Propeller: все *Cog-и* блокированы и входы/выходы находятся в третьем состоянии. Propeller перестартует через 50 мсек после перехода RESn из нижнего уровня в верхний.
XI	I	Вход для кварца. Может быть подключен к выходу осциллятора (при этом XO не подключается), либо к одному из выводов кварцевого резонатора (XO подключается к другому выводу резонатора), в зависимости от установок в регистре CLK. Внешние резисторы либо конденсаторы не нужны.
XO	O	Выход для кварца. Обеспечивает обратную связь для внешнего резонатора, либо не подключается, в зависимости от установок в регистре CLK . Внешние резисторы либо конденсаторы не нужны.

ИМС Propeller (P8X32A) имеет 32 линии ввода/вывода (Порт A, линии с P0 по P31). Четыре из этих линий, P28-P31, имеют специальные функции при включении питания

либо сбросе. При включении питания либо возникновении сигнала сброса, по линиям P30 и P31 происходит связь с хостом для программирования, а по P28 и P29 производится доступ к внешней 32 кБ ЭСППЗУ (24LC256).

Технические характеристики

Табл. 1-2: Технические характеристики	
Модель	P8X32A
Напряжение питания	3.3В DC (Максимальное потребление тока должно быть ограничено до 300 мА)
Частота внешнего кварца	От 0 до 80 МГц (от 4 МГц до 8 МГц при работе с PLL)
Системная частота	От 0 до 80 МГц
Внутренний RC-генератор	12 МГц или 20 кГц (приблизительно; может меняться в пределах 8 МГц – 20 МГц, или 13 кГц – 33 кГц, соответственно)
Основное ОЗУ/ПЗУ	64 кБ; 32 кБ ОЗУ + 32 кБ ПЗУ
ОЗУ *Cog*	512 x 32 бит на каждый
Организация ОЗУ/ПЗУ	Адресуемая как Long (32-бита), Word (16-бит), или Byte (8-бит)
Линии ввода/вывода	32 CMOS сигнала с логическим порогом VDD/2.
Выдаваемый/потребляемый ток на один вывод	40 мА
Потребление тока @ 3.3В, 70 °F	500 мкА на MIPS (1MIPS = Частота в МГц / 4 * Количество активных *Cog*)

Внешние соединения

На Рис. 1-1 приведен пример схемы подключения, которая обеспечивает связь хоста и ЭСППЗУ с ИМС Propeller. В этом примере доступ к хосту осуществляется через устройство *Propeller Clip* (преобразователь последовательного интерфейса USB в TTL).

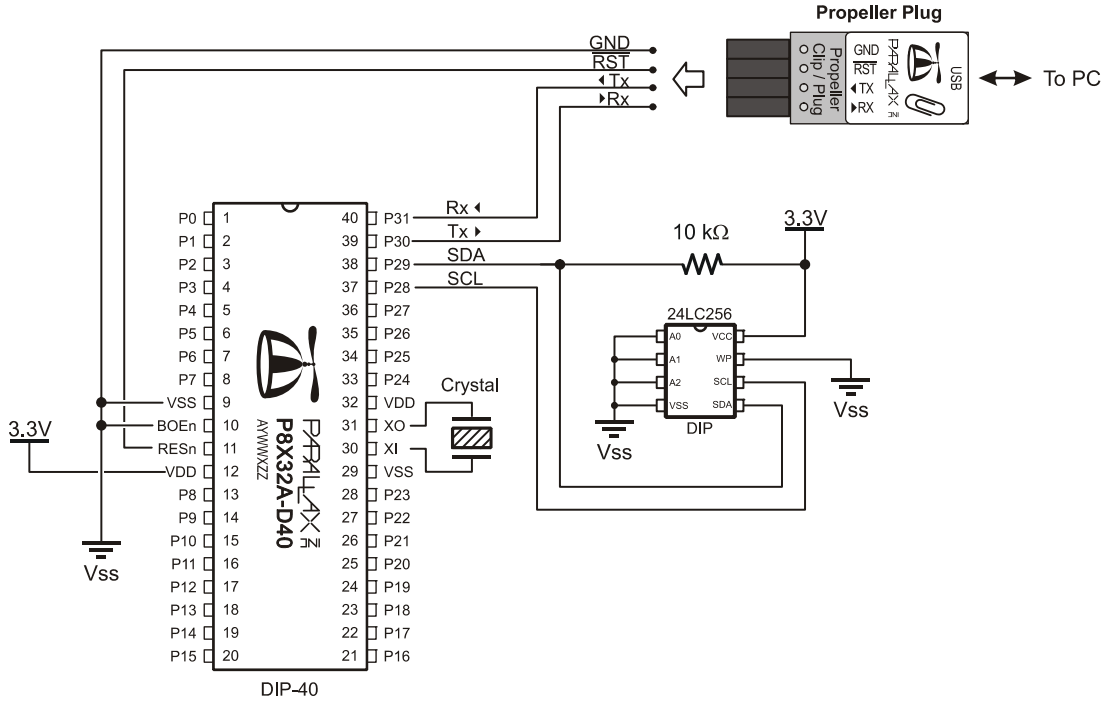


Рис. 1-1: Пример схемы подключения, позволяющей программировать ИМС Propeller и внешнюю 32 Кбайт ЭСППЗУ, и обеспечивающей работу ИМС Propeller с внешним кварцевым резонатором.

Начальная загрузка

По включению питания (+100 мсек), возникновению сигнала Сброс (переходе на входе RESn из низкого состояния в высокое), либо при программном сбросе:

1. ИМС Propeller запускает внутренний генератор в медленном режиме (≈ 20 кГц), выжидает порядка 50 мсек (задержка сброса), переключает внутренний генератор в быстрый режим (≈ 12 МГц), и после этого загружает и выполняет встроенную программу загрузчика в первом процессоре (*Cog0*).
2. Загрузчик выполняет одну или более из нижеследующих задач, в таком порядке:
    a. Выполняет установление связи с хостом (например, ЭВМ), на линиях P30 и P31. Если связь с хостом установлена, загрузчик обменивается данными с хостом, чтобы идентифицировать ИМС Propeller и, при возможности, загрузить программу в основное ОЗУ и, опционально, ─ во внешнюю 32 кБ ЭСППЗУ.
    b. Если связь с хостом не была установлена, загрузчик обращается к внешней 32 кБ ЭСППЗУ (24LC256) по линиям P28 и P29. Если ЭСППЗУ обнаружена, весь 32 кБ массив загружается в основное ОЗУ ИМС Propeller.
    c. Если ЭСППЗУ не была обнаружена, загрузчик останавливает выполнение, *Cog0* блокируется, ИМС Propeller переходит в отключенное состояние и настраивает все линии на ввод.
3. Если какой-либо из шагов 2a или 2b был успешным, программа была загружена в основное ОЗУ, и хост не передал команды «Заснуть», - процессор *Cog0* перегружается, загружает встроенный интерпретатор *Spin*, и программа пользователя выполняется из основного ОЗУ.

Исполнение приложения

Приложение для ИМС Propeller – это программа пользователя, откомпилированная в двоичный вид и загруженная в ОЗУ микросхемы и, возможно, во внешнюю ЭСППЗУ. Приложение состоит из кода, написанного на языке Propeller *Spin* (код верхнего уровня) с возможностью подключения компонентов на языке ассемблера для Propeller (код нижнего уровня). Код, написанный на языке *Spin*, интерпретируется во время выполнения процессором (*Cog*) с запущенным *Spin*-интерпретатором, в то время как код, написанный на языке ассемблера, выполняется в своем исходном виде

непосредственно в *Cog*. Каждое приложение для Propeller состоит, как минимум, из небольшого *Spin*-кода, а в общем случае может быть написано полностью на *Spin* либо с различными комбинациями *Spin* и ассемблера. Интерпретатор языка *Spin* запускается на шаге 3 процедуры загрузки, описанной выше, и обеспечивает выполнение приложения.

После того, как процедура загрузки завершена и приложение запущено в процессоре *Cog0*, все дальнейшие действия определяются самим приложением. Приложение имеет полный контроль над такими параметрами, как внутренняя частота, использование линий ввода/вывода, регистров конфигурации, а также тем, когда, какие и сколько процессоров (*Cog*-ов) запущены в любой момент времени. Все это, включая внутреннюю частоту, может изменяться в процессе исполнения, поскольку контролируется приложением. См. Глава 3: Программирование ИМС Propeller .

Режим Останова

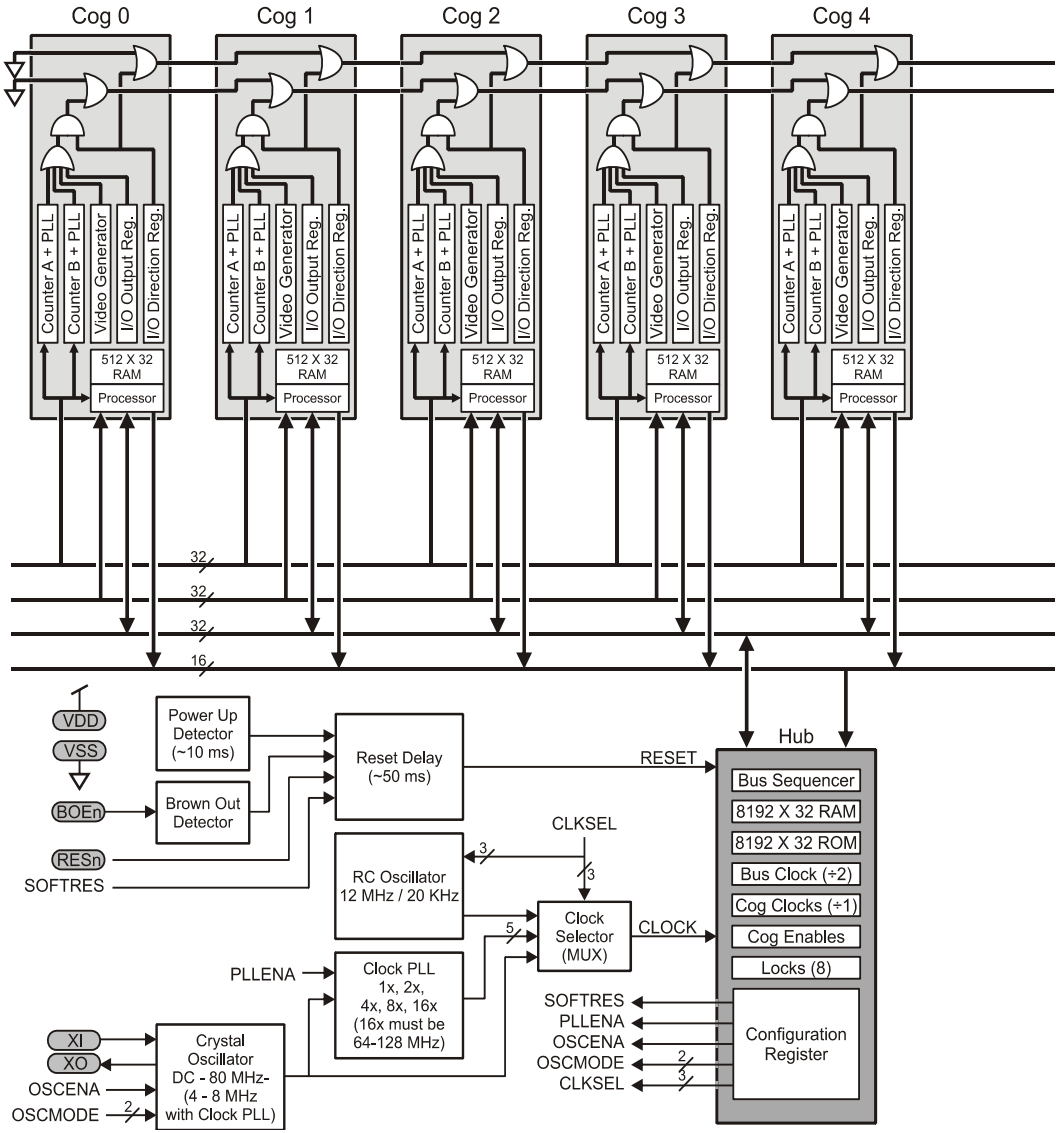
Когда ИМС Propeller переходит в режим «Останов», внутренняя генерация прекращается, что приводит к остановке всех процессоров и переводу всех линий на ввод (высокий импеданс). Режим останова инициируется одним из трех событий:

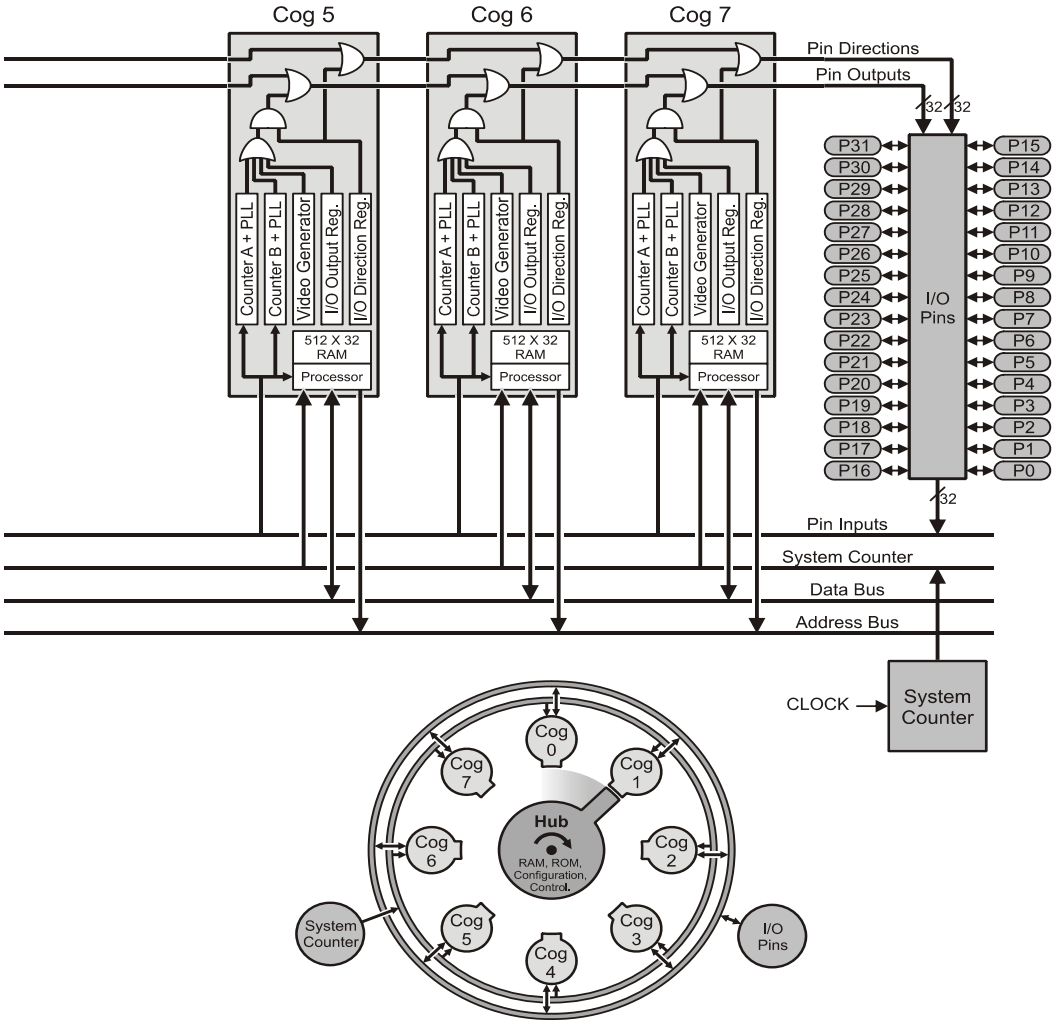
1) VDD опускается ниже порога засыпания (≈2.7В), при включенной цепи детектора засыпания,
2) сигнал на линии RESn переходит в низкий уровень, или
3) приложение запрашивает перегрузку (см. команду `REBOOT`, стр. 339).

Режим «Останов» прерывается, когда напряжение питания поднимается выше порога срабатывания схемы детектора засыпания, а также на линии RESn присутствует сигнал высокого уровня.

Блок схема

Рис. 1-2: Блок-схема ИМС Propeller





Hub and Cog Interaction

Очень важным для ИМС Propeller является взаимодействие процессоров *Cog* и Переключателя *Hub* (перев. с англ. «вал»). Именно *Hub* контролирует, какой из Cog может получить доступ к взаимоисключающим ресурсам, таким, как основное ОЗУ/ПЗУ, регистры конфигурации и т.д. *Hub* предоставляет отдельный доступ каждому *Cog* в каждый момент времени по круговой схеме (“*round robin*”), независимо от того, сколько *Cog* в данный момент запущено, поддерживая, таким образом, синхронизацию.

Разделяемые ресурсы

В ИМС Propeller имеется два типа разделяемых ресурсов: 1)общие, и 2) взаимоисключающие. Общие ресурсы могут использоваться в любое время любым количеством *Cog*-ов. Взаимоисключающие ресурсы могут также быть доступны каждому из *Cog*-ов, но только по одному *Cog* в каждый момент времени. Общие разделяемые ресурсы – это линии ввода/вывода и Системный Счетчик (System Counter). Все другие разделяемые ресурсы являются взаимоисключающими по своей природе, и доступ к ним контролирует Переключатель *Hub*. См. секцию Переключатель (*Hub*) на странице 29.

Системный генератор

Системный генератор (показанный на Рис. 1-2 как “CLOCK”) – это главный источник синхронизации для практически каждого компонента ИМС Propeller. Сигнал частоты Системного Генератора получают от одного из трех возможных источников: 1) внутреннего RC-Генератора, 2) блока умножителя ФАПЧ (PLL), или 3) генератора на кварцевом резонаторе (внутренняя цепь, подключенная к кварцевому резонатору либо осциллятору). Источник определяется установкой регистра CLK, который доступен как во время компиляции, так и во время исполнения приложения. Единственными компонентами, не использующими системную частоту напрямую, являются переключатель *Hub* и шина *Bus*; для синхронизации они используют системную частоту, разделенную на два (2).

Процессоры (*Cogs*)

В состав ИМС Propeller входит восемь процессоров (вычислительных ядер), называемых *Cog*, с номерами от 0 до 7. Каждый *Cog* включает в себя следующие компоненты (см. Рис. 1-2): Блок Процессора (Processor block), внутреннее 2 кБ ОЗУ (2 KB RAM) с организацией 512 двойных слов (512 x 32 бит), два Модуля Счетчиков (Counter Modules) со своими цепями ФАПЧ (PLL’s), Генератор Видеосигнала (Video Generator), Выходной Регистр ввода/вывода (I/O Output Register), Регистр Направления ввода/вывода (I/O Direction Register), и другие регистры, не указанные на блок-схеме. Полный перечень регистров процессоров приведен в Табл. 1-3. Все *Cog*-и выполнены абсолютно одинаковыми и могут выполнять задачи независимо друг от друга.

Все восемь процессоров тактируются от одного источника системной частоты, поэтому у каждого из них одна и та же временная база; все активные *Cog*-и выполняют инструкции одновременно (см. выше, Системный генератор). Все они так же имеют

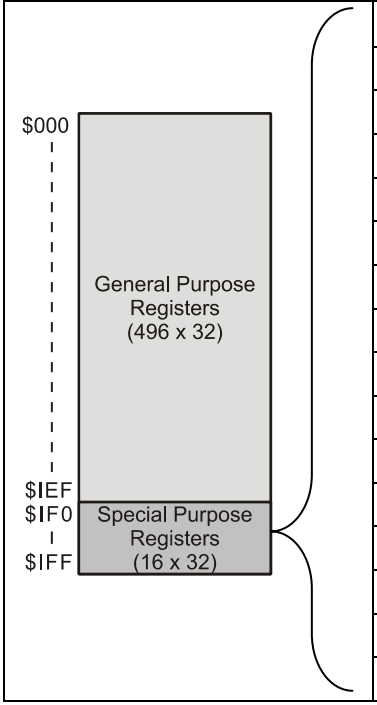
доступ к одним и тем же аппаратным ресурсам, таким как линии ввода/вывода, Основное ОЗУ и Системный Счетчик (см. выше Разделяемые ресурсы).

Каждый процессор может быть запущен либо остановлен во время выполнения приложения, может быть запрограммирован для одновременного выполнения совместных задач: либо независимо, либо скоординировано с другими *Cog* через Основное ОЗУ. Независимо от природы использования *Cog*-ов, разработчик приложений для ИМС Propeller обладает полным контролем над тем, как и когда каждый из процессоров будет задействован; ни компилятор, ни какая-либо операционная система не занимаются разделением задач между *Cog*-ами. Такое построение системы дает разработчику возможность абсолютно четкого согласования процессов во времени, контроля потребления и отклика на внешние события при разработке встраиваемых систем.

Каждый *Cog* имеет свое собственное ОЗУ, называемое *Cog RAM*, которое состоит из 512-ти 32-х битных регистров. Всё ОЗУ (*Cog RAM*) является памятью общего назначения, кроме последних 16-ти регистров – Регистров Специальных Функций (РСФ), которые описаны в Табл. 1-3. Память *Cog RAM* используется для хранения исполнимого кода, данных и переменных, а на последние 16 адресов отображены Системный Счетчик, линии ввода/вывода и локальная периферия *Cog*-а.

При загрузке *Cog*-а, адреса с 0 ($000) до 495 ($1EF) последовательно загружаются из Основной ОЗУ/ПЗУ, а его регистры специальных функций, с адресами от 496 ($1F0) до 511 ($1FF) обнуляются. После загрузки, *Cog* начинает исполнение инструкций, начиная с адреса $000 в его ОЗУ (*Cog RAM*). Он продолжает выполнять код то тех пор, пока не будет остановлен, перегружен самим собой или другим *Cog*-ом, либо пока не возникнет сигнал Сброс.

Табл. 1-3: Регистры специальных функций в ОЗУ *Cog*

Карта *Cog RAM*	Адрес	Имя	Тип	Описание
	$1F0	PAR	Чтение[1]	Параметр загрузки, стр. 330
	$1F1	CNT	Чтение [1]	Системный Счетчик,стр.215
	$1F2	INA	Чтение [1]	Состояние входов P31 - P0
	$1F3	INB [3]	Чтение [1]	Состоян. входов P63- P32[2]
	$1F4	OUTA	Чтение/Запись	Значения выходов P31 - P0
	$1F5	OUTB [3]	Чтение/Запись	Значен. выходов P63 – P32[2]
	$1F6	DIRA	Чтение/Запись	Направление P31 - P0
	$1F7	DIRB [3]	Чтение/Запись	Направление P63 - P32[2]
	$1F8	CTRA	Чтение/Запись	Управление счетчиком A
	$1F9	CTRB	Чтение/Запись	Управление счетчиком B
	$1FA	FRQA	Чтение/Запись	Частота счетчика A
	$1FB	FRQB	Чтение/Запись	Частота счетчика B
	$1FC	PHSA	Чтение/Запись [2]	ФАПЧ счетчика A
	$1FD	PHSB	Чтение/Запись [2]	ФАПЧ счетчика B
	$1FE	VCFG	Чтение/Запись	Настройка Видео
	$1FF	VSCL	Чтение/Запись	Масштаб Видео

Прим. 1: Для ассемблера, доступен только как регистр-источник (т.е. `MOV DEST, SOURCE`**)**
См. разделы языка Ассемблер для PAR, стр. 491; CNT, стр. 440, и INA, INB, на стр. 457.
Прим. 2: Для ассемблера, только для чтения как регистр-источник (т.е. `MOV DEST, SOURCE`**)**
См. разделы языка Ассемблер для PHSA, PHSB на стр. 492.
Прим. 3: Зарезервированы для дальнейшего использования.

Любой из Регистров Специальных Функций может быть доступен через:

1) Его физический адрес (Propeller-ассемблер),
2) Его предопределенное имя (*Spin* или Propeller-ассемблер), или
3) Переменную типа массив (РСФ) с индексом от 0 до 15 (*Spin*).

Пример на языке ассемблера Propeller:

```
MOV   $1F4, #$FF    'Set OUTA 7:0 high
MOV   OUTA, #$FF    'Same as above
```

Пример на языке Propeller *Spin*:

```
SPR[$4] := $FF      'Set OUTA 7:0 high
OUTA := $FF         'Same as above
```

Переключатель (*Hub*)

Для обеспечения целостности системы, взаимоисключающие ресурсы не должны быть доступны более чем одному процессору в одно и то же время. Именно переключатель *Hub* обеспечивает такую целостность путем управления доступом к взаимоисключающим ресурсам, организуя очередь для доступа к ним между процессорами от *Cog0* до *Cog7* и заново от *Cog0* по круговой схеме "*round robin*". Переключатель *Hub* и шина *Bus*, которой он управляет, работают на половинной (от системной) частоте. Это значит, что *Hub* предоставляет *Cog*-у доступ к взаимоисключающим ресурсам один раз на каждые 16 системных циклов. *Hub*-инструкции, – инструкции ассемблера Propeller, которые требуют доступ к взаимоисключающим ресурсам, – требуют для своего выполнения 7 циклов, но сначала им необходимо быть синхронизированными с началом Окна Доступа к Переключателю (*Hub Access Window*). Синхронизация с Окном Доступа к Переключателю занимает до 15 циклов: 16 минус 1, если окно только что прошло, плюс 7 циклов ─ для запуска *Hub*-инструкции. Таким образом, для выполнения *Hub*-инструкции необходимо от 7 до 22 циклов.

На Рис. 1-3 и Рис. 1-4 показаны примеры, где *Cog0* получил *Hub*-инструкцию для выполнения. На Рис. 1-3 показан наилучший вариант: *Hub*-инструкция была получена как раз в начале окна доступа для этого *Cog*-а. *Hub*-инструкция выполняется без задержки (7 циклов), оставляя дополнительные 9 циклов для выполнения этим *Cog*-ом других инструкций перед появлением следующего Окна Доступа к Переключателю.

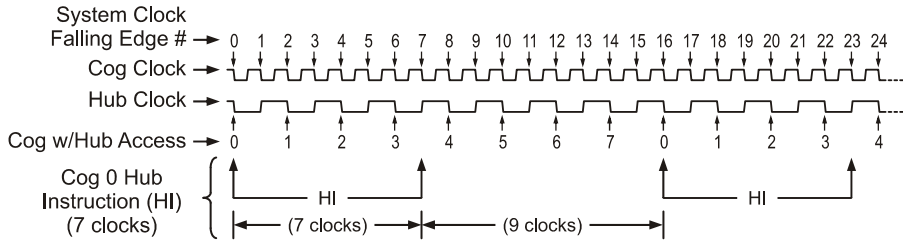


Рис. 1-3: Взаимодействие *Cog* и *Hub* – наилучший вариант

На Рис. 1-3 показана наихудшая ситуация, когда *Hub*-инструкция пришла на следующий цикл после начала окна доступа для *Cog0*: *Cog* только что пропустил окно. Процессор ждет следующего окна доступа (15 циклов), после чего выполняет инструкцию доступа к *Hub*-у (7 циклов), что в сумме требует 22 цикла для выполнения этой *Hub*-инструкции. Здесь так же остаются дополнительные 9 циклов для выполнения этим *Cog*-ом других инструкций перед появлением следующего Окна Доступа к Переключателю. Для получения максимальной эффективности от ассемблерных процедур, которые требуют частого доступа к взаимоисключающим ресурсам, полезно чередовать *Hub*-инструкции с инструкциями, не требующими доступа к *Hub*, чтобы уменьшить количество циклов ожидания следующего окна доступа. Поскольку большинство ассемблерных инструкций выполняются за 4 цикла, между смежными *Hub*-инструкциями могут быть выполнены две обычные.

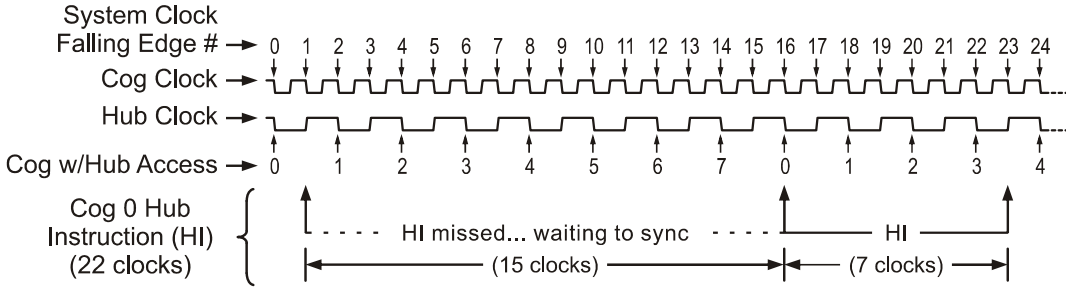


Рис. 1-4: Взаимодействие *Cog* и *Hub* – наихудший вариант

Необходимо помнить, что *Hub*-инструкции одного *Cog*-а никак не мешают выполнению инструкций остальных процессоров благодаря архитектуре самого Переключателя. Например, *Cog1* может начать выполнение *Hub*-инструкции во время цикла 2 Системной Частоты с возможным перекрытием выполнения в *Cog0* обычной инструкции, без никаких вредных эффектов. В это время все остальные процессоры могут продолжать выполнение обычных, не-*Hub* инструкций, либо ожидать их индивидуальных Окон Доступа к Переключателю независимо от того, чем заняты остальные.

Линии В/В

ИМС Propeller имеет 32 линии ввода/вывода, 28 из которых являются полноценными портами общего назначения. Четыре линии ввода/вывода (28 - 31) имеют особое назначение во время Начальной Загрузки, но доступны как порты общего назначения

после нее; см. секцию Начальная загрузка на стр. 22. После загрузки, любое количество линий ввода/вывода может быть использовано любым *Cog* в любой момент времени, поскольку линии ввода/вывода являются одним из общих ресурсов. Разработчик приложения должен сам следить за тем, чтобы во время выполнения программы любые два процессора не приводили к коллизиям на одной и той же линии ввода/вывода.

Для понимания аппаратной реализации линий ввода/вывода, при дальнейшем чтении необходимо обращаться к описанию архитектуры Cog-ов, см. Рис. 1-2 на стр. 24.

Каждый *Cog* имеет свой собственный 32-битный Регистр Направления и 32-битный Регистр Выхода. Биты этих регистров влияют на направление и состояние выходов соответствующих линий ввода/вывода микросхемы. Желаемые значения направлений и состояния выходов каждого *Cog*-а проходят через всю группу *Cog*-ов, чтобы, в конце концов, стать тем, что обозначено как "Направления линии" ("Pin Directions") и "Выходы линии" ("Pin Outputs") в правой верхней части Рис.1-2 стр. 24.

Группа *cog*-ов определяет Направление линии и Выход линии следующим образом:

1. Направление линии является результатом сложения по ИЛИ Регистров Направления всех *cog*-ов.
2. Выход линии является результатом сложения по ИЛИ состояний выходов этой линии у всех *cog*-ов. Состояние выхода линии каждого из *cog*-ов определяется значениями соответствующих битов его внутренних модулей ввода/вывода (Счетчиков, Видеогенератора и Регистра Выхода), сложенных вместе по ИЛИ, а затем – умноженных по И на соответствующие биты его Регистра Направления.

По существу, направление и состояние выхода каждой линии ввода/вывода соединены «монтажным ИЛИ» всей группы процессоров. Это позволяет *Cog*-ам иметь доступ и влиять на линии ввода/вывода одновременно, при этом исключая необходимость в любого рода арбитре доступа к ресурсам при отсутствии электрических соединений между самими процессорами.

Результат такой структуры соединений линий ввода/вывода может быть легко описан следующими простыми правилами:

> A. Вывод является входом только если ни один из активных *Cog*-ов не устанавливает его как выход.
>
> B. На выводе будет низкий уровень только в том случае, когда все активные *Cog*-и, которые установили его как выход, установят его в ноль.
>
> C. На выводе будет высокий уровень, если любой из активных *Cog*-ов, установивших его выходом, установит его в единицу.

В Табл. 1-4 показаны некоторые из возможных вариантов воздействия группы процессоров на отдельную линию ввода/вывода (в этом примере это P12). Для упрощения, в этих примерах считается, что аппаратный бит ввода/вывода 12 каждого *Cog*-а (а не бит его Выходного Регистра) сброшен в ноль (0).

Табл. 1-4: Примеры использования линий B/B

	Бит 12 Регистра Направления *Cog*	**Бит 12 Регистра Вывода *Cog***	**Состояние линии B/B P12**	**Правило**
Номер*Cog*	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7		
Пример 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	Вход	A
Пример 2	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	Выход =0	B
Пример 3	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	Выход =1	C
Пример 4	1 0 0 0 0 0 0 0	0 1 0 0 0 0 0 0	Выход =0	B
Пример 5	1 1 0 0 0 0 0 0	0 1 0 0 0 0 0 0	Выход =1	C
Пример 6	1 1 1 1 1 1 1 1	0 1 0 1 0 0 0 0	Выход =1	C
Пример 7	1 1 1 1 1 1 1 1	0 0 0 1 0 0 0 0	Выход =1	C
Пример 8	1 1 1 0 1 1 1 1	0 0 0 1 0 0 0 0	Выход = 0	B

Примечание: В Регистре Направления B/B значение «1» на месте бита устанавливает соответствующую линию B/B на вывод, а «0» – на ввод.

Регистр Направления и Регистр Состояния Выходов любого неактивного (отключенного) процессора устанавливаются в ноль, таким образом, исключая данный *Cog* из цепи влияния на состояние линий B/B, которые контролируются оставшимися активными процессорами.

Каждый процессор так же имеет свой собственный 32-битный Регистр Ввода. Этот входной регистр на самом деле является псевдорегистром; каждый раз, при чтении из этого регистра, возвращается реальное состояние выводов, независимо от их направления.

Системный Счетчик

Системный Счетчик – это глобальный, доступный только для чтения, 32-битный счетчик, который инкрементируется на каждом цикле Системной Частоты. Процессоры могут прочитать значение Системного Счетчика (через их регистр `CNT`, страница 215) для выполнения расчета времени и могут использовать команду `WAITCNT`

(стр. 373) для создания эффективных задержек в рамках выполнения своих задач. Системный Счетчик – это общий ресурс. Все *Cog*-и могут читать его одновременно. Он не обнуляется при старте, поскольку на практике используется для измерения разностных задержек. Если процессору необходимо узнать величину промежутка от како-го либо момента времени, ему нужно просто прочесть и сохранить начальное значение счетчика в тот момент времени и сравнить полученное после значение с сохраненным.

Регистр CLK

Регистр CLK осуществляет управление Системной Частотой, то есть он определяет ее источник и характеристики. Точнее, регистр CLK настраивает цепи RC генератора, ФАПЧ(PLL), кварцевого генератора и цепь выбора частоты (см Рис. 1-2: Блок-схема ИМС Propeller на стр. 24). Он настраивается во время компиляции объявлением константы `_CLKMODE` (стр. 208) и доступен для записи во время выполнения с помощью *Spin*-команды `CLKSET` (стр. 213) либо ассемблерной инструкции `CLKSET`.

Каждый раз, при изменении значения этого регистра, копия записанного значения должна быть помещена по адресу регистра режима генератора *Clock Mode* (доступного как BYTE[4] в основном ОЗУ), а значение устанавливаемой частоты должно быть помещено по адресу регистра частоты генератора *Clock Frequency* (доступного как LONG[0] в основном ОЗУ), для того, чтобы объекты, использующие эти параметры, всегда имели правильные данные для расчетов (см. `CLKMODE`, стр. 208 и `CLKFREQ`, стр. 204). Рекомендуется по возможности использовать команду *Spin* `CLKSET` (стр. 213), так как она автоматически обновляет все нужные регистры необходимой информацией.

Лишь определенные комбинации битов в регистре CLK подходят для установки режимов генерации. См. константу `_CLKMODE` на стр. 208 и Таблицу 4-4 на стр. 210 для дальнейшей информации. Объект «Clock» в библиотеке Propeller Library также может быть полезен, поскольку он использует методы изменения частоты и временной базы.

Табл. 1-5: Структура Регистра CLK

Бит	7	6	5	4	3	2	1	0
Имя	RESET	PLLENA	OSCENA	OSCM1	OSCM0	CLKSEL2	CLKSEL1	CLKSEL0

Табл. 1-6: Регистр CLK. Бит RESET (Бит 7)	
Бит	**Действие**
0	Всегда устанавливайте в 0, если не хотите выполнить перезагрузку контроллера.
1	Аналогичен аппаратному Сбросу – перегружает контроллер. Команда REBOOT на *Spin* записывает ‘1’ в бит RESET.

Табл. 1-7: Регистр CLK. Бит PLLENA (Бит 6)	
Бит	**Действие**
0	Отключает цепь ФАПЧ(PLL). В это значение PLLENA устанавливают константы RCFAST и RCSLOW при объявлении _CLKMODE.
1	Включает цепь ФАПЧ(PLL). Каждая из констант PLLxx при объявлении _CLKMODE устанавливает PLLENA в это значение во время компиляции. Частота с вывода XIN внутри блока ФАПЧ умножается на 16. Для передачи сигнала с XIN к цепи ФАПЧ, OSCENA должен всегда быть ‘1’. Внутренняя частота ФАПЧ должна находиться в пределах от 64 МГц до 128 МГц – это соответствует частоте на XIN от 4 МГц до 8 МГц. Перед переключением одного из выходов цепи ФАПЧ битами CLKSELx, необходимо выждать 100 мксек для ее стабилизации. После того, как цепи кварцевого генератора и ФАПЧ включены и стабилизированы, можно свободно переключаться между всеми возможными источниками, изменяя биты CLKSELx.

Табл. 1-8: Регистр CLK. Бит OSCENA (Бит 5)	
Бит	**Действие**
0	Отключает цепь кварцевого генератора. Константы RCFAST и RCSLOW при объявлении _CLKMODE настраивают OSCENA в это же значение.
1	Включает цепь кварцевого генератора, так что сигнал частоты может быть введен с XIN; либо XIN и XOUT работают вместе как генератор с обратной связью. Константы XINPUT и XTALx при объявлении _CLKMODE настраивают OSCENA таким же образом. Биты OSCMx выбирают режим работы цепи кварцевого генератора. Примечание: Для кварцевых резонаторов внешние резисторы или конденсаторы не нужны. Перед переключением источника частоты битами CLKSELx необходимо выждать 10 мсек для стабилизации генератора. Во время включения цепи кварцевого генератора, ФАПЧ может быть уже включен для минимизации времени ожидания стабилизации частоты.

Табл. 1-9: Регистр CLK. Биты OSCMx (Биты 4:3)					
OSCMx		Параметр _CLKMODE	Сопротивление XOUT	Емкость XIN/XOUT	Диапазон частот
1	0				
0	0	XINPUT	Не определено	6 pF (только емкость вывода)	Вход от 0 до 80 МГц
0	1	XTAL1	2000 Ω	36 pF	От 4 до 16 МГц Кварц/Резонатор
1	0	XTAL2	1000 Ω	26 pF	От 8 до 32 МГц Кварц/Резонатор
1	1	XTAL3	500 Ω	16 pF	От 20 до 60 МГц Кварц/Резонатор

Табл. 1-10: Регистр CLK. Биты CLKSELx (Биты 2:0)						
CLKSELx			Параметр _CLKMODE	Частота ядра	Источник	Примечания
2	1	0				
0	0	0	RCFAST	~12 МГц	Внутренний	Нет внешних компонентов. Может меняться от 8 МГц до 20 МГц.
0	0	1	RCSLOW	~20 kHz	Внутренний	Очень малое потребление. Нет внешних компонентов. Может меняться от 13 кГц до 33 кГц.
0	1	0	XINPUT	XIN	Генератор	OSCENA должен быть '1'.
0	1	1	XTALx и PLL1x	XIN x 1	Ген+ФАПЧ	OSCENA и PLLENA должны быть '1'.
1	0	0	XTALx и PLL2x	XIN x 2	Ген+ФАПЧ	OSCENA и PLLENA должны быть '1'.
1	0	1	XTALx и PLL4x	XIN x 4	Ген+ФАПЧ	OSCENA и PLLENA должны быть '1'.
1	1	0	XTALx и PLL8x	XIN x 8	Ген+ФАПЧ	OSCENA и PLLENA должны быть '1'.
1	1	1	XTALx и PLL16x	XIN x 16	Ген+ФАПЧ	OSCENA и PLLENA должны быть '1'.

Биты защиты

Для обеспечения механизма взаимоисключающего доступа процессоров к ресурсам, заданным пользователем, существует восемь битов защиты (известных также как «семафоры»). Если какой-либо блок памяти должен использоваться двумя или более процессорами одновременно, и этот блок состоит из более чем одного двойного слова

(четырех байт), то каждый из *Cog*-ов будет выполнять несколько операций чтения и записи с этим блоком для получения либо обновления данных. Это ведет к вероятной ситуации возникновения в этом блоке памяти конфликтов чтения/записи, когда один *Cog* может в него писать, а второй, в это же время, – читать. Такая ситуация приводит к ошибкам чтения либо ошибкам записи.

Биты защиты – это глобальные биты, доступные через *Hub* при помощи следующих *Hub*-инструкций: LOCKNEW, LOCKRET, LOCKSET и LOCKCLR. Поскольку эти биты доступны только через *Hub*, то в каждый момент времени повлиять на них может только один *Cog*, обеспечивая, таким образом, эффективный механизм контроля. Переключатель *Hub* имеет реестр используемых битов защиты с их текущими состояниями, и процессоры по необходимости могут их запросить, инвертировать, установить либо сбросить во время выполнения. Для более детальной информации см. LOCKNEW, 268; LOCKRET, 271; LOCKSET, 272; и LOCKCLR, 266.

Основная память

Основная память – это блок памяти объемом 64 кБ (16к x 32бита), который доступен всем процессорам как взаимоисключающий ресурс – через Переключатель. Основная память состоит из 32 кБ основного ОЗУ (RAM) и 32 кБ основного ПЗУ (ROM). Блок основного ОЗУ является памятью общего назначения, а также месторасположением приложения, загруженного из хоста, либо загруженного с внешней 32 кБ ЭСППЗУ. Блок основного ПЗУ содержит код и данные, очень важные для функционирования ИМС Propeller: наборы символов (знакогенератор), таблицы *log*, *anti-log* и *sin*, а так же загрузчик и интерпретатор языка *Spin*. Организация основной памяти показана на Рис. 1-5.

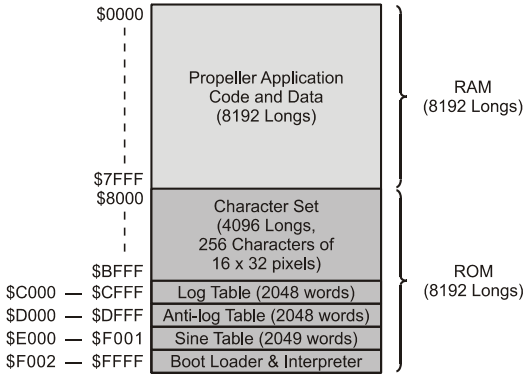


Рис. 1-5: Карта основной памяти

Основное ОЗУ

Первая половина основной памяти – это ОЗУ. Это пространство используется для Ваших программ, данных, переменных и стека (-ов), иначе называемых Вашим Propeller-Приложением.

При загрузке программы из хоста или внешней ЭСППЗУ, записывается полностью все это пространство памяти. Первые 16 адресов, $0000 – $000F, содержат инициализационные данные, используемые загрузчиком и интерпретатором. Исполнимый код и данные Вашей программы начинаются с адреса $0010 и продолжаются на некоторое количество двойных слов (*longs*). Пространство после Вашего исполнимого кода, простирающееся до $7FFF, используется для переменных и стека.

В инициализационной области хранятся два значения, которые могут понадобиться Вашей программе: *long* по адресу $0000 содержит начальную частоту ядра, в герцах, а следующий байт, по адресу $0004, содержит начальное значение, записываемое в регистр CLK. Эти два значения могут быть прочитаны либо записаны с использованием их физических адресов (`LONG[$0]` и `BYTE[$4]`), а также с использованием их предопределенных имен (**`CLKFREQ`** и **`CLKMODE`**). Если Вы измените значение регистра CLK без использования команды **`CLOCKSET`**, Вам будет необходимо так же обновить значения по этим адресам, чтобы объекты, которые их используют, имели правильную информацию.

Основное ПЗУ

Вторая половина Основной Памяти – это ПЗУ. Это пространство используется для хранения таблицы символов (знакогенератора), математических функций, загрузчика и интерпретатора языка *Spin*.

Знакогенератор

Первая половина ПЗУ предназначена для хранения набора из 256 определений символов. Каждое определение символа имеет 16 пикселей в ширину и 32 пикселя в высоту. Эти определения могут использоваться для видео дисплеев, графических ЖКИ, печати и т.д. Набор символов базируется на Северо-Американской/Западно-Европейской кодировках (базовая Latin и расширенная Latin-1), с большим количеством дополнительных специальных символов. Специальные символы предназначены для рисования форм сигналов и электрических схем, отображения

греческих символов, используемых в электронике, а так же нескольких стрелок и маркеров.

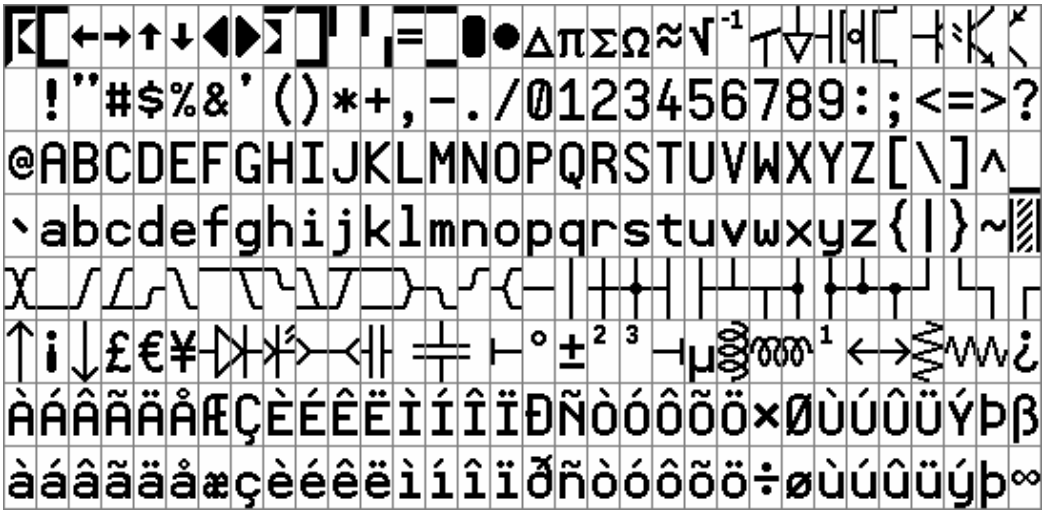


Рис. 1-6: Символы шрифта Propeller

Определения символов, пронумерованные от 0 до 255 слева-на-право сверху-вниз, приведены на Рис. 1-6. В ПЗУ они скомпонованы парами смежных четных-нечетных символов, слитых вместе для получения 32-х *long*-значений. Первая пара символов расположена по адресу $8000-$807F. Вторая пара занимает байты $8080-$80FF, и так далее, до последней пары по адресу $BF80-$BFFF. Программа *Propeller Tool* включает интерактивную таблицу символов, (Help → View Character Chart…) которая имеет просмотрщик ПЗУ, указывающий, где и как расположен в памяти каждый символ.

Пары символов слиты строка к строке таким образом, что каждые 16 горизонтальных пикселей одного символа перемежаются через один с пикселями своего соседа так, что пиксели четного символа занимают биты 0, 2, 4, ...30, а нечетного - биты 1, 3, 5, ...31. Пиксели слева занимают младшие биты, справа – старшие, как показано на Рис. 1-7. Такой формат выделяет 1 *long* (4 байта) на каждую строку пикселей для пары символов. В итоге 32 таких *long*-а, построенных от верха символов к низу, обеспечивают полное определение слитой пары символов. Определения закодированы в таком виде для того, чтобы аппаратная видео-подсистема процессоров могла оперировать слитыми *long*-ами напрямую, используя выбор цвета для отображения четного либо нечетного символа. Такой формат дает преимущество и для получения «интерактивных» пар символов (см. следующий параграф), которые являются

четырехцветными символами, используемыми для рисования фасонных кнопок, линий и индикаторов фокуса.

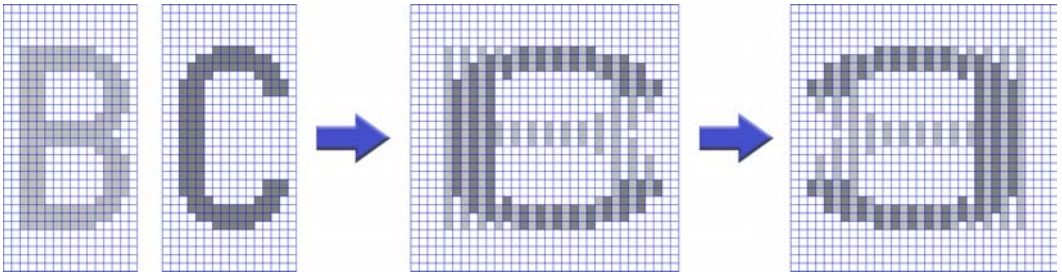


Рис. 1-7: Чередование символов в ИМС Propeller

Коды некоторых символов имеют общепринятые значения, такие как 9 для *Tab*, 10 для *Line Feed*, и 13 для *Carriage Return*. Эти коды символов вызывают действия, поэтому не подходят под определение статических символов. По этой причине их определения были использованы для создания специальных четырехцветных символов. Эти четырехцветные символы используются во время выполнения программы для рисования граней 3-D кнопок и имеют вид знакомест 16 x 16 пикселей, в отличие от нормальных знакомест 16 x 32 пикселей. Они занимают пары четных/нечетных символов 0-1, 8-9, 10-11, и 12-13. На Рис. 1-8 приведен пример кнопки с 3D-скошенными гранями, выполненной из некоторых таких символов.

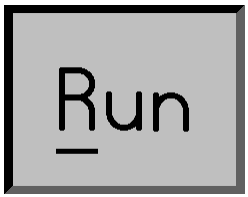


Рис. 1-8: Кнопка с 3D-скошенными гранями

Программа *Propeller Tool* включает в себя и использует шрифт *Parallax True Type*®, который повторяет дизайн аппаратного шрифта ИМС Propeller. Используя этот шрифт и *Propeller Tool*, вы можете включать в исходный текст своего приложения схемы, временные диаграммы и диаграммы любых других типов.

Таблицы Log и Anti-Log

Таблицы *log* и *anti-log* полезны для преобразования значений между их обычной и экспоненциальной формами представления.

Когда числа представлены в экспоненциальной форме, сложные математические операции могут быть реализованы более простыми действиями. Например, умножение и деление становятся сложением и вычитанием. Возведение в квадрат становится сдвигом влево, а квадратный корень – сдвигом вправо. Кубический корень даст деление на 3. Затем, для получения конечного результата, число необходимо преобразовать назад, в обычную форму представления.

См. Приложение B: Математические примеры и таблицы функций на стр. 540 для более детальной информации.

Таблица Sin

В таблице *sin* содержится 2049 беззнаковых 16-битных значений функции *sin*, изменяющихся от 0° до 90° включительно (разрешение 0.0439°). Значения *sin* для всех остальных квадрантов, занимающих углы от > 90° до < 360°, могут быть вычислены путем простейших преобразований по этой одно-квадрантной таблице. Таблица *sin* может быть использована при расчетах, связанных с угловыми величинами.

См. Приложение B: Математические примеры и таблицы функций на стр. 540 для более детальной информации.

Загрузчик и интерпретатор Spin

Последняя секция в основном ОЗУ содержит программы загрузчика и интерпретатора языка *Spin*.

Загрузчик отвечает за инициализацию ИМС Propeller при включении питания либо сбросе. Когда начата процедура загрузки, загрузчик копируется в ОЗУ *Cog* 0, и этот процессор выполняет код, начиная с адреса 0. Программа загрузчика сначала проверяет линии связи с хостом и ЭСППЗУ для загрузки кода/данных, затем соответствующим образом обрабатывает полученную информацию, и в конце – либо запускает программу интерпретатора в ОЗУ *Cog* 0 (перезаписывая ее поверх самого себя) для запуска приложения пользователя, либо переводит ИМС Propeller в режим останова. См секцию Начальная загрузка на стр.22.

Программа интерпретатора *Spin* извлекает и выполняет Propeller-приложение пользователя из основного ОЗУ. В зависимости от кода приложения, это может привести к запуску дополнительных *Cog*-ов для выполнения дополнительных частей *Spin* или ассемблерного кода. См секцию Начальная загрузка на стр.22.

Глава 2: Работа с программой Propeller Tool

Эта глава описывает особенности программы *Propeller Tool*, начиная с её концепции и структуры, предоставляя далее описание организации экранных форм, подробно останавливаясь на функциях меню и расширенных возможностях, и завершая описанием клавиш быстрого выбора команд (*shortcut keys*).

Общие положения

В течение более чем 20-ти летнего периода времени, инженерный состав компании Parallax использовал большое количество сред разработки. Часто, в процессе разработки приложений, мы ловили себя на мыслях:

- Было бы хорошо, если бы функцию “x” было легче обнаружить/запустить.
- Где файлы моего проекта и почему их так много?
- Смогу ли я легально инсталлировать/перекомпилировать/установить это на другой компьютер, а возможно, и через несколько лет?
- Существует ли менее дорогое решение?

Этот опыт привел нас к необходимости возобновить наше стремление к созданию для наших продуктов простых и недорогих инструментов.

Именно с такими мыслями и был разработан программный пакет *Propeller Tool*, призванный предоставить множество полезных функций, оставаясь простой и, вместе с тем, состоятельной средой разработки, обеспечивающей быстрый и легкий процесс создания программных объектов ИМС Propeller.

Программное обеспечение *Propeller Tool* состоит из единственного исполнимого файла, нескольких *on-line* файлов помощи и файлов *Propeller*-библиотек, сохраненных инсталлятором в одной папке; обычно это папка C:\Program Files\Parallax Inc\Propeller. Исполнимый файл пакета, “*Propeller.exe*”, может быть скопирован и запущен из любой папки на Вашем компьютере, он не зависит от каких-либо специфических файлов кроме тех, которые входят в стандартную установку операционной системы.

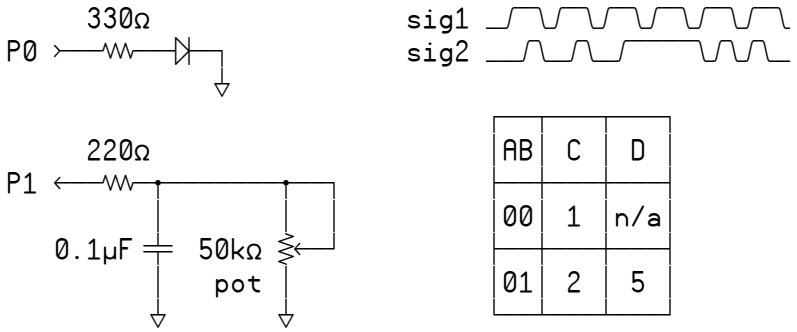
Каждый библиотечный файл (файлы с расширением “*.spin*”) является самодостаточным объектом, пригодным для использования в Ваших *Propeller*-проектах, включающий в себя как исходный текст, так и документацию. На самом деле

они являются просто текстовыми файлами, с ANSI- либо Unicode- кодировкой, которые могут редактироваться в любом текстовом редакторе, поддерживающем этот тип кодировки; даже Notepad в Windows® 2000 (и старше) поддерживает текстовые файлы с кодировкой ANSI и Unicode.

Вы заметили, что мы упомянули о документации, встроенной в файл объекта? Мы приветствуем написание документации пользователя на программный объект внутри самого исходного файла объекта. Это позволяет уменьшить количество файлов в проекте и увеличить вероятность того, что документация будет всегда синхронизирована с текущей версией исходного кода. Для дальнейшего осуществления этой идеи, мы создали:

- Два типа комментариев в исходном тексте, 1) комментарии кода (для описания частей исходного текста), и 2) комментарии документации (так же вводимые в коде, но предназначенные для чтения при помощи функции “просмотр документации” (*documentation view*)).
- Режим “просмотр документации» в программе *Propeller Tool*, который выбирает для просмотра документацию на программный объект из его исходного текста.
- Специальный шрифт – «*Parallax*», который содержит специальные символы для отображения фрагментов схем, временных диаграмм и таблиц, в рамках документации на объект.

Шрифт *Parallax* – это шрифт *True Type*®, встроенный в исполнимый файл *Propeller Tool*. Он разработан в том же стиле, что и шрифт, встроенный в ПЗУ ИМС Propeller. Используя специальные символы шрифта, в документацию на объект можно включить полезные диаграммы для инженерных целей, к примеру, такие:



AB	C	D
00	1	n/a
01	2	5

Рис. 2-1: Графика, построенная при помощи шрифта Parallax

После того, как *Propeller Tool* был запущен хотя бы один раз, этот шрифт становится доступным и для других приложений на данном компьютере. Таким образом, Вы можете видеть специальные диаграммы и при использовании других текстовых редакторов, таких как Notepad, или даже в вашем почтовом клиенте, если он поддерживает Unicode-кодированный текст (требование для отображения специальных символов).

Каждый объект, создаваемый Вами в проекте, будет сохранен в том же формате, как и файл библиотеки (с расширением ".spin"), в рабочей папке по Вашему выбору. Все это задумано, чтобы способствовать использованию существующих объектов, разработанных нами либо пользователями продуктов Propeller.

Более подробную информацию о файлах, объектах, их документации, библиотечных файлах и исходном коде предоставляет Глава 3: Программирование ИМС Propeller .

Организация Экрана

Главное окно программы *Propeller Tool* разбито на четыре секции, называемые панелями ("panes"). Каждая панель имеет свои специфические функции.

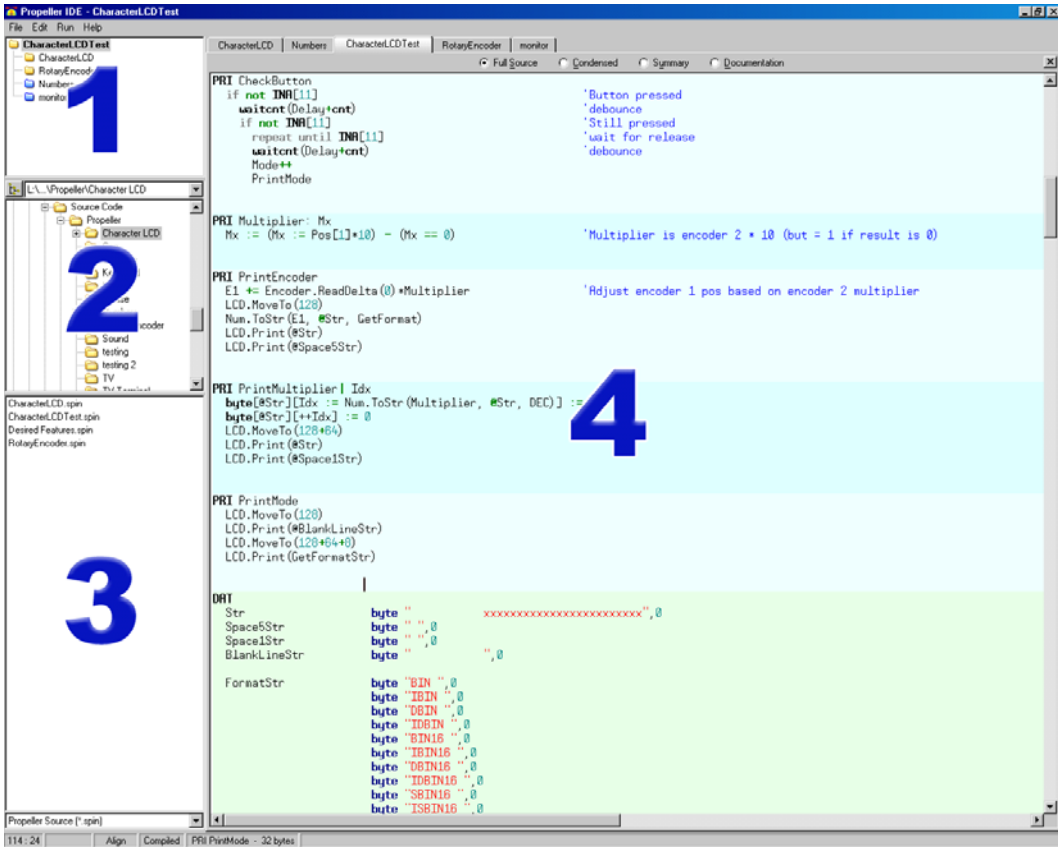


Рис. 2-2: Главное окно программы Propeller Tool содержит четыре панели.

Панели один, два и три являются частью Интегрированного Браузера. Интегрированный браузер – это часть окна, слева от Панели Редактора (панель четыре); он обеспечивает просмотр проекта, над которым Вы работаете, а так же папок и файлов на диске. Интегрированный браузер отделен от панели Редактора при помощи высокой вертикальной разделительной полосы, его размеры можно изменять в любой момент при помощи мыши. Он даже может быть спрятан, путем уменьшения его размеров до нуля (нажать левую кнопку и потянуть его вертикальный разделитель), выбором File → Hide Explorer, или нажатием *Ctrl+E*. Опции меню и быстрые клавиши переключают браузер между: 1) Видимым (с заданным ранее размером), и 2) Невидимым (полностью свернутым к левой кромке *Propeller Tool*.

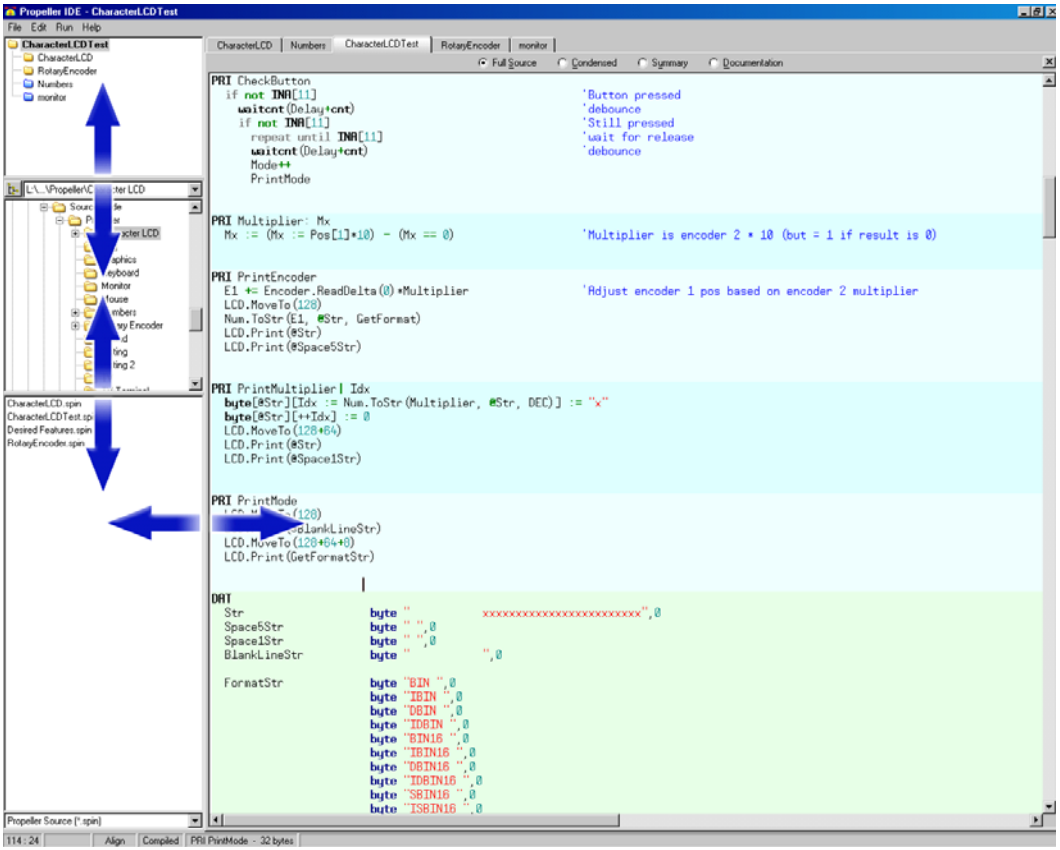


Рис. 2-3: Встроенный браузер и его компоненты могут изменять свои размеры при помощи разделительных полос.

Панель 1: Вид объекта

Панель один – это панель «Вид объекта». Язык ИМС Propeller – *Spin*, является объектно-ориентированным и *Propeller*-проект может состоять из многих объектов. «Вид объекта» показывает иерархию последнего успешно откомпилированного проекта, обеспечивая полезную обратную связь, отображающую структуру взаимосвязей объектов в Вашем проекте. Используя «Вид объекта», вы можете определить, какие объекты используются, как они согласовываются друг с другом, их расположение на диске (рабочая папка, папка библиотек, либо только редактор),

результаты оптимизации (если есть) и любые потенциальные коллизии между объектами. См «Вид Объекта (Object View)» на стр.64 для подробной информации.

Панель 2: Поле последних открытых папок и перечень директорий

Панель 2 содержит два компонента: 1) поле последних открытых папок, и 2) перечень директорий. Эти два компонента работают совместно, обеспечивая доступ для навигации по доступным на Вашем компьютере дисковым накопителям. Перечень директорий отображает иерархию папок в рамках каждого диска, и им можно управлять таким же образом, как и левой панелью Windows® Explorer.

Поле последних открытых папок (над перечнем папок) отображает выпадающий список специальных папок и папок, из которых Вы загружали файлы в последний раз. Выбор папки из этого списка приводит к тому, что перечень директорий мгновенно отображает содержимое указанной папки. К тому же, если Вы выберете в перечне папку, которая уже есть в списке последних открытых папок, поле списка автоматически обновится и покажет эту папку.

Первыми в списке последних открытых файлов находятся “Propeller Library” и “Propeller Library – Demos.” Эти папки добавляются автоматически, чтобы всегда указывать путь, где находятся библиотеки и демонстрационные программы к ним. Эти файлы устанавливаются при инсталляции *Propeller Tool*.

Если же Вы выбираете папку, которой нет в списке последних открытых, поле последней открытой папки будет пустым. Кнопка слева от поля последних открытых папок переключает функции обоих полей между: 1) отображением каждого диска и папки, и 2) отображением только недавно использовавшихся дисков и директорий. Установка режима отображения только недавно открывавшихся папок позволяет удобно и быстро добираться до наиболее часто используемых папок среди большого их количества, не относящихся к Propeller.

Панель 3: Перечень файлов и поле фильтра

Панель три включает два компонента: 1) перечень файлов, и 2) поле фильтра. Перечень файлов отображает все файлы, содержащиеся в выбранной в перечне директорий папке, которые удовлетворяют критерию фильтра, заданному в поле фильтра. Перечень файлов может использоваться в том же стиле, что и правая панель в Windows Explorer.

Поле фильтра, находящееся под перечнем файлов, отображает выпадающий список расширений файлов, называемых фильтрами, которые нужно показывать в перечне файлов. Обычно оно установлено для отображения только файлов *Spin* (которые имеют расширение “.spin”). но может так же быть установлено для отображения только

текстовых, либо всех файлов. Если Вы открыли папку, но не видите файлов, которые ожидали, убедитесь, что текущее значение фильтра правильное.

Файлы из перечня могут быть открыты в редакторе путем: 1) двойного щелчка на нужном файле, 2) отметив и перетянув файл на панель редактора (панель четыре), или 3) правым щелчком мыши и выбором из выпадающего меню команды Open.

Панель 4: Панель редактора

Панель четыре – это панель редактора. Панель редактора обеспечивает возможность просмотра открытых Вами файлов исходного кода на *Spin*, и является местом, где Вы можете просмотреть, отредактировать либо выполнить другие действия над файлами Вашего проекта. Каждый файл (объект исходного кода), открытый Вами, организован внутри панели редактора как индивидуальная закладка, названная именем файла, который она содержит. Активная (редактируемая в данный момент) закладка подсвечивается, в отличие от остальных. Вы можете открывать столько файлов, сколько необходимо, их количество ограничено лишь объемом доступной памяти.

Клик на необходимую закладку позволяет увидеть содержание редактируемой страницы. Вы можете переключаться между открытыми файлами путем: 1) нажатия *Alt+CrsrLeft* или *Alt+CrsrRight*, либо 2) нажатия *Ctrl+Tab* или *Ctrl+Shift+Tab*. Если Вы удержите курсор мыши над редактируемой закладкой достаточно долго, Вам будет показана подсказка с полным путем и именем файла, который в ней находится. Исходный текст на редактируемой странице имеет автоматическую синтаксис-подсветку, цветами как переднего, так и заднего планов, которая помогает выделить блоки, элементы, комментарии в отличие от исполнимого кода и т.д.

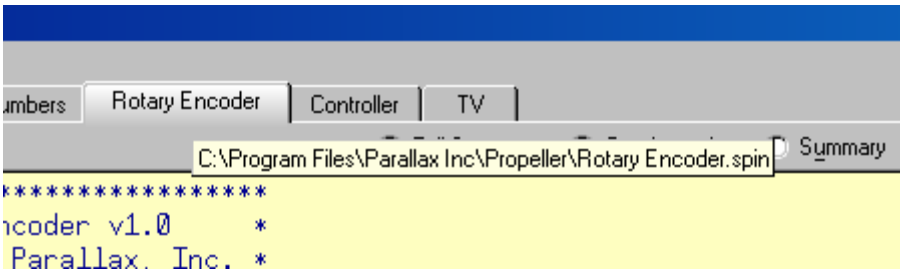


Рис. 2-4: Удержите мышь над редактируемой закладкой, чтобы увидеть полный путь и имя файла, который закладка содержит.

Каждая страница редактирования может отображать исходный текст в одном из четырех режимов:

1) Полный (Весь исходный текст)
2) Сжатый
3) Общий
4) Документация.

Режим просмотра можно увидеть либо изменить индивидуально для каждой закладки, путем:

1) выбора соответствующей радио-кнопки с помощью мыши,

2) нажатия *Alt+Up* или *Alt+Down*,

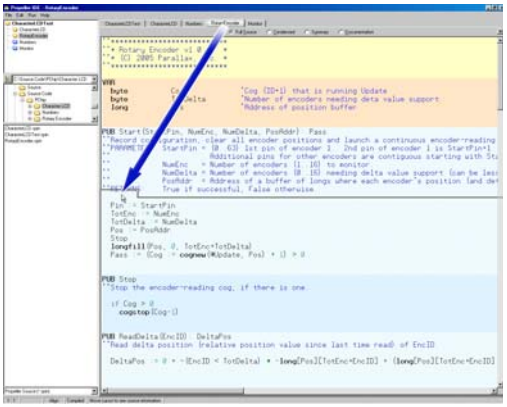
3) нажатия *Alt+<символ>*, где *<символ>* — это подчеркнутая горячая клавиша необходимого вида,

4) нажатия *Alt* и перемещения колеса мыши вверх либо вниз.

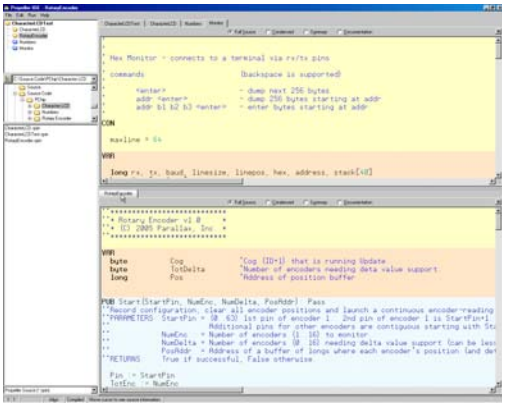
Учтите, что режим просмотра «Документация» не доступен, если в данный момент объект не может быть полностью откомпилирован. См. секцию «Режимы просмотра, Отметки и Номера » со стр. 74, для более подробной информации об этих режимах.

Поскольку проект может состоять из многих объектов, разработка может стать неудобной, если Вы не видите одновременно и объект, над которым работаете, и объект, к которому Вы подключаетесь. В этом случае помогает «Панель редактора», позволяя перемещать свои редактируемые закладки в различные места. Например, когда открыто несколько объектов, Вы можете с помощью левой кнопки мыши выделить и потянуть редактируемую вкладку объекта вниз, к нижней половине панели редактора, и просто оставить ее там. Экран изменит свой вид и покажет вам новую открытую вкладку в том месте, куда вы кинули выбранную вкладку. Вы можете и дальше продолжать выбирать, перетягивать и бросать редактируемые вкладки в это место, если это необходимо. Эти шаги продемонстрированы на Рис. 2-5.

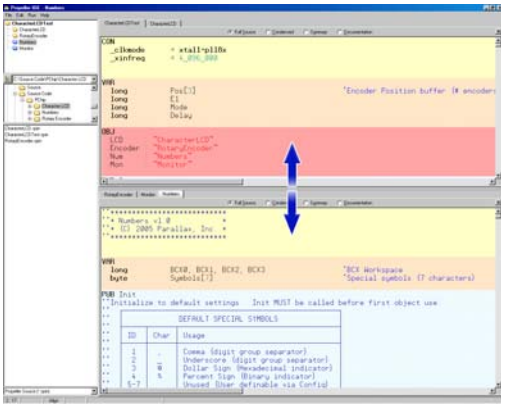
Рис. 2-5: Просмотр и расположение нескольких объектов



Шаг 1: Чтобы увидеть код нескольких объектов одновременно, выполните нажатие левой кнопки мыши и потяните вкладку в нижнюю часть панели редактора.



Шаг 2: Отпустите кнопку, чтобы бросить редактируемую вкладку. Редактируемая вкладка и ее содержимое теперь находится в новом месте.

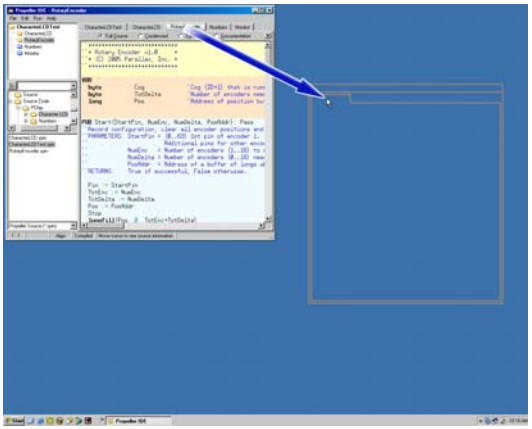


Шаг 3: Повторяйте шаги 1 и 2 при необходимости для других вкладок и измените размеры обоих областей, используя горизонтальный разделитель.

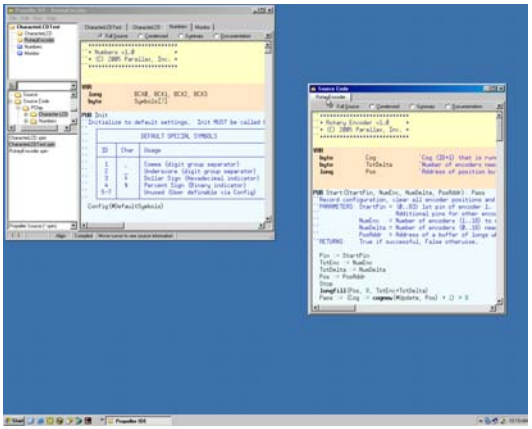
Вертикальный размер этих двух областей может быть изменен перемещением горизонтального разделителя, находящегося между ними. Объекты, к которым Вы подключаетесь, могут быть просмотрены в любом из режимов, подходящем в данный момент («Полный», «Сжатый», «Общий» и «Документация»), в то время как Ваш разрабатываемый объект остается в режиме просмотра всего исходного текста «Полный» (единственный редактируемый вид).

Панель редактора позволяет перетягивать вкладки даже за пределы *Propeller Tool*. При этом новые редактируемые вкладки займут новое окно, которое может быть изменено не зависимо от окна программы *Propeller Tool*. Это особенно полезно, когда у компьютера разработчика более одного монитора; редактируемые вкладки могут быть перетянуты из приложения на одном мониторе и кинуты на стол второго монитора.

Рис. 2-6: Расположение объектов.



Шаг 1: Если позволяет размер рабочего стола, Вы можете перетягивать редактируемые вкладки даже за пределы самого приложения; нажмите и удерживайте левую кнопку мыши и тяните вкладку в область вне Propeller Tool.



Шаг 2: Отпустите кнопку, чтобы бросить вкладку; она примет свою собственную форму, которая способна перемещаться и изменять размеры независимо от Propeller Tool. Вы можете перетягивать и бросать несколько вкладок.

Строка Статуса в нижней части приложения *Propeller Tool* разделена на шесть панелей, каждая из которых показывает полезную информацию на различных стадиях процесса разработки.

Панель один строки статуса всегда показывает номер строки и столбца в позиции курсора в текущей редактируемой вкладке.

Рис. 2-7: Строка статуса

Панель два отображает измененный статус текущей редактируемой вкладки: 1) пустое, что значит без изменений, 2) изменен, или 3) только для чтения.

Панель три отображает текущий режим редактирования: 1) Вставка (по умолчанию), 2) Выравнивание, или 3) Замена. Режим редактирования может быть изменен нажатием кнопки Insert. См. секцию «Режимы редактирования» со стр. 79 для более детальной информации о различных режимах работы редактора.

Панель четыре показывает статус компиляции текущей редактируемой вкладки: 1) пустое, что значит – не компилировался, или 2) compiled – откомпилирован. Эта панель показывает, находится ли файл текущей редактируемой вкладки в том же виде, в каком был при последней компиляции. Если код никак не изменился с момента компиляции, панель будет отображать “Compiled.”

Панель пять показывает контекстную информацию об исходном тексте текущей редактируемой вкладки, был ли он изменен со времени последней компиляции. Переместите курсор на редактируемой странице к блочным символам `CON` или `DAT` или куда-либо в пределах блоков `PUB`/`PRI`, чтобы увидеть информацию, принадлежащую этой области.

Шестая панель отображает временные сообщения о самой последней операции. Это та область строки статуса, в которой отображается сообщение об ошибке (при наличии таковой) с момента последней компиляции, до тех пор, пока следующее сообщение его не перекроет. В этом месте так же сообщается об успешной компиляции, изменениях размера шрифта и других событиях.

Вся строка статуса показывает подсказки, описывающие функции каждого пункта в строке меню, а так же многих других элементов, когда Вы задерживаете курсор мыши над ними.

Пункты меню

Меню Файл (File)

Новый (New) Создать новую вкладку с пустой страницей. Все существующие вкладки без изменений.

Открыть (Open…) Открыть файл в новой вкладке при помощи диалога открытия файла.

Открыть из (Open From…) Открыть файл в новой вкладке из последней открытой папки с использованием диалога открытия файла.

Сохранить (Save) Сохранить текущее содержимое вкладки на диск, используя текущее имя файла, если возможно.

Сохранить как (Save As…) Сохранить текущее содержимое вкладки на диск с новым именем файла, используя диалог сохранения файла.

Сохранить в (Save To…) Сохранить текущее содержимое вкладки на диск в последнюю открытую папку, используя диалог сохранения файла.

Сохранить все (Save All) Сохранить все несохраненные вкладки на диск, используя их имена файлов, если возможно.

Закрыть (Close) Закрыть текущую вкладку (с напоминанием, если не сохранено).

Закрыть все (Close All) Закрыть все вкладки (с напоминанием, если какой-либо не сохранен).

Выбрать главный файл

(Select Top Object File…) Выбрать главный файл текущего проекта. Эта установка используется для всех операций компиляции и остается неизменной до изменения пользователем.

Архивировать (Archive)

→ Проект (Project…) Собрать все файлы объектов и данных для проекта, обозначенного в панели «Вид Объекта» и сохранить их в сжатом (.zip) файле вместе с файлом “read me”,

хранящем информацию о структуре архива. Архивный файл называется именем главного файла с добавкой “Archive” и даты/времени и сохраняется в папке главного файла.

→ Проект+ *Propeller Tool*...

(Project + *Propeller Tool*...) Выполняет ту же задачу, что и предыдущий пункт, но плюс добавляет весь исполнимый файл *Propeller Tool* к архивному файлу.

Показать/Скрыть Браузер

(Hide/Show Explorer) Скрыть либо показать панели встроенного браузера (левая сторона окна приложения).

Просмотр печати

(Print Preview…) Просмотреть образец выходного файла перед печатью.

Печать (Print…) Вывести на печать содержимое редактируемой вкладки.

<недавние файлы>

<recent files> Область меню между пунктами «Печать…» и «Выход» показывает до десяти самых последних открытых файлов. Выбор одного из этих пунктов открывает соответствующий файл. Укажите мышью на любой из последних открытых файлов в меню, чтобы увидеть полный путь и имя файла в строке статуса.

Выход (Exit) Выйти из программы *Propeller Tool*.

Меню Редактировать (Edit)

Откат (Undo) Откатить последнюю выполненную операцию на текущей редактируемой странице. Для каждой редактируемой страницы существует своя история оката, пока приложение не закрыто. Доступен глубокий дамп отката, ограниченный лишь объемом памяти ЭВМ.

Повтор (Redo) Повторить последнее отмененное действие на текущей редактируемой странице. Для каждой редактируемой страницы существует своя история повтора, пока

	приложение не закрыто. Доступен глубокий дамп повтора, ограниченный лишь объемом памяти ЭВМ.
Вырезать (Cut)	Удалить отмеченный текст с текущей страницы и копировать его в буфер (Windows clipboard).
Копировать (Copy)	Копировать выделенный текст с текущей страницы в буфер обмена (Windows clipboard).
Вставить (Paste)	Вставить текст из буфера обмена на текущую страницу в текущую позицию курсора.
Выделить все (Select All)	Выделить весь текст на текущей странице.
Найти/Заменить (Find / Replace…)	Открыть диалог «Найти/Заменить»; см «Диалог Найти/Заменить», на стр. 61, для детальной информации.
Найти следующий (Find Next)	Найти следующее совпадение последней строки, введенной в диалоге «Найти/Заменить».
Заменить (Replace)	Заменить текущее выделение на строку, введенную в поле «Заменить» диалога «Найти/Заменить».
Перейти на отметку (Go To Bookmark)	Перейти на отметку 1, 2, 3… (активно только когда показаны отметки).
Увеличить текст (Text Bigger)	Увеличить размер шрифта на каждой редактируемой странице.
Уменьшить текст (Text Smaller)	Уменьшить размер шрифта на каждой редактируемой странице.
Настройки Preferences…	Открыть окно «Preferences». Пользователи могут настроить множество параметров в рамках *Propeller Tool* используя эту функцию.

Меню Выполнить (Run)

Компилировать текущий

(Compile Current)

→ Смотреть информацию

(View Info…) Компилировать исходный код в текущей вкладке и, если удачно, показать форму «Информация об Объекте» с результатами. Эта форма отображает много деталей о конечном объекте включая его структуру, размер кода, пространство переменных, свободное пространство и отчет об оптимизации.

→ Обновить статус

(Update Status) Компилировать исходный код в текущей вкладке и, если успешно, обновить информацию в строке статуса для каждого объекта в проекте.

→ Загрузить ОЗУ

(Load RAM) Компилировать исходный код в текущей вкладке и, если успешно, загрузить полученное приложение в ОЗУ ИМС Propeller и запустить его.

→ Загрузить ЭСППЗУ

(Load EEPROM) Компилировать исходный код в текущей вкладке и, если успешно, загрузить полученное приложение в ЭСППЗУ и запустить его.

Компилировать верхний

(Compile Top)

→ Смотреть информацию

(View Info…) То же, что и «Compile Current → View Info», за исключением того, что компиляция стартует с файла, указанного как «Top Object File».

→ Обновить статус

(Update Status) То же, что и «Compile Current → Update Status», за исключением того, что компиляция стартует с файла, указанного как «Top Object File».

→ Загрузить ОЗУ То же, что и «Compile Current → Load RAM + Run», за исключением того, что компиляция стартует с файла, указанного как «Top Object File».

→ Загрузить ЭСППЗУ То же, что и «Compile Current → Load ЭСППЗУ + Run», за исключением того, что компиляция стартует с файла, указанного как «Top Object File».

Опрелить Аппаратуру

(Identify Hardware…) Сканировать доступные порты для поиска ИМС Propeller и, если таковая будет найдена, показать порт, к которому подключена, а так же номер версии.

Меню Помощь (Help)

Помощь *Propeller Tool*… Показать on-line помощь о *Propeller Tool*.

Язык *Spin*… Показать on-line помощь по языку *Spin*.

Язык ассемблера … Показать on-line помощь об ассемблере Propeller.

Примеры Проектов… Показать on-line помощь включающую примеры проектов Propeller.

Просмотр Таблицы Символов

(View Character Chart…) Показать интерактивный набор символов Parallax. Этот набор символов отображает символы шрифта Parallax в трех видах: «Стандартный порядок», «Карта ПЗУ» и «Символьный порядок». «Стандартный порядок» – это порядок ANSI. «Карта ПЗУ» показывает, как данные символов организованы в ПЗУ ИМС Propeller. «Символьный Порядок» перечисляет символы в порядке по категориям (т.е. буквенные символы, цифры, пунктуация, схематические символы и т.д.). См. Таблица символов на стр. 70.

Смотреть вебсайт Parallax	Открыть вебсайт Parallax в текущем вэб-браузере.
Поддержка по E-mail	Открыть почтовый клиент по умолчанию для написания письма в Parallax.
Про (About…)	Показать окно с подробностями о программе *Propeller Tool*.

Диалог Найти/Заменить

Диалог «Найти/Заменить» используется для нахождения и/или замены текста на текущей редактируемой странице.

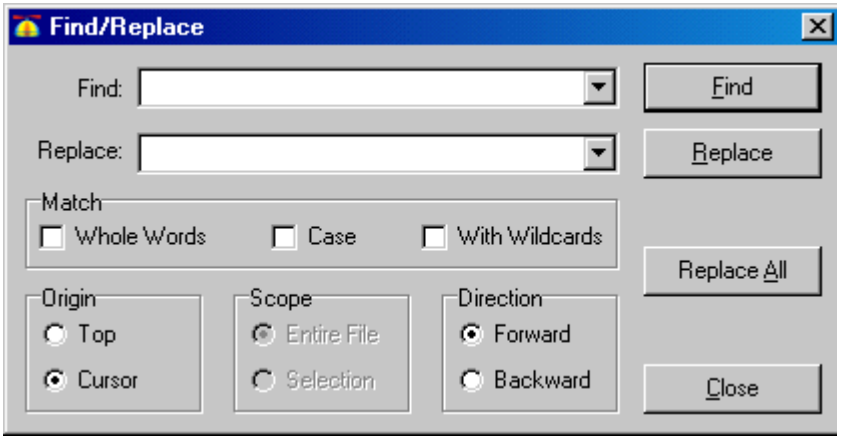


Рис. 2-8: Диалог Найти/Заменить

Найти: (Find:)

Поле «Найти:» предназначено для ввода строки для поиска. Если слово или фраза были выделены на текущей странице, когда был открыт диалог «Найти/Заменить», это слово или фраза автоматически будет занесена в поле для поиска. Это поле помнит последние десять различных фраз, введенных в него. Для выбора предыдущего слова или фразы поиска, нажмите стрелку в правой части строки поиска и выберите необходимый пункт в выпадающем списке.

Заменить: (Replace:)

Поле «Заменить:» предназначено для ввода строки, на которую будет заменена найденная. Это поле помнит последние десять различных фраз, введенных в него. Для выбора предыдущего слова или фразы для замены, нажмите стрелку в правой части строки замены и выберите необходимый пункт в выпадающем списке.

Совпадение (Match)

Группа «Совпадение» контролирует, как строка в поле «Найти:» будет соотноситься с текстом на редактируемой странице. Возможные условия: 1) Целые слова, 2)Регистр и 3) С групповыми символами (With Wildcards).

Целые слова (Whole Words)

Выберите «Целые слова», если Вы хотите, чтобы в найденном тексте слова полностью совпадали с введенными в строку поиска, а слова, которые кроме введенной последовательности символов содержат еще символы, находиться не будут.

Регистр (Case)

Отметьте «Регистр», если Вы хотите, чтобы в найденном тексте регистр всех символов был точно таким, как в веденной в поле поиска строке.

С групповыми символами (With Wildcards)

Выберите «With Wildcards» если Вы хотите, чтобы поиск выполнялся с использованием выражений с групповыми символами (wildcards) из строки поиска.

Группы «Начало» (Origin), «Диапазон» (Scope) и «Направление» (Direction) работают совместно, чтобы определять начало, размер и направление, в котором проводится поиск.

Начало (Origin)

Группа «Начало» определяет, откуда начинается поиск: с начала документа либо от позиции курсора. Выбор опции «Начало» (Top) определяет начало поиска с вершины файла (либо с вершины выбранного участка, если задана опция «Выделенное» (Selection) в группе Scope). Выбор опции «Курсор» (Cursor) определяет начало поиска от текущей позиции курсора в файле. Заметьте, что опция “Top” меняется на “Bottom”, если в группе «Направление» (Direction) выбрать «Назад» (Backward).

Диапазон (Scope)

Группа «Диапазон» определяет размер области поиска: весь файл либо текущая отмеченная область (Selection). Это удобный способ выполнить поиск либо поиск и замену в ограниченной области файла. По умолчанию, группа «Диапазон» установлена в значение «Весь Файл» (Entire File) и отключена, если только не сделано выделение области файла до открытия диалога «Найти/Заменить». Группа «Диапазон» устанавливается автоматически в значение «Выделенное» (Selection), если выделена хотя бы одна целая строка до открытия диалога «Найти/Заменить».

Направление (Direction)

Группа «Направление» управляет направлением поиска: «Вперед» (Forward) - в сторону конца файла либо «Назад» (Backward) в направлении начала файла. Если установить «Назад», первая опция группы «Область» (Origin) изменится с “Top” на “Bottom”, означая, что начало находится внизу файла или выделения.

Кнопка Найти (Find Button)

Кнопка «Найти» начинает процесс поиска, основанный на всех установках в диалоге «Найти/Заменить». Если текст на редактируемой странице удовлетворяет критериям поиска, он выделяется и выводится перед глазами, и затем кнопка «Найти» изменяется на кнопку «Найти Далее». Дальнейшие клики по кнопке «Найти Далее» приводят к выделению следующего совпадения и его показ. Для выполнения последующих поисков, Вы также можете использовать клавишу *F3*, с открытием диалога либо без него.

Кнопка Заменить (Replace Button)

Кнопка «Заменить» включается, когда что-либо было введено в строку ввода «Заменить» и совпавшая строка была найдена (по кнопке «Найти» либо клавише F3). Клик на «Заменить», или нажатие *F4* (с открытием диалога либо без него), ведет к замене совпавшей строки текста на введенную в поле «Заменить». После замены, необходимо воспользоваться кнопкой «Найти Далее», либо клавишей *F3* перед тем, как «Замена» будет доступна вновь. Удержание клавиши *Ctrl* меняет кнопку «Заменить» на «Заменить/Найти» и ее нажатие либо нажатие клавиш *Ctrl+F4* (с открытием диалога или без него), ведет к замене текущей совпавшей строки и немедленному выполнению следующей операции «Найти Далее».

Кнопка Заменить Все (Replace All Button)

Кнопка Заменить Все включается, когда в поле Заменить вводится строка. Нажатие на эту кнопку ведет к тому, что все, совпавшие с заданной, строки будут найдены и заменены на строку из поля Заменить, дальнейшему закрытию диалога и появлению окна с индикацией количества найденных и замененных строк.

Кнопка Закрыть (Close Button)

Кнопка Закрыть закрывает диалог Найти/Заменить.

Вид Объекта (Object View)

«Вид Объекта» отображает иерархический вид последнего успешно откомпилированного проекта. В программе *Propeller Tool* существует два варианта отображения вида объекта: 1) «Вид Объекта», в верхней части встроенного браузера в окне главного приложения (см. «Панель 1: », на стр. 47) и 2) «Просмотр информации об объекте», вверху слева в форме «Информация об Объекте» (см. Информация об объекте (Object Info) , на стр. 67). Оба эти вида функционируют одинаково.

«Вид Объекта» обеспечивает визуальную обратную связь со структурой объектов в последнем успешно откомпилированном проекте, а так же информацию на каждый объект в нем.

Рис. 2-9: Пример Вида Объекта, отображающего структуру приложения «ABC Product» после компиляции

На Рис. 2-9 вверху, «Вид Объекта» отображает структуру приложения «ABC Product» после последней успешной компиляции. В этом примере «ABC Product» является “верхним объектным файлом” (см. «Объекты и приложения», стр.103) и использует объекты «Numbers», «Rotary Encoder» и «Controller». Кроме того, объект «Controller» использует для своих целей объект «TV».

Приведенные имена объектов на самом деле являются именами файлов без расширений. Имя включает расширение файла только в том случае, если это - файл данных (см. `FILE`, стр. 252) кроме того оно отображается шрифтом *italics*.

Иконки слева от имени каждого объекта обозначают папку, в которой находится объект. Далее в списке отражены четыре возможных варианта:

- (желтый): Объект находится в Рабочей Папке .
- (голубой) : Объект находится в Библиотечной Папке.
- (полосы): Объект – в Рабочей папке, но другой объект с таким же именем используется также из Библиотечной Папки.

- (пустой): Объекта нет ни в одной из папок, т.к. он не был сохранен.

Рабочая папка (Work Folder)

«Рабочая папка» (желтый) – это директория, в которой находится верхний объектный файл. В каждом проекте есть одна, и только одна, рабочая папка.

Библиотечная папка (Library Folder)

«Библиотечная папка» (голубой) – это директория, где находятся объекты библиотеки *Propeller Tool*, такие, как в пакете *Propeller Tool*. «Библиотечная Папка» – это то место, откуда всегда запускается исполнимый файл *Propeller Tool*, и каждый объект (файл с расширением.spin) внутри нее рассматривается как библиотечный.

Полосатые Папки (Striped Folders)

Объекты с полосатыми иконками обозначают, что объект из рабочей папки и объект из библиотечной папки – каждый ссылаются к суб-объекту с одинаковым именем, и что этот суб-объект может существовать и в рабочей, и в библиотечной папках. Этим объектом с идентичным именем может быть: 1) точная копия этого же объекта, 2) две версии одного объекта, либо 3) два совершенно различных объекта, которые волей случая получили одинаковые имена. Независимо от реальной ситуации, рекомендуется решать такие потенциальные проблемы как можно быстрее, так как они могут в последствии привести к проблемам; например, из-за невозможности использования функции Архива.

Пустые Папки (Hollow Folders)

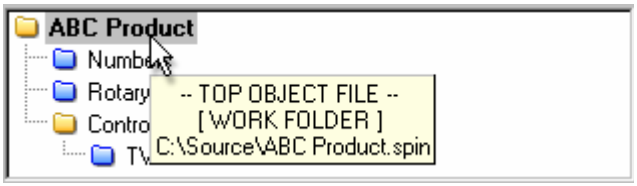
Объекты с пустыми иконками обозначают, что объект уже был создан в редакторе, но еще не был сохранен в какую-либо папку на диске. Такая ситуация, подобно описанной выше, не является большой проблемой, но может привести к серьезным последствиям, если не будет исправлена в ближайшее время.

Дополнительную информацию об объектах можно получить при использовании мыши для указания и выделения. Клик на объекте в панели «Вид Объекта» открывает этот объект в панели редактора. Левый клик открывает объект в режиме «Всего Исходного текста», правый – в режиме «Документация», а двойной клик открывает и его, и все его суб-объекты, в режиме просмотра «Весь Исходный текст». Если объект уже был открыт, панель редактора просто делает соответствующую вкладку активной и переключается в необходимый режим просмотра: «Весь Исходный текст» при левом или двойном клике, и «Просмотр Документации» – при правом клике.

Удержание курсора мыши над объектом в панели «Вид Объекта» приводит к появлению подсказки с дополнительной информацией об этом объекте. Рис. 2-10a показывает подсказку для объекта «ABC Product». Эта подсказка отображает: 1) объект «ABC Product» является верхним файлом проекта, 2) он находится в рабочей папке, и 3) его путь и имя файла такие: C:\Source\ABC Product.spin. Из этой информации можно так же узнать, что рабочая папка для проекта:

C:\Source

a.



b.

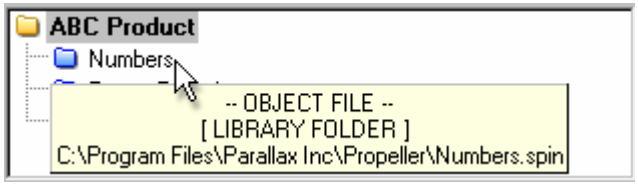


Рис. 2-10: Удерживайте курсор мыши над объектом, чтобы увидеть подсказку с дополнительной информацией.

Рис. 2-10b показывает подсказку для объекта «Numbers»: 1) он является файлом объекта (т.е. суб-объекта, а не верхнего объекта), 2) он в библиотечной папке, и 3) путь к нему и его имя файла: C:\Program Files\Parallax Inc\Propeller\Numbers.spin. Из этой информации можно так же узнать, что библиотечная папка для этого проекта это:

C:\Program Files\Parallax Inc\Propeller.

Периодический просмотр подсказок в панели «Вид Объекта» является хорошей практикой, так как из них можно получить дополнительную полезную информацию, такую как предупреждения о конфликтах и результы оптимизации.

Информация об объекте (Object Info)

Форма «Информации об объекте» отображает детали о только что успешно откомпилированном (с помощью функции «Compile Current/Top → View Info…») объекте. В верхней части этой формы, так же, как и во встроенном браузере, находится «Вид Объекта» (см. «Вид Объекта (Object View)», стр. 64). Под «Видом» и «Информацией объекта» находятся две панели с общей информацией.

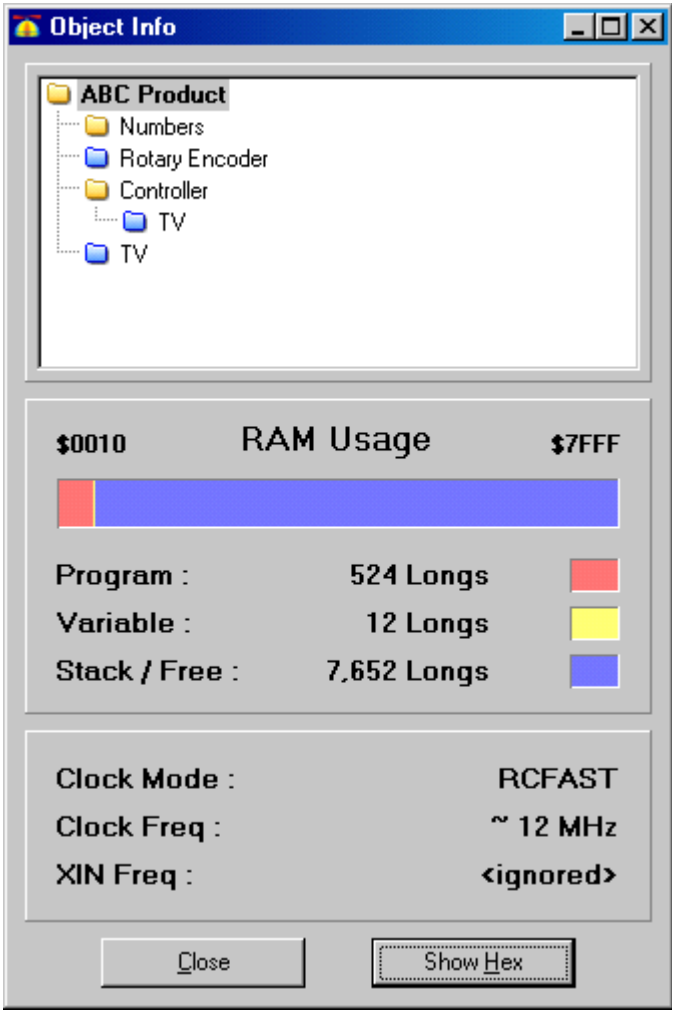


Рис. 2-11: Форма Информации об объекте

В этом примере показано окно информации об объекте, отображающее детали компиляции проекта "ABC Product".

Просмотр Информации об объекте (Info Object View)

«Просмотр информации» об объекте работает абсолютно аналогично панели «Вид Объекта» (см. Вид Объекта (Object View), стр. 64), с некоторыми исключениями:

- Клик на объекте в просмотре информации на объект обновляет на экране информацию, принадлежащую этому объекту.
- Двойной клик на объекте в просмотре информации на объект открывает этот объект в панели редактора.
- Файлы данных невозможно отметить при просмотре информации на объект.

Панель использования ОЗУ (RAM Usage Panel)

«Панель использования ОЗУ» отражает статистику объема ОЗУ, занятого текущим, выбранным в просмотре информации, объектом. Горизонтальная строка-индикатор отображает общий вид ОЗУ с цветовым и цифровым представлением деталей. Например, на Рис. 2-11 показано, что объект «ABC Product» занимает 524 *long*-а (2096 байт) под программу и 12 *long*-ов (48 байт) под переменные, оставляя более 7k *long*-ов (более 30k байт) свободными.

Панель Генератора (Clock Panel)

«Панель Генератора», расположенная под «Панелью Использования ОЗУ», отражает установки частоты генератора для текущего, выбранного в панели Информации, объекта. Например, Рис. 2-11 показывает, что у объекта «ABC Product» генератор конфигурирован как `RCFAST`, приблизительно 12 МГц и без частоты на XIN.

Вид Hex

Кнопка «Показать/Скрыть Hex» показывает либо скрывает детализированный шестнадцатеричный вид объекта, как на Рис. 2-12 на следующей странице. Hex-вид показывает в шестнадцатеричной форме реальные откомпилированные данные объекта, которые заносятся в ОЗУ/ЭСППЗУ при загрузке.

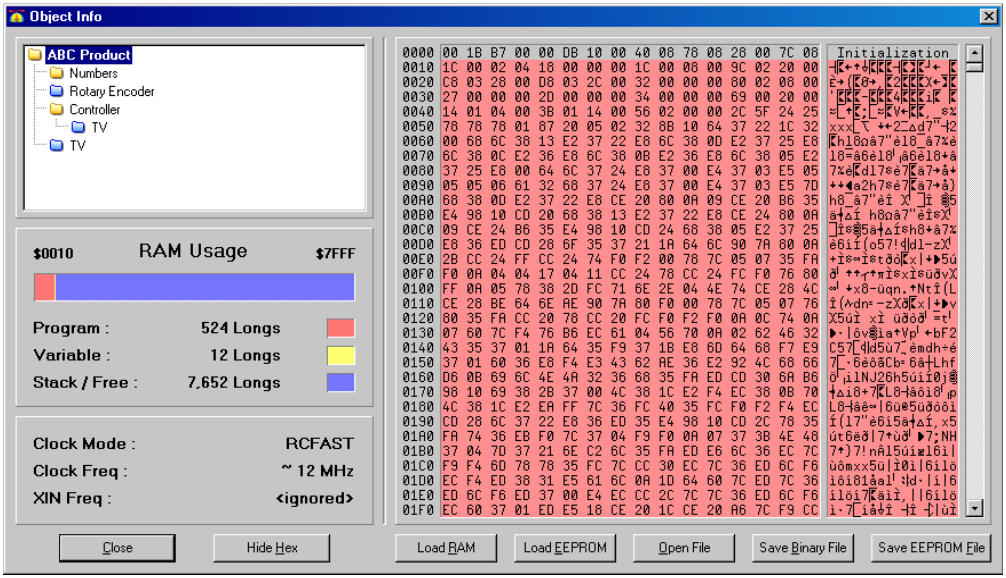


Рис. 2-12: Пример отображения «Информации об Объекте» с открытым «Hex View», отображающим шестнадцатеричные значения после компиляции «ABC Product».

Кнопки под шестнадцатеричным экраном позволяют загружать и сохранять отображаемые hex данные.

Первые две кнопки, «Загрузить ОЗУ» (Load RAM) и «Загрузить ЭСППЗУ» (Load EEPROM), выполняют те же действия, что и аналогично названные пункты меню в меню «Compile Current/Top». Важно отметить, что они используют в качестве источника загрузки текущий объект (отмеченный в окне Info Object View). Другими словами, вы можете выделить даже суб-объект в проекте и загрузить только его; но на практике эта процедура необходима только в случае если объект был разработан так, что может работать сам по себе.

Последние три кнопки, «Open File», «Save Binary File», и «Save EEPROM File» открывают либо сохраняют файл на диск. Кнопка «Open File» открывает предварительно сохраненный двоичный файл в окне «Object Info». Кнопки “save” сохраняют hex-данные текущего объекта в файл на диске. «Save Binary File» сохраняет только часть, реально используемую объектом – программу, а «Save EEPROM File» сохраняет весь образ ЭСППЗУ, включая программу, переменные, стек и свободное

пространство. Используйте команду «Save EEPROM File», если хотите получить файл, который Вы можете загрузить в ЭСППЗУ при производстве.

Таблица символов

Окно «Таблицы Символов» доступно из пункта меню «Help → View Character Chart…». Таблица Символов отображает весь набор символов шрифта Parallax, который используется программой *Propeller Tool*, а так же встроен в ПЗУ ИМС Propeller. В Таблице Символов существует три режима просмотра: 1) Стандартный, 2)Карта ПЗУ, и 3) Символьный.

В каждом из трех режимов просмотра мышь (ее левая кнопка), клавиши курсора и клавиша *Enter* могут использоваться для подсветки и выделения символа. Если был клик мыши (или нажата кнопка enter), подсвеченный символ будет введен в текущую редактируемую страницу в текущую позицию курсора. Как только новый символ подсвечен, титульная и информационная панели обновляются и показывают имя, размер и адрес выделенного символа. Вращение колеса мыши вверх и вниз изменяет размер шрифта, показанного в этом окне.

Стандартный порядок

Стандартный порядок, показанный на Рис. 2-13, показывает символы в порядке, который повторяет набор ANSI, обычно используемый в современных компьютерах.

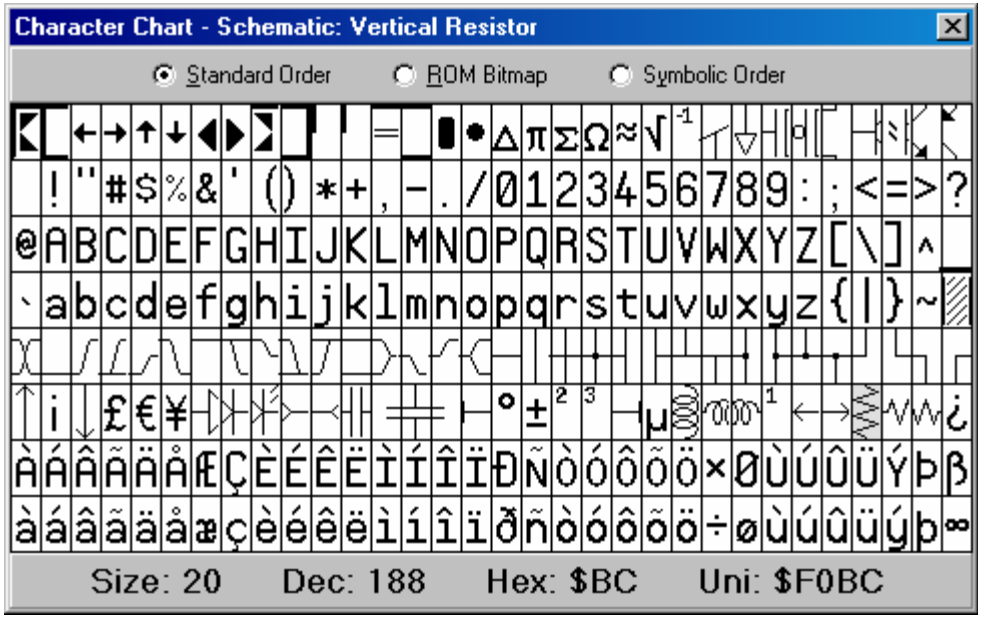


Рис. 2-13: Таблица Символов шрифта Parallax, стандартный порядок

В этом примере выбран символ вертикального резистора (в нижней правой части экрана). Информация внизу окна показывает размер шрифта, в точках, и позицию символа в таблице в десятичном, шестнадцатеричном виде и Unicode. Примечание: значение Unicode - это адрес символа в файле True Type® шрифта, который используется программой *Propeller Tool*. Десятичное и шестнадцатеричное значения являются логическими адресами символа в таблице внутри ИМС Propeller и соответствуют такому положению в наборе символов ANSI, используемом большинством компьютеров.

Карта ПЗУ

Карта ПЗУ, Рис. 2-14, показывает символы в последовательности, в которой они сохранены в ПЗУ ИМС Propeller. Для отображения битовой структуры каждого символа, в этом виде используется четыре цвета: белый, светло-серый, темно-серый и черный. В ПЗУ ИМС Propeller каждый символ определяется двумя битами цвета (четыре цвета на строку в каждом знакоместе символа). Строки каждой смежной пары символов пересекаются в памяти, в целях создания «интерактивных» символов для рисования 3D-клавиш с горячими кнопками и индикаторами фокуса; см. «Основное ПЗУ» на стр. 37. Информация в нижней части окна показывает размер шрифта, в точках, и диапазон адресов пикселей в ПЗУ ИМС Propeller для выбранного символа.

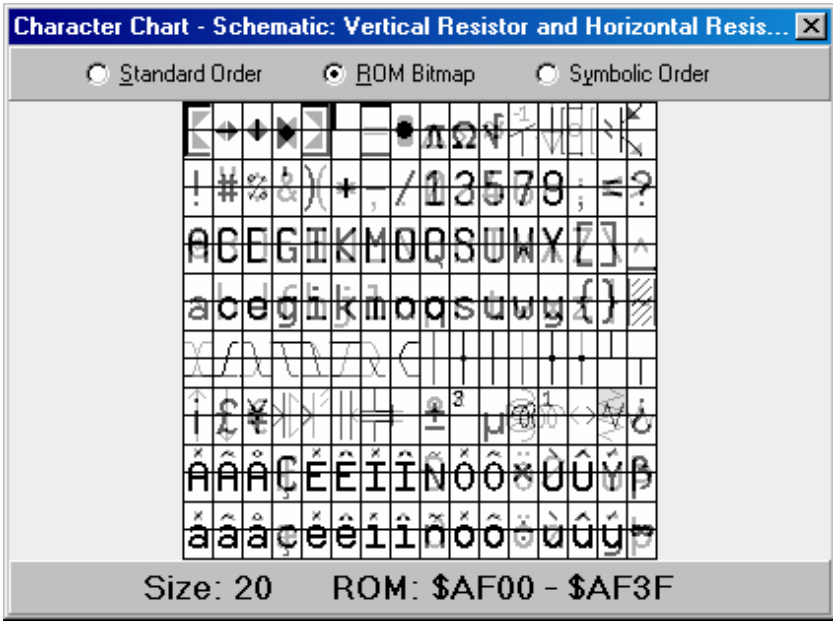


Рис. 2-14: Таблица Символов шрифта Parallax, карта ПЗУ

Символьный порядок

Символьный порядок, Рис. 2-15, отображает символы, расположенные по категориям. Такое представление полезно для возможности нахождения специальных символов в шрифте Parallax для изображения временных диаграмм, линий, стрелок и схем.

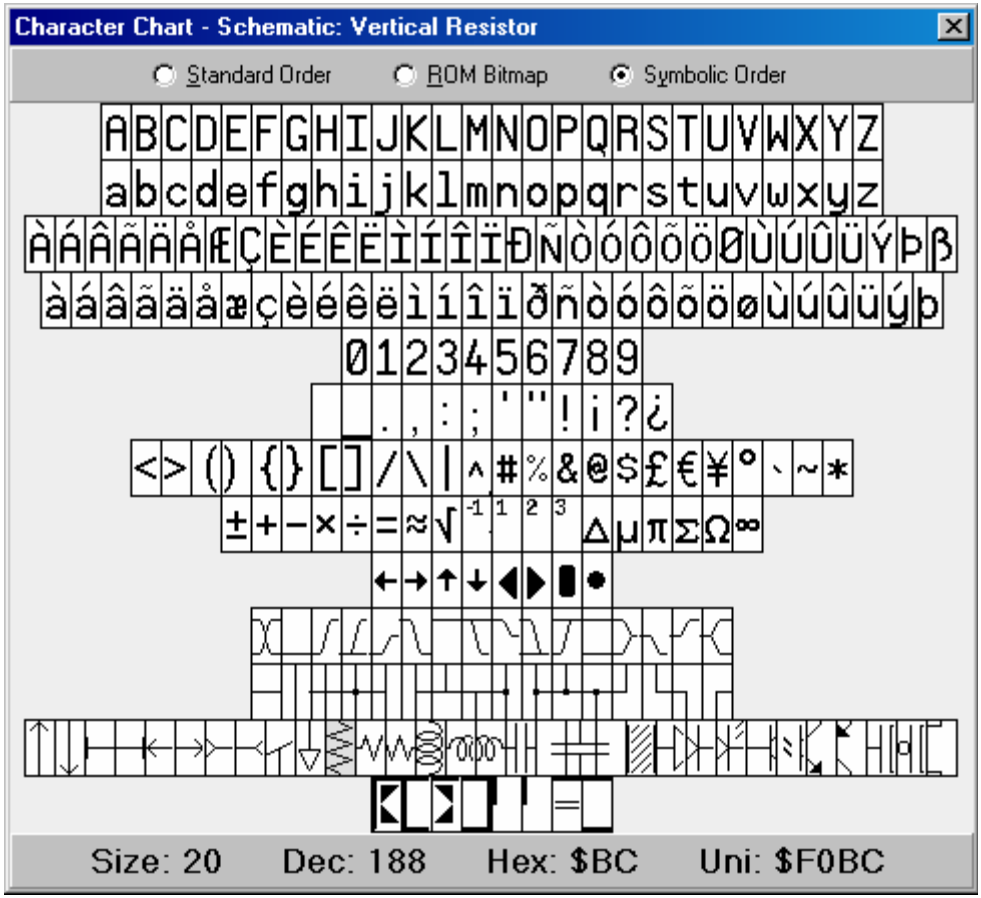


Рис. 2-15: Таблица Символов шрифта Parallax, символьный порядок

Режимы просмотра, Отметки и Номера строк

При разработке объектов, либо при их обсуждении с другими пользователями, иногда бывает сложно быстро найти нужное место в коде из-за размеров самого файла или из-за больших разделов кода и комментариев, загромождающих необходимый участок. В программе *Propeller Tool* существует множество встроенных функций, предназначенных помочь в решении этой проблемы, включая «Режимы просмотра», «Отметки» и «Нумерацию Строк».

Режимы просмотра

Каждая редактируемая вкладка может отображать исходный текст объекта в одном из четырех режимов: 1) Полный исходный код, 2) Сжатый, 3) Общий и 4) Документация.

- «Полный исходный код» отображает каждую строку исходного текста и является единственным режимом, поддерживающим редактирование.
- «Сжатый режим» скрывает каждую строку, которая содержит только комментарий к коду, а также полностью пустые строки, отображая лишь строки, пригодные для компиляции.
- «Общий режим» просмотра отображает только строки заголовков блоков (`CON`, `VAR`, `OBJ`, `PUB`, `PRI`, и `DAT`); это удобный способ наглядно увидеть общую структуру объекта.
- Режим «Документация» отображает документацию на объект, генерируемую компилятором из комментариев в исходном тексте (см. Упражнение 3: Output.spin на стр. 118 для более подробной информации).

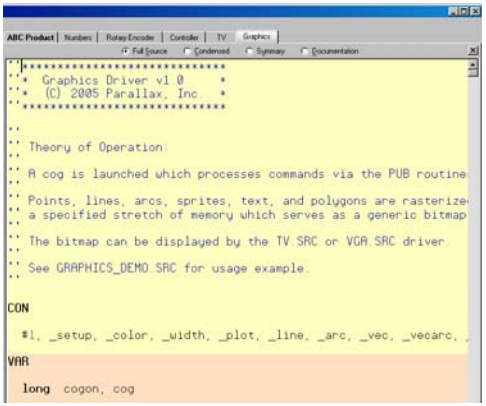
Быстро переключаясь между режимами просмотра, Вы можете находить необходимую процедуру или участок кода. Например, на Рис. 2-16a показан объект «Graphics», открытый на странице для редактирования. Если бы Вам было тяжело найти процедуру “plot” в исходном коде, Вы бы могли переключить режим просмотра в «Общий» (Рис. 2-16b), найти заголовок процедуры “plot” и кликнуть мышь на этой строке, чтобы установить на ней курсор, а затем переключиться в режим «Полный исходный код» (Рис. 2-16c). Следите за строкой, на которой установлен курсор, потому что код расширится в полный размер вверх и вниз от этой строки.

Режим просмотра может быть изменен многими способами; см. перечень «Сочетания клавиш (Shortcut Keys)», начинающийся на стр. 90. Например, находясь в любом режиме просмотра, кроме «Полного исходного кода», нажатие клавиши *Escape* вернет Вас назад в этот вид. Находясь в «Сжатом» или «Общем» режиме просмотра, двойной

клик на строке переключит режим назад в «Полный», расширяя код вверх и вниз от этой строки. Кроме того, пункты панели режима просмотра работают подобно переключателю, так что при нажатии на пункт «Общий», режим переключается поочередно между текущим режимом и режимом «Общий».

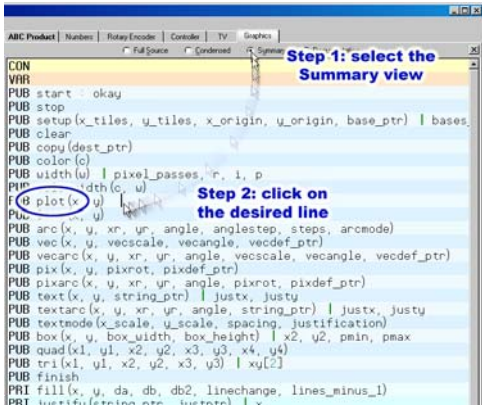
Рис. 2-16: Пример режимов просмотра

a.



Не можете найти процедуру в объекте?

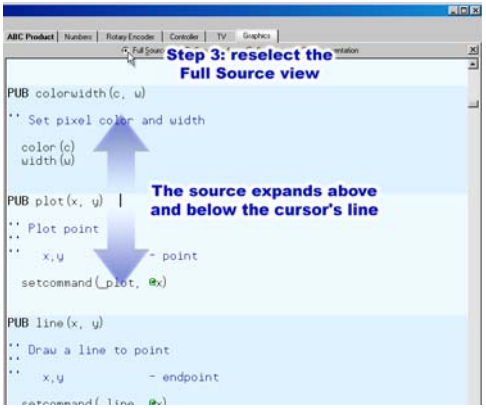
b.



Шаг 1: Выберите режим Общий.

Шаг 2: Кликните на строке процедуры.

c.



Шаг 3: Выберите опять Полный режим; код расширится по разные стороны от строки с курсором,

-либо-

Выполните двойной клик на необходимой с шага 2 строке.

Отметки

Для быстрого доступа к необходимой позиции на различных строках исходного кода каждой редактируемой страницы Вы можете устанавливать «Отметки». На Рис. 2-17 показан пример двух отметок, установленных в окне редактирования объекта «Graphics». Для включения «Отметок», нажмите *Ctrl+Shift+B*. При этом слева от редактируемой страницы появится чистая область – поле отметок. Затем кликните мышью в поле отметок возле каждой строки, к которой Вы хотите иметь быстрый доступ. В результате, при нажатии из любого места на странице *Ctrl+#* (где # - это номер отметки, к которой Вы хотите придти), курсор мгновенно перепрыгнет в эту позицию. В каждой вкладке может быть установлено до 9 отметок (1 – 9). Отметки не сохраняются в тексте, однако, установки отметок в последних 10 использованных файлах сохраняются в *Propeller Tool* и восстанавливаются при открытии этих файлов.

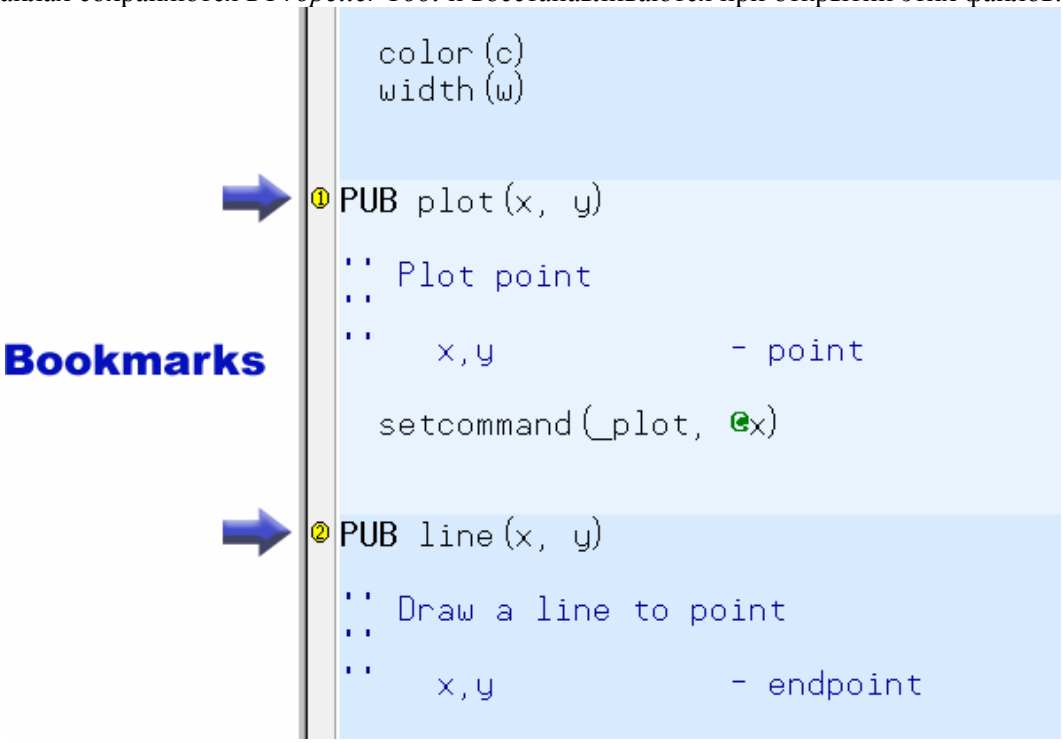


Рис. 2-17: Пример редактируемой страницы с 2-мя установленными отметками.

Кликните на Поле Отметок для установки или снятия отметок. Нажмите Ctrl+# для мгновенного доступа к существующей отметке.

Нумерация Строк

Иногда область кода легче запомнить по номеру его строки. Вы в любой момент можете включить либо выключить нумерацию строк в редактируемой странице. Номера строк показываются в «Поле номеров строк», рядом с «Полем отметок» (см. Рис. 2-18), и могут быть сделаны видимыми/невидимыми нажатием *Ctrl+Shift+N*. Строки автоматически нумеруются при их создании; номера – это только визуальные элементы и они не сохраняются в исходном тексте. Хотя номера строк и соседствуют с отметками, они независимы друг от друга и могут быть включены либо выключены отдельно. При необходимости, номера строк могут быть распечатаны.

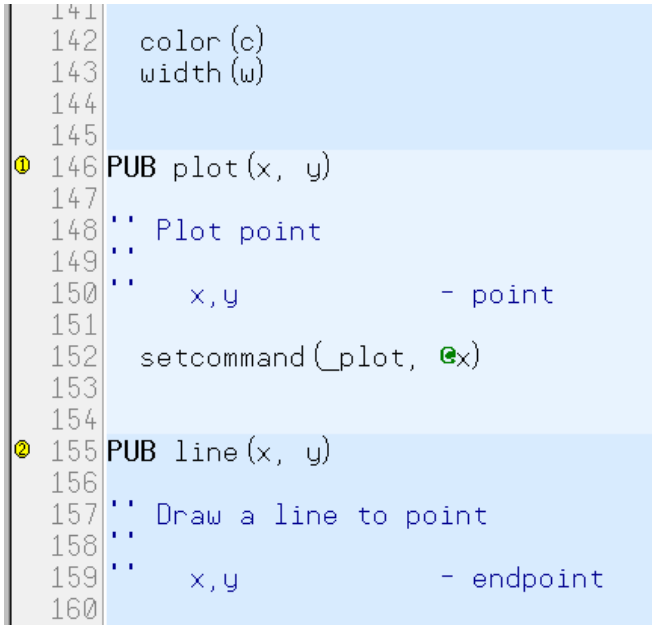
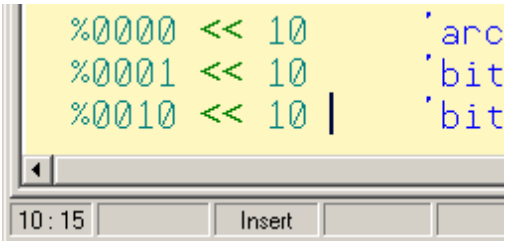


Рис. 2-18: Пример страницы редактирования с включенными отметками и нумерацией строк.

Режимы редактирования

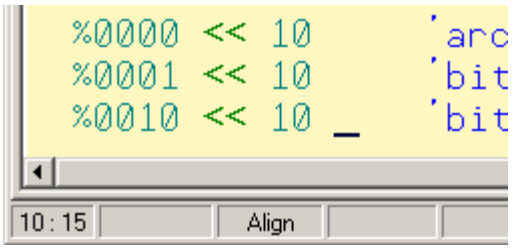
Панель редактирования обеспечивает возможность редактирования в одном из трех режимов: 1) Вставка (по умолчанию), 2) Выравнивание (только для объектов ".spin"), и 3) Замена. Вы можете переключаться между режимами, используя клавишу Insert. Текущий режим отражается как видом курсора, так и видом панели 3 строки статуса.

Рис. 2-19: Режимы редактирования



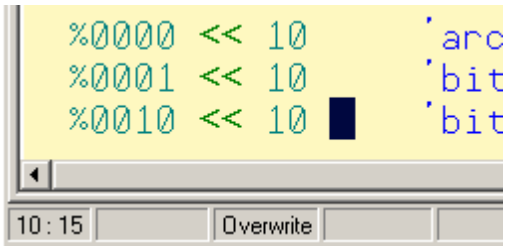
Режим Вставка

Курсор – стандартный мигающий, вертикальная линия, и строка статуса индицируют "Insert."



Режим Выравнивание

Курсор мигает подчеркиванием, и строка статуса индицирует "Align."



Режим Замена

Курсор мигает, сплошной блок, и статус индицирует "Overwrite."

Режимы Вставка и Замена

Режимы «Вставка» и «Замена» похожи на таковые во множестве других текстовых редакторов. Это единственные два режима, доступные при редактировании во вкладках, содержащих не Propeller ".spin"-объекты, а такие, например, как файлы ".txt".

Режим Выравнивание

Режим «Выравнивание» – это особый вариант режима «Вставки», разработанный специально для поддержки и написания исходного текста. Чтобы понять режим «Выравнивание», мы сначала должны рассмотреть общие приемы написания текста программ. Существует два общепринятых механизма при написании современного исходного текста: структурирование кода и выравнивание комментариев справа от кода. Так же обычным является то, что исходный код может просматриваться и редактироваться различными текстовыми редакторами. Исторически, для структурирования и выравнивания, программисты пользовались либо табуляцией, либо пробелами, причем оба метода имеют недостатки. Символы табуляции вызывают проблемы из-за различных размеров табуляции, установленных в различных редакторах. Как символы табуляции, так и пробелы вызывают проблемы выравнивания, так как последующее редактирование приведет расположенные справа комментарии к сдвижке за границы выравнивания. Вот несколько примеров. Рис. 2-20 - это наш оригинальный код.

Рис. 2-20: Общие приемы Выравнивания– оригинальный код

```
PRI CheckButton
  if not INA[11]                    'Button pressed
    waitcnt(Delay+cnt)              'debounce
    if not INA[11]                  'Still pressed
      repeat until INA[11]          'wait for release
      waitcnt(Delay+cnt)            'debounce
      Mode++
      PrintMode
```

Если в этом коде использовались символы табуляции для выравнивания комментариев, изменение "Delay" на "BtnDelay" приведет к смещению комментариев вправо, если введенный текст пересечет границы табуляции.

Рис. 2-21: Общие приемы Выравнивания – выравнивание табуляцией

```
PRI CheckButton
  if not INA[11]                    'Button pressed
    waitcnt(BtnDelay+cnt)           'debounce
    if not INA[11]                  'Still pressed
      repeat until INA[11]          'wait for release
      waitcnt(BtnDelay+cnt)         'debounce
      Mode++
      PrintMode
```

Если в оригинальном коде были использованы символы табуляции для выравнивания комментариев, изменение "Delay" на "BtnDelay" приводит к тому, что второй комментарий неожиданно смещается вправо на еще один размер табуляции.

Если в оригинальном коде для выравнивания использовались пробелы, изменение "Delay" на "BtnDelay" приведет к смещению комментариев вправо на три символа.

Рис. 2-222: Общие приемы Выравнивания – Выравнивание пробелами

```
PRI CheckButton
  if not INA[11]                'Button pressed
    waitcnt(BtnDelay+cnt)       '    debounce
    if not INA[11]              'Still pressed
      repeat until INA[11]      'wait for release
      waitcnt(BtnDelay+cnt)     '    debounce
      Mode++
      PrintMode
```

Если в оригинальном коде были использованы пробелы для выравнивания комментариев, и используется стандартный режим редактирования Вставка, изменение "Delay" на "BtnDelay" приводит к смещению второй и пятой строк на три пробела вправо.

Для кода *Spin*, редактор в *Propeller Tool* решает эту проблему во-первых, запрещением символов табуляции (нажатие клавиши *Tab* вводит соответствующее количество пробелов), и, во-вторых, предоставлением режима редактирования «Выравнивание». Находясь в режиме «Выравнивание», символы, вводимые в строку, влияют на свои соседние символы, но не на те, которые отделены более чем одним пробелом. В результате, комментарии и другие примитивы, отделенные более чем одним пробелом, сохраняют свое положение максимально долго, как показано на Рис. 2-23.

Рис. 2-23: Эффекты режима Выравнивания

```
PRI CheckButton
  if not INA[11]                'Button pressed
    waitcnt(BtnDelay+cnt)       'debounce
    if not INA[11]              'Still pressed
      repeat until INA[11]      'wait for release
      waitcnt(BtnDelay+cnt)     'debounce
      Mode++
      PrintMode
```

При использовании режима Выравнивания, изменение "Delay" на "BtnDelay" оставляет все комментарии на их первоначальной, выровненной позиции. В этом случае нет необходимости в ручном повторном пере-выравнивании.

Поскольку режим «Выравнивание» сохраняет максимально возможный объем выравниваний, при следующем редактировании программистом тратится намного меньше времени на повторное выравнивание кода. Вдобавок, так как вместо табуляций используются пробелы, код сохраняет один и тот же вид в любом редакторе, который отобразит его как текст, разделенный пробелами.

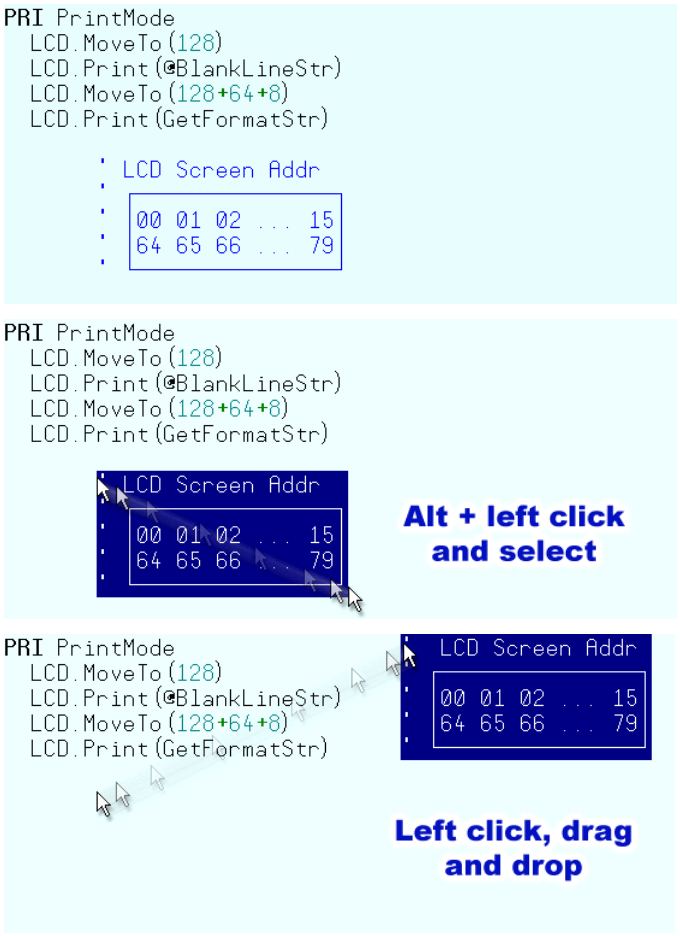
Однако режим «Выравнивание» идеален не для всех случаев. Для написания кода рекомендуется использовать режим «Вставки», и ненадолго переключаться в режим Выравнивания для оформления существующего кода, где важно выравнивание. Клавиша *Insert* переключает режим по очереди: Insert → Align → Overwrite и далее опять Insert. Клавиши *Ctrl+Insert* переключают режим лишь между Insert и Align. Небольшая практика использования режимов Align и Insert поможет Вам писать программы с меньшими затратами времени.

Отметьте, что не-*Spin* источник (без расширения .spin) не позволяет использовать режим «Выравнивание». Так происходит из-за того, что для не-*Spin* источников программа *Propeller Tool* сохраняет все существующие в них символы табуляции, и вводит символ *tab* при нажатии клавиши *Tab*, чтобы сохранить оригинальное назначение файла, который может представлять собой разделенный табуляцией источник данных для *Spin*-программ либо других применений, где табуляция важна.

Выделение и перемещение блока

Кроме обычного выделения текста при помощи мыши, программа *Propeller Tool* позволяет блочное выделение (прямоугольные области текста). Чтобы выделить блок, сначала нажмите и удерживайте клавишу *Alt*, затем выполните левый клик и потяните мышь для выделения области текста. После того, как блок выделен, операции вырезания и копирования ведут себя, как и при других выделениях. Рис. 2-24 демонстрирует блочное выделение и перемещение блока текста при помощи мыши.

Рис. 2-24: Блочное выделение и перемещение выделенного фрагмента



Оригинальный код. Нам бы хотелось переместить комментарии "LCD Screen Addr" вправо от процедуры `PrintMode`***.***

Сначала нажмите и удерживайте клавишу Alt. Затем сделайте левый клик и потяните мышью для выделения.

В завершение, кликните и потяните (из любого места в выделенной области) и оставьте выделенное в необходимом месте.

Отступы и Выступы

Обычной практикой программирования, для облегчения читаемости кода, является структурирование блоков кода, находящихся в циклах либо в условном выполнении. Подобное действие называется введением Отступов. Будем называть противоположное действие, т.е. сдвиг текста влево как введение Выступов. Язык *Spin* требует такого рода форматирование для отображения, какие строки находятся внутри циклов или условных блоков. Программа *Propeller Tool* включает следующие функции для облегчения достижения результата при создании или редактировании кода.

Одиночные Строки (Single Lines)

Для кода *Spin* программа *Propeller Tool* использует набор фиксированных позиций табуляции, которые Вы можете изменить посредством меню Edit → Preferences. Каждый блок в *Spin* (`CON`, `VAR`, `OBJ`, `PUB`, `PRI`, и `DAT`) имеет свои собственные установки табуляции.

Клавиша *Tab* перемещает курсор в следующую позицию табуляции (вправо), а Shift + Tab перемещает курсор в предыдущую позицию табуляции (влево). Кроме того, клавиша *Backspace* перемещает курсор в предыдущую позицию, в зависимости от текста вокруг; подробнее об этом позднее.

Установки табуляции по умолчанию для блоков `PUB` и `PRI` включают позиции для каждых двух символов в начале строки для поддержки обычной структуризации кода. Например, ниже на Рис. 2-25 показан *Public*-метод `FSqr`, содержащий строки на разных уровнях отступа, каждый уровень на два символа друг от друга.

```
PUB FSqr
  repeat 31
    result |= root
    if result ** result > m
      result ^= root
    root >>= 1
  m := result >> 1
```

Рис. 2-25: Фиксированные установки по умолчанию для блоков PUB и PRI

Этот код можно быстро ввести, используя клавишу Tab, при следующей последовательности ввода с клавиатуры:

- Ввести: “PUB FSqr” <Enter>
- Ввести: <Tab> “repeat 31” <Enter>
- Ввести: <Tab> “result |= root” <Enter>, и т.д.

Отметьте, что клавиша *Enter* автоматически выравнивает курсор на текущий уровень отступа; это значит, что требуется только одно нажатие клавиши *Tab* для Отступа на следующий уровень.

Если при нажатии клавиши *Tab* справа от курсора находятся символы, они также смещаются вправо, как на Рис. 2-26.

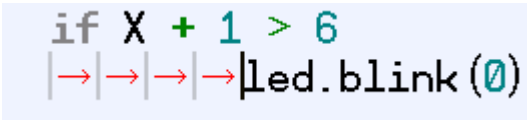


Рис. 2-26: Отступ

Если курсор находится непосредственно слева от первого символа в строке, то как клавиши *Shift* + *Tab*, так и *Backspace* приведут к смещению и курсора, и текста влево на предыдущую позицию табуляции; т.е. к Выступу. Если, однако, курсор находится не сразу слева от первого символа в строке, клавиша *Backspace* работает как обычно (удаляя предыдущий символ), а *Shift* + *Tab* перемещают в предыдущую позицию только курсор.

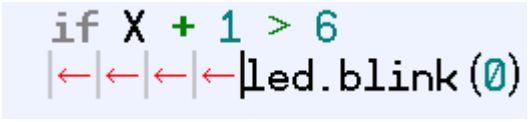


Рис. 2-27: Выступ

Несколько строк (Multiple Lines)

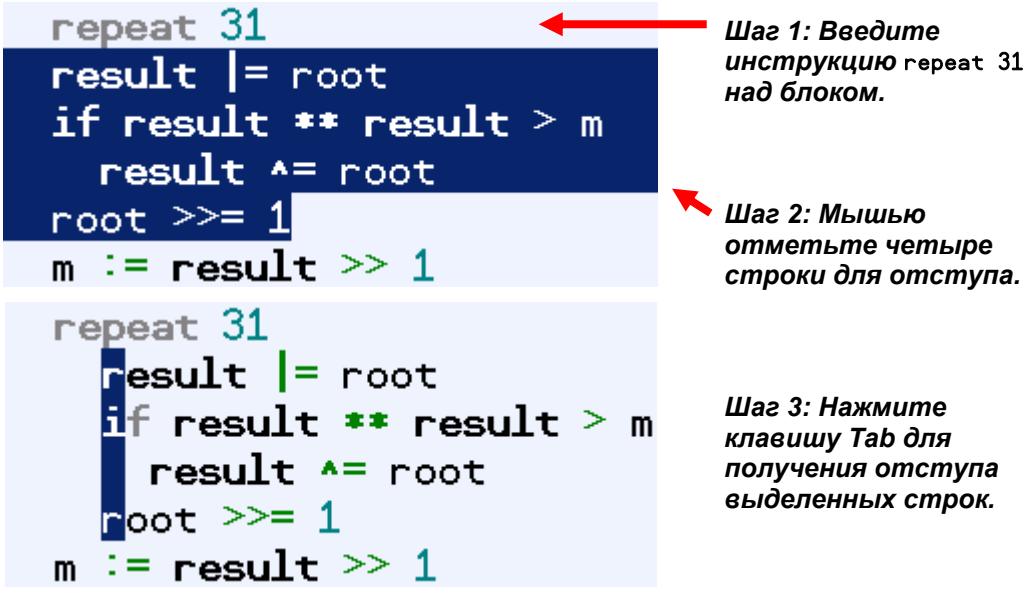
Как и с одиночными строками, отступ либо выступ на фиксированные позиции так же легко выполняется и для нескольких строк. Посмотрите на Рис. 2-28.

```
result |= root
if result ** result > m
  result ^= root
root >>= 1
m := result >> 1
```

Рис. 2-28: Пример блока кода. Мы хотим, чтобы первые четыре строки повторились 31 раз.

Предположим, что нам хочется взять первые четыре строки из этого примера и поместить их в цикл повтора – для повтора этих строк 31 раз. Вы можете быстро достигнуть этого, используя следующие шаги: 1) введите строку “repeat 31” выше существующих строк, 2) используя мышь, выделите четыре строки для отступа, и 3) нажмите клавишу Tab. Эти шаги показаны на Рис. 2-29.

Рис. 2-29: Code Block Indenting



Отметьте, что четыре строки, которые мы выделили во втором шаге, находятся сейчас в следующей фиксированной позиции табуляции (два пробела справа от начала “repeat”) и выделение сменилось на один столбец, указывающий на первые символы строк. Выделение изменилось для индикации, что мы выполнили отступ нескольких строк. Повторное нажатие клавиши *Tab* приведет к дальнейшему отступу выделенной группы строк, а нажатие *Shift* + *Tab* приведет к выступу группы этих строк.

Любая группа смежных строк может быть подвергнута отступу либо выступу подобным образом. Само выделение, однако, не обязательно должно включать целую строку; для корректной работы оно должно включать как минимум один символ более чем одной строки. Такой тип выделения называется поточным выделением (“stream”).

Второй тип выделения – блочное выделение (см. «Выделение и перемещение блока », стр. 83), также может быть использован для получения выступов и отступов нескольких строк. Например, на Рис. 2-30 показан пример с комментариями справа от строк кода.

Рис. 2-30: Пример блока кода с комментариями справа

```
repeat 31                           'loop 31 times
  result |= root                    'OR result w/root
  if result ** result > m           'calculate square
    result ^= root
  root >>= 1                        'shift root right
m := result >> 1
```

Если мы выделим первые несколько символов комментариев блоком (*Alt* + Левая кнопка мыши и потянуть, Рис. 2-31), мы можем нажать клавишу Tab для получения отступа этих комментариев на следующую позицию табуляции. Нажатие Shift + Tab приведет к их выступу, как минимум до любого из символов, на который они натолкнутся слева, как на Рис. 2-32.

Рис. 2-31: Использование блочного выделения для выступа комментариев

```
repeat 31                           'loop 31 times
  result |= root                    'OR result w/root
  if result ** result > m           'calculate square
    result ^= root
  root >>= 1                        'shift root right
m := result >> 1
```

Шаг 1:Выделите блоком строки комментариев (Alt + Левая кнопка мыши и потянуть).

```
repeat 31                           'loop 31 times
  result |= root                    'OR result w/root
  if result ** result > m           'calculate square
    result ^= root
  root >>= 1                        'shift root right
m := result >> 1
```

Шаг 2: Нажмите клавишу Tab для получения выступа комментариев.

Индикаторы Блок-Групп

Иногда сложно увидеть, как группы кода логически организованы только по их уровню отступа. Программа *Propeller Tool* может, при необходимости, индицировать логические блок-группы условно выполняемых блоков либо блоков циклов, при помощи Индикаторов блок-групп, как показано на Рис. 2-32. Для включения/выключения этой функции нажмите *Ctrl+I*.

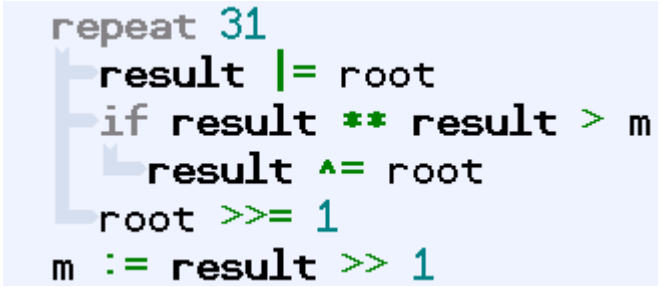


Рис. 2-32: Индикаторы Блок-Группы

Отметьте, что такими индикаторами отступа дополняется лишь компилируемый код, который находится внутри условно выполнимого блока либо цикла. Такое представление служит визуальной помощью, позволяющей видеть, как будет выполняться код. В то же время сами индикаторы не влияют на код либо исходный файл физически, что могут делать только сами выступы или отступы.

Сочетания клавиш (Shortcut Keys)

Перечень по функциям

В Табл. 2-1 сочетания клавиш сгруппированы по выполняемым функциям. В Табл. 2-6, которая начинается со стр. 95, сочетания клавиш сгруппированы не по функциям, а по клавишам.

Табл. 2-1: Сочетания клавиш – Перечень по функциям	
Сочетания в Инструментах	
Функция	**Клавиши**
Новый	Ctrl + N
Открыть	Ctrl + O
Закрыть	Alt + Q -или- Ctrl + W
Сохранить	Ctrl + S
Сохранить все	Ctrl + Alt + S
Печать	Ctrl + P
Показать/Скрыть Отметки	Ctrl + Shift + B
Установить Отметку на текущей строке	Ctrl + B
Включить Индикаторы Блок-Групп	Ctrl + I
Показать/Скрыть Браузер	Ctrl + E
Показать/Скрыть Номера Строк	Ctrl + Shift + N
Увеличить размер шрифта	Ctrl + Вверх -или- Ctrl + Колесо Мыши Вверх
Уменьшить размер шрифта	Ctrl + Вниз -или- Ctrl + Колесо Мыши Вниз
Выбрать просмотр Всего исходного кода	Alt + S
Выбрать Сжатый режим просмотра	Alt + C
Выбрать Общий режим просмотра	Alt + U
Выбрать режим просмотра Документации	Alt + D
Выбрать следующ. вид (в сторону Всего кода)	Alt + Вверх
Выбрать следующий вид (в сторону Документ.)	Alt + Вниз-или- Alt + Колесо Мыши Вниз
Установить фокус на активной странице	Esc

Табл. 2-2: Сочетания клавиш – Перечень по функциям (продолжение)	
Сочетания Компилятора	
Функция	**Клавиши**
Выбрать верхний файл	Ctrl + T
Определить аппаратуру	F7
Компилировать текущий файл и Смотреть информацию	F8
Компилировать текущий файл и Обновить Статус	F9
Компилировать текущий файл, Загрузить ОЗУ и Запустить	F10
Компилировать текущий файл, Загрузить ЕППЗУ и Запустить	F11
Компилировать верхний файл и Смотреть информацию	Ctrl + F8
Компилировать верхний файл и Обновить Статус	Ctrl + F9
Компилировать верхний файл, Загрузить ОЗУ и Запустить	Ctrl + F10
Компилировать верхний файл, Загрузить ЕППЗУ и Запустить	Ctrl + F11
Сочетания для Навигации	
Выбрать следующую вкладку редактирования	Alt + Лево -или- Ctrl + Tab
Выбрать предыдущую вкладку редактирования	Alt + Лево -или- Ctrl + Shift + Tab
Пролистать одну страницу вверх	Page Up
Пролистать одну страницу вниз	Page Down
Сдвинуться влево	Shift + Колесо Мыши Вверх
Сдвинуться вправо	Shift + Колесо Мыши Вверх
Перейти в начало следующего слова	Ctrl + Вправо
Перейти в начало предыдущего слова	Ctrl + Влево
Перейти к началу строки	Home
Перейти в конец строки	End
Перейти к началу страницы	Ctrl + Page Up
Перейти в конец страницы	Ctrl + Page Down
Перейти к началу файла	Ctrl + Home
Перейти в конец файла	Ctrl + End

Табл. 2-3: Сочетания клавиш – Перечень по функциям (продолжение)	
Сочетания для навигации (продолжение)	
Функция	**Клавиши**
Выделить слово	Двойной Клик
Выделить строку	Тройной Клик
Выделить до начала следующего слова	Ctrl + Shift + Вправо
Выделить до начала предыдущего слова	Ctrl + Shift + Влево
Выделить до начала строки	Shift + Home
Выделить до конца строки	Shift + End
Выделить до начала страницы	Ctrl + Shift + Page Up
Выделить до конца страницы	Ctrl + Shift + Page Down
Выделить до предыдущей страницы сверху	Shift + Page Up
Выделить до следующей страницы снизу	Shift + Page Down
Выделить до начала файла	Ctrl + Shift + Home
Выделить до конца файла	Ctrl + Shift + End
Сочетания для Редактирования	
Отменить	Ctrl + Z
Повторить	Ctrl + Shift + Z
Выделить все	Ctrl + A
Копировать в буфер обмена	Ctrl + C
Вырезать в буфер обмена	Ctrl + X
Вставить из буфера обмена	Ctrl + V
Найти / Заменить	Ctrl + F
Найти далее	F3
Заменить	F4
Заменить и Найти далее	Ctrl + F4
Смена режима на Выравнивание, Вставку или Замену	Insert
Сменить режим между Выравниванием и Вставкой	Ctrl + Insert
Вставить пробелы до следующей позиции табуляции	Tab
Удалить пробелы до предыдущей позиции табуляции	Shift + Tab

Табл. 2-4: Сочетания клавиш – Перечень по функциям (продолжение)	
Сочетания для Редактирования (продолжение)	
Функция	**Клавиши**
Удалить текущую линию	Ctrl + Y
Удалить до конца строки	Ctrl + Shift + Y
Переименовать Папку/Файл (в Списке)	F2
Сочетания для ввода Символов	
Вставить символ степени отрицательной единицы ($^{-1}$)	Ctrl + Alt + 1
Вставить символ степени единицы (^{1})	Ctrl + Shift + 1
Вставить символ степени два (^{2})	Ctrl + Shift + 2
Вставить символ степени три (^{3})	Ctrl + Shift + 3
Вставить символ маркера (•)	Ctrl + Shift + .
Вставить символ прямоугольного маркера (▮)	Ctrl + Alt + .
Вставить символ левого маркера(◀)	Ctrl + Shift + Alt + <
Вставить символ правого маркера (▶)	Ctrl + Shift + Alt + >
Вставить символ маркера стрелки влево (←)	Ctrl + Shift + Alt + Влево
Вставить символ маркера стрелки вправо (→)	Ctrl + Shift + Alt + Вправо
Вставить символ маркера стрелки вверх (↑)	Ctrl + Shift + Alt + Вверх
Вставить символ маркера стрелки вправо (→)	Ctrl + Shift + Alt + Вправо
Вставить символ Евро(€)	Ctrl + Shift + $
Вставить символ Йены (¥)	Ctrl + Alt + $
Вставить символ Фунта Стерлинга (£)	Ctrl + Shift + Alt + $
Вставить символ стрелки влево (←)	Ctrl + Alt + Влево
Вставить символ стрелки вправо (→)	Ctrl + Alt + Вправо
Вставить символ стрелки вверх (↑)	Ctrl + Alt + Вверх
Вставить символ стрелки вниз(↓)	Ctrl + Alt + Вниз
Вставить символ градуса Цельсия (°)	Ctrl + Shift + %
Вставить символ плюс/минус (±)	Ctrl + Shift + -

Табл. 2-5: Сочетания клавиш – Перечень по функциям (продолжение)	
Сочетания для ввода Символов (продолжение)	
Функция	**Клавиши**
Вставить символ умножения (×)	Ctrl + Shift + *
Вставить символ деления (÷)	Ctrl + Shift + /
Вставить символ корня (√)	Ctrl + Shift + R
Вставить символ бесконечности (∞)	Ctrl + Shift + I
Вставить символ Дельта (Δ)	Ctrl + Shift + D
Вставить символ Мю (μ)	Ctrl + Shift + M
Вставить символ Омега(Ω)	Ctrl + Shift + O
Вставить символ Пи(π)	Ctrl + Shift + P
Вставить символ Сигма(Σ)	Ctrl + Shift + S

Перечень по клавишам

Табл. 2-6: Сочетания клавиш – перечень по клавишам	
Одиночная клавиша или клик мыши	
Клавиша	**Функция**
F2	Переименовать Папку/Файл (в Списке)
F3	Найти далее
F4	Заменить
F7	Определить аппаратуру
F8	Компилировать текущий файл и Смотреть информацию
F9	Компилировать текущий файл и Обновить Статус
F10	Компилировать текущий файл, Загрузить ОЗУ и Запустить
F11	Компилировать текущий файл, Загрузить ЕППЗУ и Запустить
End	Перейти в конец строки
Esc	Установить фокус на активной странице
Home	Перейти к началу строки
Insert	Смена режима на Выравнивание, Вставку или Замену
Page Down	Пролистать одну страницу вниз
Page Up	Пролистать одну страницу вверх
Tab	Вставить пробелы до следующей позиции табуляции
Двойной Клик	Выделить слово
Тройной Клик	Выделить строку
Ctrl + …	
Ctrl + A	Выделить все
Ctrl + B	Установить отметку на текущей строке
Ctrl + C	Копировать в буфер обмена
Ctrl + E	Скрыть/Показать браузер
Ctrl + F	Найти/Заменить
Ctrl + I	Включить индикацию Блок-Групп

Табл. 2-7: Сочетания клавиш – перечень по клавишам (продолжение)	
Ctrl + … (продолжение)	
Клавиши	**Функция**
Ctrl + N	Новый
Ctrl + O	Открыть
Ctrl + S	Сохранить
Ctrl + P	Печать
Ctrl + T	Выбрать верхний файл
Ctrl + V	Вставить из буфера обмена
Ctrl + W	Закрыть
Ctrl + X	Вырезать в буфер обмена
Ctrl + Y	Удалить текущую строку
Ctrl + Z	Отменить
Ctrl + F4	Заменить и Найти далее
Ctrl + F8	Компилировать верхний файл и Смотреть информацию
Ctrl + F9	Компилировать верхний файл и обновить Статус
Ctrl + F10	Компилировать верхний файл, Загрузить ОЗУ и Выполнить
Ctrl + F11	Компилировать верхний файл, Загрузить ЭСППЗУ и Выполнить
Ctrl + F4	Заменить и Найти далее
Ctrl + Down	Уменьшить размер Шрифта
Ctrl + End	Перейти в конец файла
Ctrl + Home	Перейти к началу файла
Ctrl + Insert	Сменить режим между Выравниванием и Вставкой
Ctrl + Left	Перейти к началу предыдущего слова
Ctrl + Page Down	Перейти к концу страницы
Ctrl + Колесо Мыши Вниз	Уменьшить размер шрифта
Ctrl + Колесо Мыши Вверх	Увеличить размер шрифта
Ctrl + Up	Увеличить размер шрифта

Табл. 2-8: Сочетания клавиш – перечень по клавишам (продолжение)	
Alt + …	
Alt + C	Выбрать Сжатый режим просмотра
Alt + D	Выбрать режим просмотра Документации
Alt + S	Выбрать режим просмотра Всего исходного кода
Alt + Q	Закрыть
Alt + U	Выбрать режим просмотра Общий
Alt + Down	Выбрать следующ. вид (в сторону вида Документации)
Alt + Left	Выбрать предыдущую вкладку для редактирования
Alt + Mouse Wheel Down	Выбрать следующ. вид (в сторону вида Документации)
Alt + Mouse Wheel Up	Выбрать следующ. вид (в сторону Всего кода)
Alt + Right	Выбрать следующую вкладку для редактирования
Alt + Up	Выбрать следующ. вид (в сторону Всего кода)
Shift + …	
Shift + End	Выделить до конца строки
Shift + Home	Выделить до начала строки
Shift + Page Down	Выделить до следующей страницы вниз
Shift + Page Up	Выделить до предыдущей страницы вверх
Shift + Tab	Удалить пробелы до предыдущей позиции табуляции
Shift + Колесо Мыши Вниз	Сместиться вправо
Shift + Колесо Мыши Вверх	Сместиться влево
Ctrl + Alt + …	
Ctrl + Alt + .	Вставить символ прямоугольного маркера (▮)
Ctrl + Alt + $	Вставить символ Йены (¥)
Ctrl + Alt + 1	Вставить символ степени минус один ($^{-1}$)
Ctrl + Alt + S	Сохранить все
Ctrl + Alt + Down	Вставить символ стрелки вниз (↓)
Ctrl + Alt + Left	Вставить символ стрелки влево (←)
Ctrl + Alt + Right	Вставить символ стрелки вправо (→)
Ctrl + Alt + Up	Вставить символ стрелки вверх (↑)

Табл. 2-9: Сочетания клавиш – перечень по клавишам (продолжение)	
Ctrl + Shift + …	
Клавиши	**Функция**
Ctrl + Shift + $	Вставить символ Евро (€)
Ctrl + Shift + %	Вставить символ градус Цельсия (°)
Ctrl + Shift + *	Вставить символ умножения (×)
Ctrl + Shift + -	Вставить символ плюс/минус (±)
Ctrl + Shift + .	Вставить символ маркера (•)
Ctrl + Shift + /	Вставить символ деления (÷)
Ctrl + Shift + =	Вставить символ приблизительно равно (≈)
Ctrl + Shift + 1	Вставить символ степени один (^{1})
Ctrl + Shift + 2	Вставить символ степени два (^{2})
Ctrl + Shift + 3	Вставить символ степени три (^{3})
Ctrl + Shift + B	Показать/Скрыть Отметки
Ctrl + Shift + D	Вставить символ Дельта (Δ)
Ctrl + Shift + I	Вставить символ бесконечности (∞)
Ctrl + Shift + M	Вставить символ Мю (μ)
Ctrl + Shift + N	Показать/Скрыть Номера Строк
Ctrl + Shift + O	Вставить символ Омега (Ω)
Ctrl + Shift + P	Вставить символ Пи (π)
Ctrl + Shift + R	Вставить символ корня(√)
Ctrl + Shift + S	Вставить символ Сигма(Σ)
Ctrl + Shift + Y	Удалить до конца строки
Ctrl + Shift + Z	Повторить
Ctrl + Shift + End	Выделить до конца файла
Ctrl + Shift + Home	Выделить до начала файла
Ctrl + Shift + Left	Выделить до начала предыдущего слова
Ctrl + Shift + Page Down	Выделить до конца страницы
Ctrl + Shift + Page Up	Выделить до начала страницы
Ctrl + Shift + Right	Выделить до начала следующего слова

Ctrl + Shift + … (cont.)	
Клавиши	**Функция**
Ctrl + Shift + Tab	Выбрать предыдущую вкладку для редактирования

Табл. 2-10: Сочетания клавиш – перечень по клавишам (продолжение)

Ctrl + Shift + Alt…	
Клавиши	**Функция**
Ctrl + Shift + Alt + $	Вставить символ Фунта Стерлинга (£)
Ctrl + Shift + Alt + <	Вставить символ левого маркера (◀)
Ctrl + Shift + Alt + >	Вставить символ правого маркера (▶)
Ctrl + Shift + Alt + Down	Вставить символ маркера стрелки вниз (↓)
Ctrl + Shift + Alt + Left	Вставить символ маркера стрелки влево(←)
Ctrl + Shift + Alt + Right	Вставить символ маркера стрелки вправо (→)
Ctrl + Shift + Alt + Up	Вставить символ маркера стрелки вверх (↑)

Глава 3: Программирование ИМС Propeller

Эта глава написана с предположением, что Вы хорошо знакомы с основными концепциями программирования на других языках программирования, включая объектно-ориентированные языки. Хотя здесь и будет приведено обсуждение некоторых базовых концепций, однако предварительные знания и опыт программирования все же желательны.

Вдобавок к вышесказанному, этот материал рекомендуется читать только после прочтения Глав 1 и 2. Если Вы не прочитали, по крайней мере, всю Главу 1 и большую часть Главы 2, пожалуйста, сделайте это перед чтением этой главы. Здесь будут использоваться многие, описанные ранее и более не раскрываемые детально, моменты.

Нижеследующее Описание программирования ИМС Propeller описывает концепции программирования кристалла Propeller шаг-за-шагом, с предоставлением кратких примечаний на протяжении всего курса. Лучшим вариантом было бы прочесть эту главу с ее начала до конца, без пропусков, при этом работая с компьютером и ИМС Propeller и прорабатывая каждый приведенный пример. Самые первые упражнения являются базовыми, но каждое последующее раскрывает более глубокий материал.

Общие положения

Контроллер Propeller (его аппаратное, встроенное и программное обеспечения) был разработан с использованием как многих известных, так и множества инновационных концепций. С этой точки зрения можно сказать, что нами «с чистого листа» разработаны аппаратное, встроенное и программное обеспечения, а так же два языка программирования для его сопровождения (*Spin* и *Propeller*-Ассемблер), чтобы предоставить пользователям наиболее простой и эффективный путь управления контроллером Propeller.

Для полного понимания и эффективного использования этих инструментов и языков, Вам следует подойти к ним с особым, открытым осознанием. Другими словами, постарайтесь не позволить стандартным концепциям и методам программирования отстранить Вас от использования преимуществ, предоставляемых ИМС Propeller и её языков программирования. Мы полагаем, что некоторые такие общепринятые концепции на самом деле не предназначены для использования в реальном масштабе времени, они скорее вносят лишний беспорядок тем, кто ими пользуется.

Языки ИМС Propeller (Spin и Propeller Ассемблер)

ИМС Propeller программируется с помощью двух языков программирования, разработанных специально для нее: 1) язык *Spin*, ─ объектно-ориентированный язык верхнего уровня, и 2) *Propeller*-Ассемблер, ─ высоко-оптимизированный язык нижнего уровня. В языке ассемблера есть множество аппаратных команд, которые имеют прямые эквиваленты в языке *Spin*. Это позволяет легче справиться как с изучением обоих языков, так и с использованием ИМС Propeller вообще.

Язык *Spin* компилируется программным обеспечением *Propeller Tool* в пакеты данных (tokens), которые, по ходу исполнения, обрабатываются встроенным в ИМС Propeller Интерпретатором *Spin*. Те, кто хорошо владеет другими языками программирования, отмечают, что язык *Spin* легок для изучения и хорошо приспособлен для многих задач. При помощи *Spin* Вы можете не только с легкостью решать относительно низкоскоростные, высокоуровневые задачи, но и написать код для более скоростных, таких как асинхронная последовательная связь до 19200 бод.

Язык Propeller Ассемблер транслируется программой *Propeller Tool* в машинный код и выполняется в чистом виде по ходу работы. Программисты на Ассемблере получают удовольствие от построения языка и возможности решать высокоскоростные задачи с применением очень малого количества кода.

Объекты Propeller (см. ниже) могут быть написаны как полностью на *Spin*, так и с использованием различных комбинаций языков *Spin* и Propeller Ассемблер. Объекты могут быть написаны и полностью на ассемблере, но, по крайней мере, две строки *Spin*-кода необходимы для запуска конечного приложения.

Объекты Propeller

Язык *Spin* является объектно-ориентированным и служит базой для любого Propeller Приложения.

Что такое Объект?

Объекты – это те же программы, только написанные таким образом, чтобы: 1) создать инкапсулированную структуру, 2) выполнять отдельную задачу, и 3) иметь возможность использования другими приложениями.

Например, объекты «Keyboard» (Клавиатура) и «Mouse» (Мышь) поставляются в составе программного обеспечения *Propeller Tool*. Объект «Keyboard» – это программа, которая обеспечивает интерфейс ИМС Propeller со стандартной PC-совместимой клавиатурой. Аналогично, объект «Mouse» поддерживает интерфейс со стандартной

компьютерной мышью. Оба этих объекта являются самодостаточными программами с тщательно проработанными программными интерфейсами, позволяющими другим объектам и приложениям с легкостью их использовать.

Используя готовые объекты, можно очень быстро создавать сложные приложения. К примеру, если приложение включает в себя объекты «Keyboard» и «Mouse», то с помощью еще всего нескольких строк кода можно реализовать стандартный интерфейс пользователя. Поскольку объекты инкапсулированы и обеспечивают четкий программный интерфейс, разработчики приложений не обязательно должны точно знать, как объект реализует свою задачу, чтобы его использовать. Так же, как водитель машины, не обязательно зная, как устроен двигатель, но, понимая интерфейс (педали сцепления, газа и тормоза и т.д.), может заставить автомобиль развивать либо замедлять скорость движения.

Хорошо написанные объекты, созданные одними разработчиками, могут затем с легкостью использоваться в своих приложениях многими другими.

Объекты и приложения

Объекты в ИМС Propeller состоят из кода *Spin* и, опционально, кода *Propeller*-ассемблера, см. Рис. 3-1. Далее мы будем называть их просто “объектами”.

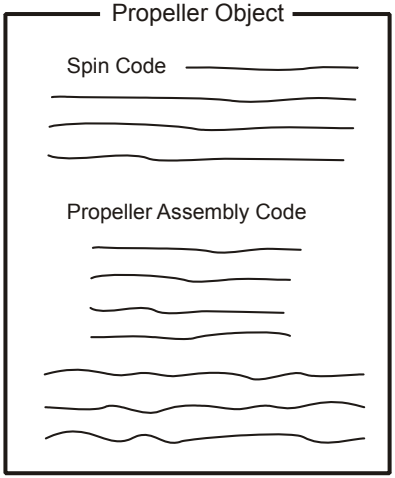


Рис. 3-1: объект Propeller

Объекты хранятся на Вашем компьютере как файлы с расширением “*.spin*”, поэтому каждый *Spin*-файл вы должны рассматривать как объект.

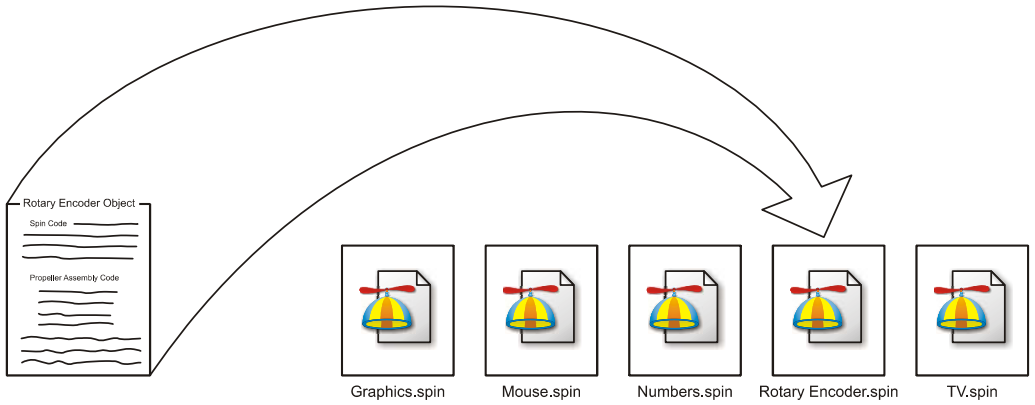


Рис. 3-2: Файлы Объектов состоят из кода Spin, и, возможно, Propeller Ассемблера, и хранятся как файлы ".spin" на жестком диске Вашего компьютера.

Каждый объект может рассматриваться как "кирпичик" приложения. Объект может использовать один или более других объектов для построения более сложных приложений. Обычно это называется "ссылка" или "подключение" другого объекта. Когда объект ссылается на другой объект, он формирует иерархию, где он является верхним объектом, как на Рис. 3-3. Самый верхний объект называется "Верхний Объектный Файл" и является стартовой точкой при компиляции Propeller приложений.

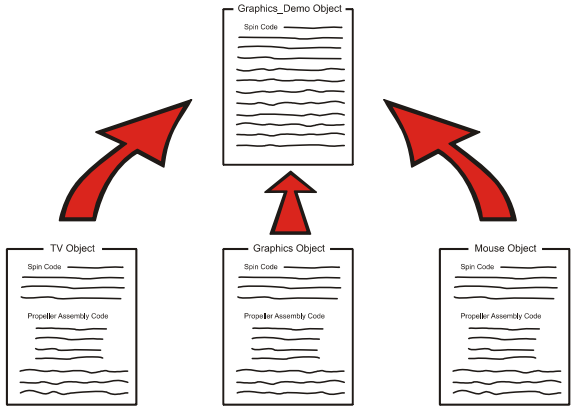


Рис. 3-3: Иерархия Объекта

После компиляции, объект Graphics Demo является "Верхним Объектным Файлом", ссылающимся на другие три объекта, показанные снизу.

На рисунке выше, объект «Graphics Demo» ссылается на другие три объекта: «TV», «Graphics», и «Mouse». При компиляции объекта «Graphics Demo», он рассматривается как Верхний Объектный Файл; при этом остальные три объекта загружаются и компилируются с ним, образуя в результате завершенную программу, называемую Propeller Приложением или, для краткости, просто “приложением”.

Приложения формируются из одного или более объектов. В действительности приложение – это специальным образом откомпилированный двоичный поток, состоящий из исполнимого кода и данных, и выполняемый в ИМС Propeller.

После загрузки, приложение сохраняется в Основной ОЗУ ИМС Propeller и, при необходимости, во внешней ЭСППЗУ. Во время исполнения, приложение выполняется одним или более *Cog* по указанию самого приложения.

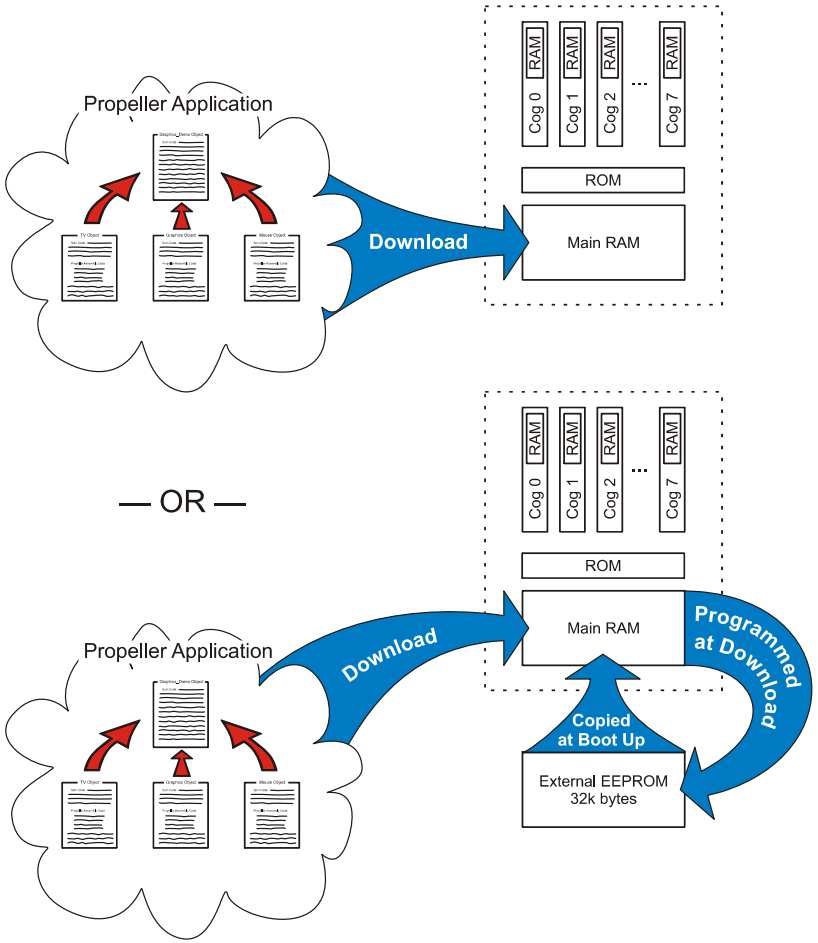


Рис. 3-4: Загрузка

Приложения, состоящие из одного или более объектов, загружаются в Основное ОЗУ ИМС Propeller и, при необходимости, во внешнюю ЭСППЗУ.

Подключение для Загрузки

Для загрузки приложения с компьютера в ИМС Propeller, ее необходимо правильно подключить.

- Если у Вас есть демонстрационная плата Propeller Demo Board (Rev C или D), она включает все необходимые компоненты, в том числе и ИМС Propeller. Подключите ее к источнику питания и USB-порту, и включите питание. Возможно, Вам также придется установить USB-драйвера, как указано в документации на Propeller Demo Board.
- Если у Вас нет Propeller Demo Board, мы будем считать, что Вы имеете ИМС Propeller и что Вы умеете собирать электрические схемы. Для получения информации о типах корпусов обратитесь к стр. 14 (где указано назначение выводов ИМС Propeller) и секции «Внешние соединения» на стр. 17, где приведен пример схемы для подключения питания и программирования. Если Вы используете устройство Propeller Plug, Вам, возможно, также будет необходимо установить драйвера USB, как указано в его документации. В оставшейся части этой главы работа будет вестись со схемой, идентичной Propeller Demo Board. Кроме упомянутых источника питания и цепей программирования, добавьте в свою макетную плату компоненты и цепи из приведенной ниже схемы. Вы также можете обратиться к схеме самой Propeller Demo Board, которую можно загрузить с вебсайта Parallax.

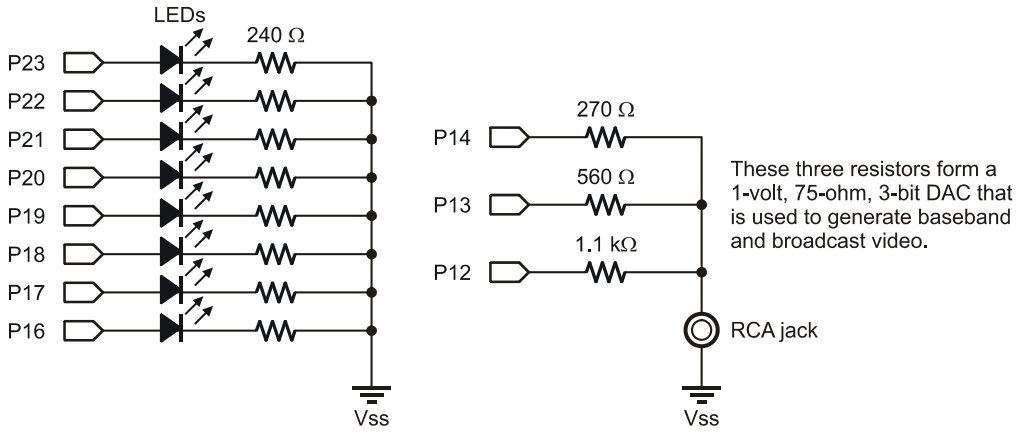


Рис. 3-5: Схема для макетирования

Если Вы правильно выполнили подключения, указанные выше, то Вы уже готовы к проверке и идентификации ИМС Propeller программой *Propeller Tool*. Запустите программу *Propeller Tool* (Версии 1.x.x) и затем нажмите клавишу *F7* (либо выберите

Run → Identify Hardware… из меню). Если ИМС Propeller правильно подключена к источнику питания и PC, Вы увидите диалог “Information”, такой как на Рис. 3-6.

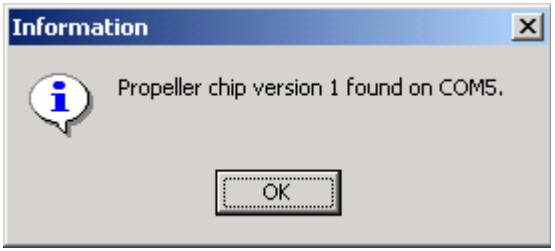


Рис. 3-6: Информационный Диалог

Порт (COM5) на Вашем компьютере может быть другим.

Кратко: Введение

- ИМС Propeller программируется с использованием двух специально разработанных языков: *Spin* и Propeller ассемблер.
  - *Spin* – это объектно-ориентированный язык верхнего уровня, интерпретируемый при исполнении приложения.
  - Propeller ассемблер – это оптимизированный язык нижнего уровня, который выполняется в чистом виде при исполнении приложения.
- Объекты – это программы, которые:
  - Инкапсулированы (самодостаточны).
  - Выполняют отдельную задачу.
  - Могут быть использованы многими приложениями.
- Хорошо написанные объекты одного разработчика могут быть с легкостью использованы многими другими приложениями различных разработчиков.
- Объект в ИМС Propeller:
  - Состоит из двух или более строк *Spin* и, возможно, кода ассемблера.
  - Хранится в компьютере в виде файлов с расширением ".spin".
  - Может использовать один или более других объектов для создания сложного приложения.
- Приложения Propeller:
  - Состоят из одного или более объектов.
  - Представляют откомпилированные двоичные потоки, включающие исполнимый код и данные.
  - Запускаются в ИМС Propeller в одном или более процессоров (*Cogs*), как указано в самом приложении.
- Самый верхний объект в откомпилированном приложении называется "Верхний Объектный Файл" (Top Object File).

Упражнение 1: Output.spin – Наш первый объект

Далее приведен простой объект, написанный на *Spin*, который периодически переключает линию В/В из одного состояния в другое. Запустите программу *Propeller Tool* и введите эту программу в редакторе. Через мгновение мы покажем, как она работает. Убедитесь, что строка "`PUB`" начинается со столбца 1 (самый левый край панели редактора) и тщательно проверьте отступы в каждой строке; это очень важно для правильной работы.

```
PUB Toggle
  dira[16]~~
  repeat
    !outa[16]
    waitcnt(3_000_000 + cnt)
```

Отступы в коде очень важны, в то же время регистр – нет. Код в языках ИМС Propeller не чувствителен к регистру. Однако, в рамках этой книги, зарезервированные слова указываются жирным шрифтом всеми заглавными символами, за исключением отрывков и выдержек из кода, ─ чтобы помочь Вам освоиться с ними.

После проверки верности напечатанного кода, нажмите клавишу *F10* (либо выберите Run → Compile Current → Load RAM + Run из меню) для компиляции и загрузки программы нашего примера. Если введенная программа синтаксически верна, ИМС Propeller правильно подключена к источнику питания и PC, Вы увидите диалог "Propeller Communication", который мгновенно появится на экране (такой, как на Рис. 3-7), после чего светодиод на линии B/B 16 ИМС Propeller начнет мигать около двух раз в секунду. Выполненные нами действия отображены в верхней части Рис. 3-4 на стр. 106.

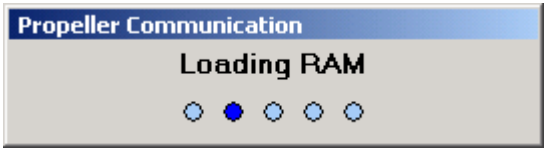


Рис. 3-7: Диалог связи с ИМС Propeller

На самом деле, произошедшие действия выполнились слишком быстро, чтобы их заметить, так как программа в примере очень маленькая. Когда Вы нажали F10, введенный исходный код был откомпилирован и преобразован в приложение программой *Propeller Tool*. Затем была обнаружена ИМС Propeller, подключенная к PC, и приложение было загружено в ее ОЗУ. В завершение, ИМС Propeller запустила приложение из ОЗУ, мигая светодиодом на линии B/B 16.

Разница между загрузкой в ОЗУ и ЭСППЗУ

Перед тем, как мы опишем работу кода, давайте глубже присмотримся к процессу загрузки. Так как наш код был загружен только в ОЗУ, выключение питания либо сброс чипа приведет к потере содержимого и остановке программы. Нажмите кнопку сброса. Светодиод погаснет и никогда больше не засветится.

Что делать, если мы не хотим, чтобы выполнение программы прерывалось необратимо? Для этого мы должны загружать код не только в ОЗУ но и в ЭСППЗУ. Давайте загрузим код еще раз, но на этот раз нажмем клавишу *F11* (или выберем Run → Compile Current → Load EEPROM + Run из меню) для компиляции и загрузки программы нашего примера в ЭСППЗУ. Этот процесс показан в нижней части Рис. 3-4, на стр. 106. Как можно понять из рисунка, программа, на самом деле, сперва загружается в ОЗУ, затем ИМС Propeller программирует свою внешнюю ЭСППЗУ, после чего запускает приложение из ОЗУ, мигая светодиодом на линии В/В 16.

Вы, скорее всего, заметили, что диалог “Propeller Communication” находился на экране намного дольше - время программирования у ЭСППЗУ намного больше, чем у ОЗУ.

Теперь снова нажмите кнопку сброса. Когда вы отпустите кнопку, вы заметите задержку около 1.5 секунд, после чего светодиод опять начнет мигать. Это подтверждает, что мы добились того, чего хотели – приложение сохранено в нашей ИМС Propeller.

При пробуждении из состояния сброса, ИМС Propeller выполнила процедуру начальной загрузки, описанную на стр. 20. Во время выполнения этой процедуры, микросхема определила, что ей необходимо загружаться из внешней ЭСППЗУ, после чего ей потребовалось примерно 1.5 секунды для копирования всех 32 кБайт её содержимого в ОЗУ и запуска их на выполнение.

Загрузка приложения только в ОЗУ удобна при отладке, так как она намного быстрее. Загрузка и в ОЗУ, и в ЭСППЗУ для сохранения приложения выполняется лишь при необходимости, так как требует дополнительного времени.

Один момент: если Вы загружали приложение в ЭСППЗУ один или несколько раз, после чего изменили и загрузили его в ОЗУ, то при нажатии кнопки сброса ИМС Propeller загрузит Ваше старое приложение. Сейчас это может показаться не страшным, но такие действия могут привести к неприятным эффектам при отладке, если Вы не обратите на это внимание. Если что-то работает не так после сброса – в первую очередь убедитесь, что в ЭСППЗУ находится последняя версия приложения.

Упражнение 1: Описание объекта Output.spin

А теперь опишем наш пример:

```
PUB Toggle
  dira[16]~~
  repeat
    !outa[16]
    waitcnt(3_000_000 + cnt)
```

Первая строка, `PUB Toggle`, объявляет, что мы создаем метод "*Public*", называемый "`Toggle`". Метод – это термин объектно-ориентированного программирования, аналогичный понятию "процедура" или "подпрограмма". Мы выбрали имя `Toggle` просто потому, что оно описывает действия метода, и мы знаем, что оно уникально. Оно должно быть уникальным и должно соответствовать Правила Идентификаторов, приведенным на стр. 183. Более детально термин **PUB** ("*Public*") будет описан далее, сейчас просто отметим, что каждый объект должен включать как минимум один *Public* (**PUB**) метод.

Остальной код – это логическая часть метода `Toggle`. Мы отступили в каждой строке два пробела от колонки **PUB,** чтобы получить наглядность; эти отступы не обязательны, но считаются правильным тоном оформления для прозрачности.

Первая строка метода `Toggle` (вторая строка примера), `dira[16]~~`, устанавливает направление линии В/В 16 на вывод. Идентификатор **DIRA** – это имя регистра направления для линий В/В от P0 до P31; сброс либо установка битов в этом регистре изменяет направление соответствующей линии В/В на ввод либо вывод. Символы `[16]` следующие за `dira` показывают, что мы хотим изменить только бит 16 регистра направления, – тот, который отвечает за линию В/В 16. В конце, `~~` – это оператор пост-установки, который устанавливает бит 16 регистра направления в высокий уровень (1), что приводит к переключению направления линии В/В 16 на вывод. Оператор пост-установки позволяет кратко записать операцию, подобную `dira[16] := 1` , которая знакома Вам из других языков программирования.

Следующая строка, `repeat`, создает цикл, состоящий из двух строк ниже. Этот цикл **REPEAT** выполняется бесконечно долго, переключая P16, ожидая ¼ секунды, опять переключая P16, ожидая ¼ секунды, и так далее.

Следующая строка, `!outa[16]`, переключает состояние линии В/В 16 между высоким (VDD) и низким (VSS). Символы **OUTA** – это регистр состояния выходов для линий В/В с P0 по P31. Символы `[16]` в `!outa[16]` показывают, что мы хотим изменить состояние только бита 16 выходного регистра – того, который соответствует линии В/В 16.

Символ ! в начале выражения является побитным оператором инверсии НЕ; он изменяет состояние указанных битов на противоположное (в этом случае это относится к биту 16).

Последняя строка, `waitcnt(3_000_000 + cnt)`, организует задержку из 3 миллионов циклов частоты ядра. Буквально **WAITCNT** значит "Wait for System Counter" – Ожидать Системный Счетчик. Символика `cnt` – это регистр Системного Счетчика; **CNT** возвращает текущее значение Системного Счетчика, поэтому эта строка обозначает "ожидать значения Системного Счетчика, равного 3 миллиона плюс его текущее значение". В этом примере кода мы не задавали значения рабочей частоты для ИМС Propeller, поэтому по умолчанию она работает на внутренней высокой частоте (около 12 МГц), приводящей к задержке для 3 миллионов циклов в приблизительно ¼ секунды.

Помните, как мы говорили о необходимости уделять большое внимание форматированию каждой строки? Вот как раз то место, где отступы обязательны: язык *Spin* использует уровни отступов в строках, следующих за условными операторами либо операторами циклов (**IF**, **CASE**, **REPEAT**, и т.д.) для определения, какие именно линии принадлежат к данной структуре. В этом случае, поскольку две строки, следующие за оператором `repeat` , имеют одинаковый отступ вправо как минимум на один пробел за столбец оператора `repeat`, эти две линии считаются частью цикла `repeat`. Если у Вас возникают трудности при распознавании структурных групп, программа *Propeller Tool* может отобразить их более наглядно, явно видимыми, при помощи функции Индикаторов Блок-Групп. Используйте *Ctrl+I* для переключения этой функции. Рис. 3-8 отображает код нашего примера с включенной функцией индикаторов.

```
PUB Toggle
  dira[16]~~
  repeat
    !outa[16]
    waitcnt(3_000_000 + cnt)
```

Рис. 3-8: Индикаторы Блок-Групп

Ctrl+I включает и выключает функцию

Если Вы еще не сохранили этот пример объекта, Вы можете это сделать, нажав *Ctrl+S* (или выбрав File → Save из меню). Вы можете сохранить его в выбранную Вами папку, но убедитесь, что Вы сохраняете его под именем "Output.spin", поскольку некоторые дальнейшие упражнения будут на него ссылаться.

Кратко: Упр. 1

- Приложения загружаются либо только в ОЗУ, либо и в ОЗУ, и в ЭСППЗУ ИМС Propeller.
    - То, что в ОЗУ, – не сохраняется при выключении питания либо сбросе.
    - То, что в ЭСППЗУ, – загружается в ОЗУ при старте примерно за 1½ сек.
    - Для загрузки текущего объекта:
        - Только в ОЗУ: нажать *F10* или выбрать Run → Compile Current → Load RAM + Run.
        - В ОЗУ + ЭСППЗУ: *F11* или выбрать Run → Compile Current → Load EEPROM + Run.
- Язык *Spin*:
    - Метод обозначает “процедура” или “подпрограмма”
    - `PUB` *Symbol* объявляет *Public* метод, называемый *Symbol*. Каждый объект должен содержать как минимум один метод *Public* (`PUB`). См. `PUB` на стр. 334 и Правила Идентификаторов на стр. 183.
    - `DIRA` – регистр направления для линий В/В 0-31; каждый его бит устанавливает направление соответствующей линии В/В на ввод (0) либо вывод (1). См. `DIRA, DIRB` на стр. 249.
    - `OUTA` – регистр состояния выходов для линий В/В 0-31; каждый его бит устанавливает состояние выхода соответствующей линии В/В в низкий (0) либо высокий (1) уровень. См. `OUTA, OUTB` на стр. 326.
    - В регистре могут использоваться индексы, например [16], для доступа к конкретному его биту. См. `DIRA, DIRB` на стр. 249 или `OUTA, OUTB` на стр.326.
    - Оператор **~~**, следующий за регистром или переменной, устанавливает его бит(ы) в единицу. См. Распространение Знака 15 или Пост-Установка ‘**~~**’ на стр. 306 в секции Операторы Spin.
    - Оператор **!** перед регистром или переменной, инвертирует его бит(ы). См. «Побитовое НЕ (NOT) ‘**!**’» на стр. 317 в секции Операторы Spin.
    - `REPEAT` создает цикл. См. `REPEAT` на стр. 340.
    - `WAITCNT` формирует задержку. См. `WAITCNT` на стр. 373.
    - Отступы в начале строк:
        - Показывают принадлежность к предыдущей структуре; обязательны для строк, следующих за условными либо цикловыми операторами (как `REPEAT`). (Отступы не обязательны после индикаторов блока, таких как `PUB`.)
        - *Ctrl+I* переключает видимость индикаторов блок-групп.

Процессоры (*Cogs*)

ИМС Propeller имеет восемь одинаковых процессоров (*Cogs*). Каждый *Cog* может быть индивидуально запущен либо остановлен в любой момент времени по указанию выполняемого приложения. Каждый процессор может быть запрограммирован для выполнения как независимых, так и совместных с другими задач, что определяется приложением и может изменяться во время его выполнения.

В примере Output.spin мы не указывали, какой(-кие) *Cog* должен был выполнять наше приложение, но тогда как же оно работало? Для ознакомления, Вам следует прочесть «Глава 1. Начальная загрузка», стр. 22, и «Исполнение», стр. 22, а здесь мы обсудим это немного глубже.

В нашем примере, после включении питания, ИМС Propeller запускает первый процессор (*Cog* 0) и загружает в него программу загрузчика. Программа загрузчика копируется из Основного ПЗУ ИМС Propeller во внутреннее ОЗУ *Cog* 0. Затем *Cog* 0 выполняет программу загрузчика в своей внутренней памяти, которая вскоре определяет, что ей необходимо скопировать код пользователя из внешней ЭСППЗУ. Далее *Cog* 0 копирует все 32 кБайт содержимого ЭСППЗУ в Основное 32 кБайт ОЗУ ИМС Propeller (отдельно от внутреннего ОЗУ *Cog*). Затем программа загрузчика заставляет *Cog* 0 перегрузить себя внутренним Интерпретатором *Spin*; программа загрузчика в *Cog* 0 останавливается в этой точке, поскольку она перезаписывается программой Интерпретатора.

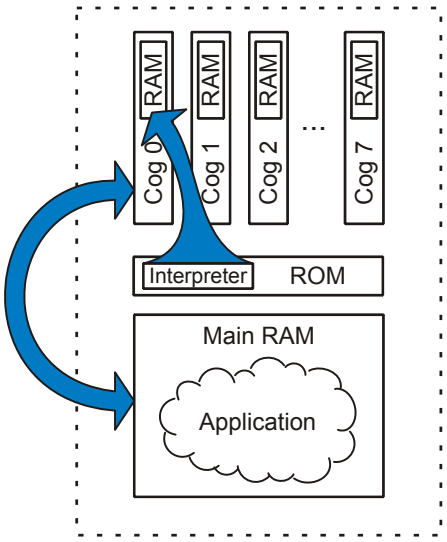


Рис. 3-9: Выполнение Output.spin

Отметьте, что именно Интерпретатор Spin, а не приложение Spin, загружено в ОЗУ Cog. Приложение Spin расположено в Основной ОЗУ и интерпретируется программой Интерпретатора Spin, выполняемой в Cog.

Итак, сейчас *Cog* 0 выполняет Интерпретатор *Spin*, который выбирает и исполняет код нашего приложения из основного ОЗУ. Это показано на Рис. 3-9. Поскольку наше приложение полностью состоит из интерпретируемого кода *Spin*, оно остается только в Основной Памяти, в то время как процессор, выполняющий Интерпретатор *Spin* (*Cog* 0 в этом случае) читает, интерпретирует и исполняет его код. Больше во время загрузки или выполнения приложения не было запущено ни оного из *Cog*; остальные семь процессоров остаются в состоянии бездействия, практически совсем не потребляя ток. Позже мы изменим наше приложение так, чтобы запустить и остальные процессоры.

Упражнение 2: Output.spin - Константы

Давайте немного доработаем нашу программу. Предположим, мы хотим облегчить процесс назначения линии В/В и длительности используемой задержки. Так, как это написано сейчас, нам бы пришлось найти и изменить в двух местах номер линии, и еще величину задержки в третьем месте. Мы можем улучшить код, определив эти параметры в отдельном месте, которое легко найти и отредактировать. Посмотрите на следующий пример и отредактируйте Ваш код в соответствии с ним (все новые либо измененные элементы в нем мы подсветили).

```
CON
  Pin   = 16
  Delay = 3_000_000

PUB Toggle
  dira[Pin]~~
  repeat
    !outa[Pin]
    waitcnt(Delay + cnt)
```

Новый блок `CON` в верхней части кода определяет глобальные константы для объекта (см. `CON`, стр. 228.) В нем мы создали два идентификатора, `Pin` и `Delay`, и сопоставили им значения констант, соответственно 16 и 3000000. Сейчас мы можем использовать имена `Pin` и `Delay` в любом месте кода для представления наших значений констант 16 и 3000000. Вы, наверное, обратили внимание, что мы использовали символы подчеркивания (_) для разделения групп тысяч в числе 3000000? Символы подчеркивания разрешены в любом месте внутри констант, а запятые или пробелы – нет; такой формат позволяет представить большие числа более удобно читаемыми.

В методе Toggle мы заменили оба значения 16 идентификатором Pin, и заменили 3_000_000 идентификатором Delay. При компилировании, *Propeller Tool* будет использовать значения констант в местах с соответствующими идентификаторами. В дальнейшем это позволит более легко изменять номер линии или значение задержки, поскольку нам будет достаточно изменить их значения в верхней части кода, где их легко найти и отредактировать.

Попробуйте изменить константу Delay с 3000000 на 500000 и загрузите опять; светодиод теперь мигает с частотой 12 раз в секунду (24 переключения в секунду). Вы также можете изменить константу Pin с 16 на 17 и загрузить вновь, чтобы увидеть мигание другого светодиода.

ПРИМЕЧАНИЕ: Вы также можете использовать линии с 18 по 23, однако на плате Propeller Demo Board они соединены в пары с резисторами для цепи драйвера VGA, поэтому мигать будут по два светодиода сразу.

Указатели Блоков

Вы, возможно, заметили, что задний фон блоков кода CON и PUB окрашивался в различные цвета, когда Вы вводили его в редакторе. Такой метод используется программой *Propeller Tool* для отображения различных блоков кода.

Код *Spin* организован в блоках, которые имеют различные назначения. CON и PUB являются указателями группы элементов, которые указывают соответственно начало “блока констант” и “блока метода *Public*”. Каждый указатель блока элементов должен начинаться в первом столбце строки текста (крайняя левая позиция области окна редактирования). В языке *Spin* есть шесть типов блоков: CON, VAR, OBJ, PUB, PRI, и DAT. Далее приводится список указателей блоков и их назначение:

CON Блок Глобальных Констант. Определяет идентификаторы констант, которые могут быть использованы в любой части объекта (а иногда и вне его), где разрешено использование численных значений.

VAR Блок Глобальных Переменных. Определяет идентификаторы переменных, которые могут быть использованы в любой части объекта, где разрешено использование переменных.

OBJ Блок Ссылок на Объекты. Определяет идентификаторы ссылок на другие существующие объекты. Они используются для доступа к другим объектам и принадлежащим этим объектам методам и константам

PUB Блок *Public*-метода. *Public*-методы – это подпрограммы, которые доступны как внутри, так и вне объекта. Подпрограммы *Public* обеспечивают связь с объектом; это канал, по которому внешние методы взаимодействуют с объектом. В каждом объекте должно быть хотя бы одно **PUB**-объявление.

PRI Блок *Private*-метода. *Private*-методы – это подпрограммы, которые доступны только внутри объекта. Поскольку они не видимы снаружи, они обеспечивают уровень инкапсуляции для защиты важных элементов в рамках объекта и помогают поддерживать целостность объекта.

DAT Блок Данных. Определяет таблицы данных, буферы памяти и код на ассемблере Propeller. Данные этого блока могут быть сопоставлены символическим именам и могут быть доступны в коде *Spin* и ассемблере.

В объекте могут встречаться несколько блоков каждого типа, распределенных в любом необходимом порядке, но обязательно должен быть как минимум один блок **PUB**. Хотя количество блоков и их порядок довольно гибко, обычно блоки **CON**, **VAR**, **OBJ** и **DAT** присутствуют только один раз, а блоки **PUB** и **PRI** - много раз, и предлагаемый порядок их расположения для типичных программ – это порядок, приведенный в списке выше.

Самый первый блок **PUB** в самом первом объекте (Верхнем Объектном Файле, откуда начинается компиляция) автоматически становится стартовой точкой приложения; при старте он выполняется в первую очередь. Больше ни один из методов *Public* или *Private* автоматически не выполняется, они выполняются согласно естественному порядку, определяемому приложением.

Для облегчения определения типа, а также начала и конца каждого из блоков, программа *Propeller Tool* автоматически окрашивает задний фон каждого блока различным цветом, даже если два последовательно идущих блока — одного типа. Это никак не влияет на сам исходный код, это просто индикатор для облегчения работы с экраном, который предназначен для решения типичной проблемы с исходным текстом, а именно: когда текста становится больше, тяжелее быстро найти необходимый метод, перемещаясь вверх и вниз и не имея какого-нибудь разделителя между методами. Выделение программных блоков цветом фона служит автоматическим разделителем, сохраняющим Ваше время на создание текстовых разделителей вручную.

Упражнение 3: Output.spin - Комментарии

Наш объект Output стал лучше, но он все еще может быть сделан более читабельным. Как насчет добавления некоторых комментариев к коду для облегчения понимания его другими пользователями? Следующий пример работает так же, как и ранее, но имеет

несколько комментариев (предназначенных для чтения людьми, комментарии – это не выполнимый код) сверху и справа нашего существующего кода.

Эти комментарии должны помочь людям понять, что делает код. Существует четыре типа комментариев, поддерживаемых программой *Propeller Tool* (все они показаны в этом примере):

'... – Комментарий одиночной строки кода (апостроф).

''... – Комментарий одиночной строки документации (два апострофа, НЕ символ кавычек).

{...} – Комментарий нескольких строк кода (фигурные скобки).

{{...}} – Комментарий нескольких строк документации (двойные фигурные скобки)

```
{{Output.spin

Toggles Pin with Delay clock cycles of high/low time.}}

CON
  Pin   = 16                  { I/O pin to toggle on/off }
  Delay = 3_000_000           { On/Off Delay, in clock cycles}

PUB Toggle
''Toggle Pin forever
{Toggles I/O pin given by Pin and waits Delay system clock cycles
in between each toggle.}

  dira[Pin]~~                 'Set I/O pin to output direction
  repeat                      'Repeat following endlessly
    !outa[Pin]                '  Toggle I/O Pin
    waitcnt(Delay + cnt)      '  Wait for Delay cycles
```

Комментарии одиночной строки начинаются с как минимум одного апострофа (') и продолжаются до конца строки. Выполнимый код может находиться слева от комментария, но не справа него, поскольку в таком случае код будет "закомментирован". Комментарии "`'Set I/O pin...`" и "`'Repeat following...`" – это примеры комментариев одиночных строк.

Комментарии нескольких строк начинаются с как минимум одной открывающей фигурной скобки ({). В отличие от комментариев одиночной строки, выполнимый код может находиться как слева, так и справа от комментариев нескольких строк. Комментарии нескольких строк могут вообще целиком находиться на одной строке, либо занимать несколько строк. Комментарии {On/Off Delay...} и {Toggles I/O pin given...} – это примеры таких комментариев.

Если комментарий начинается лишь с одного апострофа (') либо одной открывающей фигурной скобки ({), то это – "комментарий кода", предназначенный для чтения разработчиками при просмотре самого исходного кода.

Если же комментарий начинается либо с двух апострофов (''), либо с двух открывающих фигурных скобок ({{), без пробелов между ними, то это – "комментарий документации", специальный тип комментария, который не только виден внутри кода, но и который может быть извлечен программой *Propeller Tool* в форматированный документ, без исполнимого кода, для обеспечения легкого восприятия.

Как обсуждалось ранее (Глава 2, Режимы просмотра, стр. 74), редактор программы *Propeller Tool* имеет режим просмотра документации. Если приведенный выше код ввести в ее редакторе со включенным режимом просмотра документации, процесс компиляции будет сопровождаться выводом комментариев документации и некоторой статистики об откомпилированном коде. Выглядит это так:

```
Output.spin

Toggles Pin with Delay clock cycles of high/low time.
Object "Output" Interface:

PUB Toggle

Program:    8 Longs
Variable:   0 Longs

------------
PUB Toggle

Toggle Pin forever
```

Сравнив это с нашим кодом, Вы должны узнать текст, взятый прямо из наших комментариев. Секция " Object "Output" Interface: " создается программой *Propeller Tool* автоматически; в ней перечисляются все *Public*-методы (в нашем случае PUB

Toggle), показывается размер программы, 8 longs (32 байта) и сообщается о том, что в программе не используется ни одной переменной. Далее опять перечисляются все *Public*-методы, с подчеркиванием каждого метода и комментариями документации, которые ему принадлежат. Эта секция отображает *Public*-метод Toggle и наш последний комментарий документации, " Toggle Pin forever", который объясняет, что делает метод Toggle.

Добавление комментариев документации в Ваш код позволяет Вам создать всего один файл, содержащий как исходный код, так и документацию на объект. Это чрезвычайно удобно для разработчиков, поскольку они могут легко переключиться в режим просмотра Документация, чтобы изучить незнакомый им объект. Для еще большего удобства, шрифт Parallax программы *Propeller Tool* имеет множество специальных символов, обеспечивающих включение изображений схем, временных диаграмм и математических символов прямо в объекты, как на Рис. 2-1 на стр. 44.

Кратко: Упр. 2 и 3

- ИМС Propeller имеет восемь идентичных процессоров, называемых *Cog*.
  - Любое количество *Cog*-ов может быть запущено либо остановлено в любой момент времени, как укажет приложение.
  - Каждый *Cog* может выполнять независимые либо совместные задачи.
  - При начальной загрузке, *Cog0* выполняет Интерпретатор *Spin* для выполнения *Spin*-приложения, расположенного в Основной Памяти.
- Язык *Spin*:
  - Организован в блоках, имеющих различное назначение.
    - CON – Определяет глобальные константы, см. стр. 228.
    - VAR – Определяет глобальные переменные, см. стр. 228.
    - OBJ – Определяет ссылки на объекты, см. стр. 228.
    - PUB – Определяет *Public*-метод, см. стр. 334.
    - PRI – Определяет *Private*-метод, см. стр.333.
    - DAT – Определяет данные, буферы и код ассемблера, стр. 243.
  - Указатели Блоков должны начинаться в строке с первой колонки.
  - Блок любого типа может встречаться несколько раз в любом порядке.
  - Самый первый блок PUB в самом первом объекте – это стартовая точка приложения Propeller.
  - Подчеркивания "_" в константах разделяют логические группы, подобные тысячам в десятичных числах.
  - Типы комментариев:

- Комментарии кода; видимы только в исходном коде. Удобны для заметок разработчикам и описывают работу кода.
  - '... – Однострочные; начинаются с апострофа и продолжаются до конца строки.
  - {...} – Многострочные; начинаются и заканчиваются одиночными фигурными скобками.
- Комментарии Документации; видимы в режимах просмотра Исходного кода и Документации. Удобны для документирования объекта. Могут даже включать схемы, временные диаграммы и другие специальные символы.
  - ''... – Однострочные; начинаются двойным апострофом и продолжаются до конца строки.
  - {{...}} – Многострочные; начинаются и заканчиваются двойными фигурными скобками.

Упражнение 4: Output.spin – Параметры, Вызовы и Конечные Циклы

Наш текущий объект из Упражнения 3 интересен, но все же еще не очень гибок; в нем метод `Toggle` работает с жестко заданной линией вывода и задержкой. Давайте сделаем метод `Toggle` более гибким, а так же придадим ему свойство переключать линию заданное, фиксированное количество раз. Посмотрите на следующий пример и отредактируйте свой код в соответствии с ним. Мы зачеркнули элементы, которые должны быть удалены, и выделили все новые элементы.

```
{{Output.spin

~~Toggles Pin with Delay clock cycles of high/low time.}}~~
Toggles two pins, one after another.}}

~~CON~~
~~  Pin   = 16                  { I/O pin to toggle on/off }~~
~~  Delay = 3_000_000           { On/Off Delay, in clock cycles}~~

PUB Main
  Toggle(16, 3_000_000, 10)  'Toggle P16 ten times, 1/4 s each
  Toggle(17, 2_000_000, 20)  'Toggle P17 twenty times, 1/6 s each
```

```
PUB Toggle(Pin, Delay, Count)
''Toggle Pin forever
{Toggles I/O pin given by Pin and waits Delay system clock cycles
in between each toggle.}
{{Toggle Pin, Count times with Delay clock cycles in between.}}

  dira[Pin]~~                   'Set I/O pin to output direction
  repeat Count                  'Repeat for Count iterations
    !outa[Pin]                  '  Toggle I/O Pin
    waitcnt(Delay + cnt)        '  Wait for Delay cycles
```

Откомпилируйте и загрузите это приложение, чтобы увидеть результаты. Светодиод на линии 16 должен мигнуть 5 раз (10 переключений) с периодом ¼ сек, затем он остановится, а светодиод на линии 17 будет мигать 10 раз (20 переключений) с периодом 1/6 сек.

В этом примере мы убрали блок констант (**CON**), добавив новый метод, названный `Main`, и сделали несколько важных изменений в методе `Toggle`. Метод `Toggle` все так же выполняет переключение линии, а метод `Main` говорит ему, когда и как это делать.

Метод Toggle

Давайте сначала рассмотрим более подробно метод `Toggle`. В его объявлении, справа от имени, мы добавили: `(Pin, Delay, Count)`. Тем самым мы создали "список параметров" для нашего метода `Toggle`, состоящий из трех идентификаторов: `Pin`, `Delay` и `Count`. Список параметров – это один или несколько идентификаторов, которые должны быть заменены величинами при вызове метода; подробнее об этом чуть позже. Каждый параметр – это переменная размера *long* (4-байта), которая является локальной в рамках метода; все они доступны из метода, но не за его пределами. Переменные параметров могут быть изменены внутри метода, но эти изменения не влияют ни на что за его пределами.

Итак, сейчас наш метод `Toggle` может быть вызван другими методами и в него могут передать уникальные значения в качестве `Pin`, `Delay` и `Count`; он более гибок, поскольку мы можем настроить его параметры функционирования.

Внутри `Toggle` не изменилось ничего, кроме команды **REPEAT**, которая теперь выглядит как `repeat Count`. Помните, в наших предыдущих примерах, цикл **REPEAT** был бесконечным циклом? Он никогда не завершался. Теперь, если Вы введете выражение сразу за **REPEAT**, цикл станет конечным, который повторяется количество раз, указанное

в выражении. В этом случае наш цикл REPEAT будет выполняться Count раз, после чего он завершится, и следующие снизу за концом цикла строки кода начнут выполняться.

Метод Main

А сейчас посмотрите на метод Main. Его первая строка, Toggle(16, 3_000_000, 10) – это вызов метода; она приводит к выполнению метода Toggle с использованием значения 16 для параметра Pin, значения 3 миллиона для параметра Delay, и 10 – для Count. Следующая строка выглядит похоже, Toggle(17, 2_000_000, 20), но она вызывает метод Toggle с другими параметрами: 17 для Pin, 2 миллиона для Delay, и 20 – для Count.

Заметили, что мы поместили метод Main над Toggle? Помните, что первый *Public*-метод в первом объекте выполняется автоматически, когда в ИМС Propeller запускается приложение. Мы в этом случае используем только один объект, поэтому после загрузки приложения автоматически запускается метод Main.

Когда выполняется первая строка из метода Main, Toggle(16, 3_000_000, 10), вызывается метод Toggle, и он выполняет свое действие: мигание светодиодом на выводе 16 пять раз, с задержками в промежутках по ¼ секунды. Затем, так как у Toggle больше нет исполнимого кода после цикла, выполнение возвращается к вызвавшему методу, Main, и продолжается на следующей его строке: Toggle(17, 2_000_000, 20). Когда исполняется эта строка, вызывается метод Toggle и мигает светодиодом на выводе 17 десять раз, с задержками в промежутках по 1/6 секунды. Затем метод Toggle опять передает выполнение назад методу Main, но у Main больше нет кода для выполнения, поэтому он завершается и приложение прерывается; *Cog* останавливается и ИМС Propeller переходит в режим малого потребления до следующего сброса или переключения питания.

Не смущайтесь того, как выглядит код. Два метода, Main и Toggle, представлены один рядом с другим, но они рассматриваются как различные подпрограммы, начинающиеся объявлением их блоков PUB и завершающиеся объявлением следующего блока или концом исходного кода (что окажется раньше). Другими словами, ИМС Propeller знает, что метод Toggle не является частью исполнимого кода метода Main.

Также заметьте, что в нашем примере мы все еще используем только один процессор (*Cog*), и все приложение выполняется последовательно: сначала мигание P16, затем остановка, мигание P17, остановка. В следующем упражнении мы научимся использовать несколько процессоров.

Упражнение5: Output.spin – Параллельное Выполнение

В упражнениях с 1 по 4 для выполнения приложения мы использовали всего один процессор; программа просто переключала один вывод P16, останавливалась, и затем переключала один вывод P17, после чего завершалась. Такое выполнение называется “последовательным выполнением.”

Предположим, однако, что мы хотим выполнять параллельные процессы: к примеру, одновременно переключать выводы 16 и 17, каждый на своей частоте и за разные промежутки времени. Конечно же, при разумном программировании подобные задачи могут быть решены и с использованием последовательного выполнения, но сделать это намного проще при параллельном выполнении, задействовав два из восьми процессоров ИМС Propeller. Посмотрите на следующий пример и отредактируйте свой код в соответствии с ним. Мы добавили блок переменных (VAR) и сделали небольшие изменения в методе Main.

```
{{Output.spin
Toggle two pins, ~~one after another~~ simultaneously.}}

VAR
  long  Stack[9]                'Stack space for new Cog

PUB Main
  Cognew(Toggle(16, 3_000_000, 10), @Stack)     'Toggle P16 ten…
  Toggle(17, 2_000_000, 20)                     'Toggle P17 twenty…

PUB Toggle(Pin, Delay, Count)
{{Toggle Pin, Count times with Delay clock cycles in between.}}

  dira[Pin]~~                   'Set I/O pin to output direction
  repeat Count                  'Repeat for Count iterations
    !outa[Pin]                  '  Toggle I/O Pin
    waitcnt(Delay + cnt)        '  Wait for Delay cycles
```

Блок VAR

В блоке VAR мы определили массив из *long*-ов с именем Stack, длиной в 9 элементов. Он используется в методе Main.

Метод Main

Мы изменили первую строку метода таким образом, что строка из его прежнего кода, с вызовом `Toggle`, заключена в команде **COGNEW**. Команда **COGNEW** запускает новый процессор для выполнения либо *Spin*-, либо ассемблер-кода. Для этого мы ввели `Toggle(16, 3_000_000, 10)` в качестве первого параметра, и `@Stack` – в качестве второго. Это значит, что **COGNEW** запустит новый *Cog* на выполнение метода `Toggle` и его память, начиная с адреса `Stack`, будет использована под стек. Символ **@** – это оператор взятия адреса, он возвращает реальный адрес переменной, следующей за ним

Для выполнения кода *Spin*, новый *Cog* нуждается в некотором рабочем пространстве памяти, называемом “область стека”, где он может сохранять временные элементы, такие как адреса возврата, величины возврата, промежуточные результаты вычислений и т.д. Мы решили зарезервировать область памяти из 9 *long* (36 байт), и передали адрес этой области как второй параметр, `@Stack`, команде **COGNEW**. Какого размера стек нам нужен? Это зависит от выполняемого кода *Spin* и мы обсудим это в деталях позднее. Сейчас давайте примем, что области в 9 *long* достаточно для нашего метода `Toggle`.

Откомпилируйте и загрузите Output.spin. Вы должны увидеть, что теперь светодиоды на выводах P16 и P17 мигают одновременно, с разными частотами, по 5 и 10 раз, соответственно. Происходит это потому, что сейчас мы запустили два процессора, работающих параллельно: один переключает P16, в то время как другой – P17.

Теперь о том, как это работает: *Cog0* начинает выполнять метод `Main` нашего приложения. Первая строка метода `Main` использует команду **COGNEW** для активизирования нового процессора (*Cog1*) для выполнения метода `Toggle` с переданными ему параметрами `(16, 3_000_000, 10)`. В то время, как *Cog1* стартует, *Cog0* продолжает выполнение на следующей строке метода `Main`, прямом вызове метода `Toggle` с переданными ему параметрами `(17, 2_000_000, 20)`. Таким образом, *Cog0* выполняет `Toggle` на выводе P17, в то время, как *Cog1* выполняет `Toggle` на P16. Когда их собственные задачи завершатся, каждый из них будет остановлен в результате отсутствия кода. *Cog1* останавливается, когда завершит `Toggle`. *Cog0* завершает `Toggle`, возвращается в `Main` и затем останавливается. В приведенном случае *Cog1* останавливается раньше, чем *Cog0*.

Этот процесс изображен на Рис. 3-10. ИМС Propeller загружает интерпретатор *Spin* в *Cog0* для выполнения приложения (две левые стрелки на рисунке). После этого приложение командой **COGNEW** требует запуска нового *Cog*, что заставляет ИМС Propeller загрузить интерпретатор *Spin* в следующий доступный процессор, *Cog1*, для выполнения маленькой части кода *Spin* из приложения - метод `Toggle` (две правые стрелки на рисунке). Каждый *Cog* выполняет свой код, полностью независимо от другого; это и есть настоящее параллельное выполнение. Отметьте, что к концу приложения оба процессора выполняют одну и ту же часть кода *Spin*, метод `Toggle`, но каждый из них использует свою собственную рабочую область и собственные значения для `Pin`, `Delay` и `Count`.

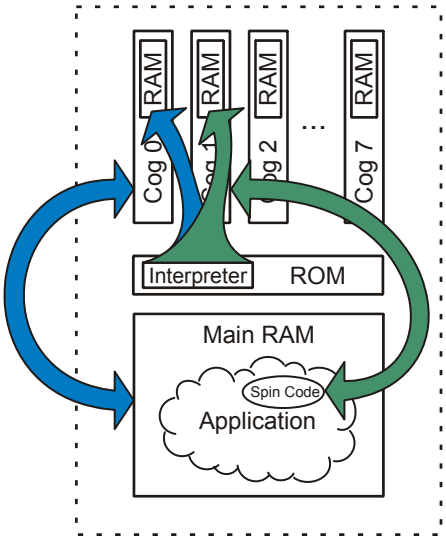


Рис. 3-10: Два *Cog*, выполняющих приложение Output и метод Toggle.

Отметьте: интерпретатор Spin загружается в ОЗУ каждого из Cog. Приложение Spin - Output, расположенное в Основном ОЗУ, интерпретируется в Cog 0, и оно запускает Cog 1 для выполнения только метода Toggle.

Кратко: Упр. 4 и 5

- Язык *Spin*:
  - Методы:
    - Для вызова методов этого же объекта, используйте *method*, где *method* – это имя метода, см. PUB на стр. 334.
    - Методы автоматически завершаются при завершении их кода, с возвратом выполнения вызывавшему.
    - Когда завершается первый метод приложения, приложение и процессор, его выполняющий, останавливаются.
  - Списки параметров
    - Методы имеют параметры в виде: *method*(*param1*, *param2*, и т.д.), см. PUB на стр. 334.
    - Параметры – это переменные размера *long*, доступные только в границах метода.
      - Они могут быть изменены в пределах метода, однако все соответствующие переменные, использовавшиеся при вызове для передачи параметров, изменены не будут.
  - Команда REPEAT:
    - Бесконечный цикл: `repeat`
    - Конечный цикл: `repeat` *expression,* где *expression* выражает необходимое количество повторений цикла, см. REPEAT на стр. 340.
  - Массивы:
    - Массивы определяются в виде *symbol* [*count*], где *symbol* – это символическое имя массива и *count* – это количество элементов в массиве, см. VAR на стр. 364.
  - Команда COGNEW:
    - Запускает другой *Cog* (процессор) для выполнения *Spin*- либо ассемблер- кода, см. COGNEW на стр. 221.
    - Обеспечивает настоящее параллельное выполнение.
    - Нуждается в адресе для резервирования рабочей области памяти под стек для выполнения кода *Spin*.
  - Оператор взятия адреса (`@`) возвращает адрес переменной, следующей за нею. См. Адрес идентификатора '`@`' на стр. 324.

Упражнение 6: Output.spin и Blinker1.spin – Используем наш объект

Теперь самое время исследовать преимущества объектов. Все предыдущие упражнения создавали приложение, состоящее только из одного объекта; объект Output.spin был собственно всем приложением. Это обычный путь в начале разработки новых объектов. Предположим, что целью всей предыдущей работы было создать объект, используя который, другие разработчики могли бы с легкостью переключать одну или более линий В/В. Возможно, такой объект и не имеет глубокого практического смысла, однако давайте будем его использовать как удобный и простой пример для обучения.

Пришло время наделить наш объект Output возможностью взаимодействия с другими объектами. Отредактируйте Ваш код, чтобы он выглядел следующим образом:

Пример объекта: Output.spin

```
{{ Output.spin }}
Toggle two pins, one after another.}}

VAR
  long  Stack[9]              'Stack space for new Cog

PUB Main
  Cognew(Toggle(16, 3_000_000, 10), @Stack)      'Toggle P16 ten…
  Toggle(17, 2_000_000, 20)                      'Toggle P17 twenty…

PUB Start(Pin, Delay, Count)
{{Start new toggling process in a new Cog.}}

  Cognew(Toggle(Pin, Delay, Count), @Stack)

PUB Toggle(Pin, Delay, Count)
{{Toggle Pin, Count times with Delay clock cycles in between.}}

  dira[Pin]~~                 'Set I/O pin to output direction
  repeat Count                'Repeat for Count iterations
    !outa[Pin]                '  Toggle I/O Pin
    waitcnt(Delay + cnt)      '  Wait for Delay cycles
```

Убедитесь, что имя файла объекта - “Output.spin” – он будет нам далее необходим.

Метод Start

Мы заменили метод `Main` методом `Start`. Метод `Start` запускает еще один *Cog* для независимого выполнения метода `Toggle`, передав ему параметры `Pin`, `Delay`, и `Count`.

Интерфейс с объектом осуществляется с использованием его *Public*-методов (**PUB**), и наш объект Output сейчас имеет два интерфейсных компонента: методы `Start` и `Toggle`.

Теперь наш объект Output может быть использован другими объектами для переключения любой линии с любой частотой и необходимым количеством повторений. Кроме того, выполнять это они могут последовательно, вызвав метод `Toggle` объекта Output, или параллельно с другими задачами, вызвав его метод `Start`.

Давайте создадим другой объект, который использует объект Output. Для этого выберем File → New из меню, при этом появится новая вкладка для редактирования. На этой новой странице введем следующий код. Обратите внимание на выделенные элементы, поскольку мы их будем вскоре обсуждать.

Пример объекта: Blinker1.spin

```
{{ Blinker1.spin }}

OBJ
  LED : "Output"

PUB Main
{Toggle pins at different rates, simultaneously}
  LED.Start(16, 3_000_000, 10)
  LED.Toggle(17, 2_000_000, 20)
```

Сохраните этот новый объект как “Blinker1.spin” в той же папке, где сохранен Output.spin. Теперь, находясь на вкладке Blinker1, нажмите *F10* для компиляции и загрузки. Хотя использовался другой программный метод, светодиоды должны мигать так же, как в Упражнении 5; Blinker1 использовал наш объект Output и вызывал его методы `Start` и `Toggle`.

Теперь опишем, как это работало. В Blinker1 мы имеем *object*-блок (**OBJ**) и *Public*-метод (**PUB**). Строка *object*-блока `LED : "Output"` объявляет, что мы собираемся использовать другой объект, с названием Output, и что мы будем обращаться к нему как `LED` в рамках текущего объекта Blinker1.

Ссылка объект-метод

В *Public*-методе, `Main`, мы имеем два вызова методов. Помните, как мы учили в Упражнении 4, что один метод может вызвать другой, просто сославшись на его имя? Это работает для методов, принадлежащих этому же объекту, но сейчас нам нужно вызвать метод из другого объекта. Чтобы это сделать, мы используем формат *object*`.`*method,* где *object* – это символическое имя, которое мы присвоили объекту в блоке `OBJ` (в этом случае `LED`) а *method* – это имя метода этого объекта. Этот прием называется ссылкой объект-метод. Объект Blinker1 обращается к объекту Output как `LED`, поэтому `LED.Start` вызывает метод `Start` объекта Output, а `LED.Toggle` вызывает его метод `Toggle`.

Поскольку Blinker1 ссылается на Output, то при компиляции Blinker1, в приложение компилируются оба этих объекта. Это показано на Рис. 3-11. Эта же структура показана в Виде Объекта, который мы рассмотрим далее.

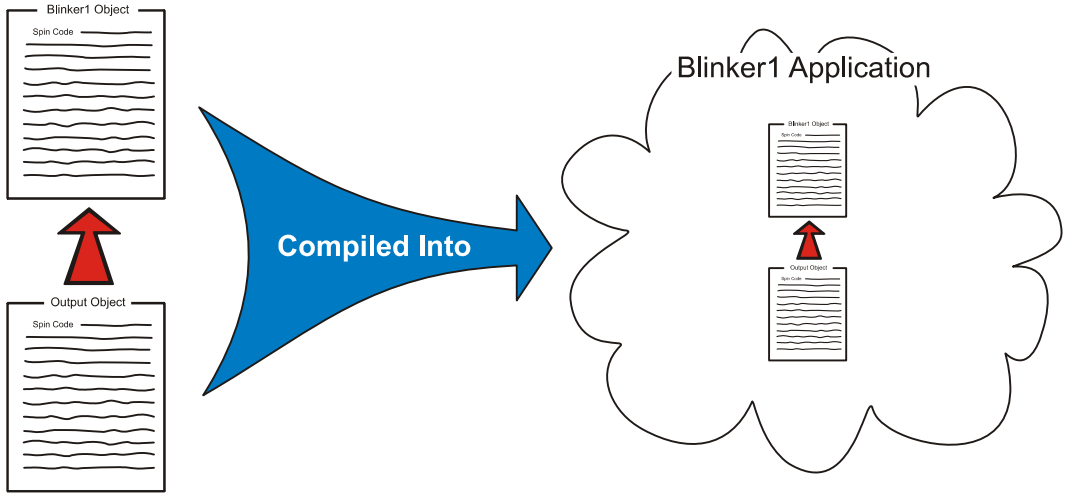


Рис. 3-11: Иерархия Blinker1 и приложение Blinker1

Вид объекта

Когда Вы откомпилировали Blinker1, панель Вид Объекта (Object View) обновилась и отображает структуру приложения. Эта панель расположена в левом верхнем углу экрана программы *Propeller Tool*, если открыт интегрированный браузер (см. Панель 1: на стр.47.) Как она должна выглядеть сейчас, показано на Рис. 3-12.

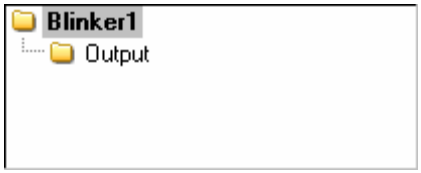


Рис. 3-12: Вид объекта Blinker1

Панель Object View обновляется после каждой успешной компиляции приложения, показывая логическую структуру этого приложения. Вид, показанный на Рис. 3-12 – это способ панели Object View отображения логической структуры, приведенной на Рис. 3-11. При компиляции необходимо хотя бы изредка проверять Object View для выяснения и, при необходимости, устранения проблем.

Структура всего Вашего приложения, либо, по крайней мере, ее вид после последней успешной компиляции, отображено в панели Object View. Её можно использовать и для анализа приложения. Например, указание мышью каждого объекта в Object View дает Вам подсказку с информацией об этом объекте. Левый клик на каждом из них либо открывает его, либо переключает на него активную вкладку редактора. Правый клик на каждом из этих объектов делает то же, что и левый, но одновременно переключает режим просмотра в просмотр Документации, а не Полного Исходного Кода.

Верхний Объектный Файл

Объект в верхней части панели Object View – это “Верхний Объектный Файл” (Top Object File) для последней успешной компиляции. Это значит, что в нашем случае компиляция начиналась с объекта “Blinker1”. По нажатию клавиши *F10* или *F11*, либо при выборе соответствующего пункта меню, программа *Propeller Tool* начала процесс компиляции, используя любую активную на данный момент вкладку. Активная вкладка – это та, которая подсвечена отлично от всех остальных; для примера см. Панель 4: Панель редактора на стр. 49 и Рис. 2-4 на стр. 49.

Если мы случайно сначала кликнули на вкладке объекта Output, а затем дали команду на компиляцию *F10* или *F11*, компиляция начнется именно с этого объекта, а не с верхнего. Это не создаст необходимого нам приложения, и панель Object View покажет в своей структуре только один объект – Output. Это произошло из-за того, что мы использовали опцию “Compile Current” (компилировать текущий), что означает компиляцию текущего активного объекта или редактируемой вкладки.

Существуют другие опции компиляции, которые могут нам помочь. Выберите меню Run (Выполнить) и посмотрите на доступные опции. Вы должны увидеть выпадающие меню “Compile Current” и “Compile Top” (Рис. 3-13).

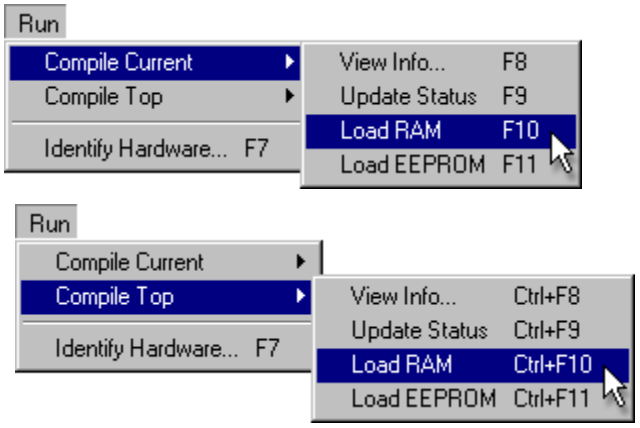


Рис. 3-13: Меню Compile Current (вверху) и меню Compile Top (внизу)

Каждое из меню ─ Compile Current и Compile Top ─ имеют одинаковые подменю, но процесс компиляции они начинают из различных мест. Меню Compile Current начинает с активной вкладки, а меню Compile Top ─ с назначенного Top Object File.

Вы в любой момент можете указать программе *Propeller Tool*, какой именно объект считать верхним "Top Object File". Сделать это можно одним из следующих способов:

1) Правый клик на вкладке необходимого объекта, выбрать "Top Object File," либо
2) Правый клик на необходимом объекте из перечня файлов (в интегрированном файловом браузере), выбрать "Top Object File," либо
3) Выбрать меню File → Select Top Object File… и указать необходимый файл из окна просмотра, либо
4) Нажать *Ctrl+T* и выбрать необходимый файл из окна просмотра.

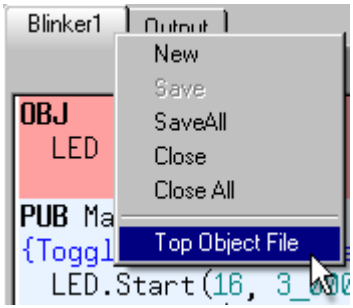
На рисунке ниже мы использовали способ #1 для выбора Blinker1 как Top Object File. Отметьте, что после этого текст названия вкладки Blinker1 стал жирным; см. Рис. 3-14b. Файл, который указан в программе *Propeller Tool* как Top Object File, всегда отображается жирным шрифтом.

Теперь, если мы используем одну из опций компиляции Compile Top, такие как *Ctrl+F10* или *Ctrl+F11*, *Propeller Tool* начнет компиляцию, начиная с Top Object File, независимо от того, какая вкладка активна. Например, на Рис. 3-14b объект Output находится на активной вкладке. Если мы нажмем *Ctrl+F10*, приложение также будет откомпилировано начиная с объекта Blinker1. А если бы мы нажали *F10*, откомпилировался бы объект Output.

Каждое сочетание клавиш для опций компиляции Compile Current – *F8*, *F9*, *F10*, и т.д., имеет похожие сочетания для компиляции Compile Top: *Ctrl+F8*, *Ctrl+F9*, *Ctrl+10*, и т.д.

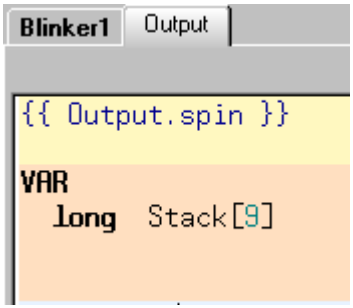
Рис. 3-14: Установка Blinker1 в качестве Top Object File

a.



Один из способов установки Top Object File – это правый клик на нужной вкладке и выбор «Top Object File».

b.



Имя Top Object File будет отображаться на его вкладке жирным шрифтом.

Какие объекты были откомпилированы?

Если возникает вопрос о том, какие файлы объектов были откомпилированы при последней успешной операции компиляции, используйте мышь для анализа структуры полученного приложения в панели Object View.

Важно отслеживать, какой файл Вы установили как Top Object File и какую опцию компиляции Вы выбрали – Текущий (Current) или Верхний (Top). Только один файл может быть установлен как Top Object File в данный момент, и *Propeller Tool* запоминает этот файл между сессиями.

Следует также помнить, что объект вовсе не должен быть открытым в *Propeller Tool* только лишь для того, чтобы быть откомпилированным. Если компилируемый объект ссылается на другой объект, последний будет также откомпилирован, не зависимо от того, открыт он в данный момент, или нет. Даже Top Object File может быть

откомпилирован, когда он не открыт. Например, нажатие *Ctrl+F10* откомпилирует последний установленный Top Object File независимо даже от того, принадлежит ли он к текущему приложению, над которым Вы работаете.

Кратко: Упр. 6

- Язык *Spin*:
  - Методы:
    - Для вызова методов из другого объекта, используйте *object*`.`*method* где *object* – это символическое имя объекта (присвоенное ему в секции `OBJ`), а *method* – это имя метода в рамках того другого объекта. См. `OBJ` на стр. 288.
    - Методы *Public* (`PUB`) являются интерфейсом объекта; другие объекты вызывают его *Public*-методы. См. `PUB` на стр. 334.
- Object View (Вид Объекта)
  - Отображает структуру последнего успешно откомпилированного приложения. См. «Вид Объекта (Object View)», стр. 64.
  - Установка мыши на отображаемые объекты показывает подсказку о них.
  - Левый клик на показанном объекте либо открывает его, либо делает его вкладку активной.
  - Правый клик на показанном объекте открывает его либо переключает режим его просмотра на Режим Документации.
- Compile Current (Компилировать Текущий) – (с *F8* по *F11*) – компилирует, начиная с текущего объекта (активной вкладки).
- Compile Top (Компилировать Верхний) – (с *Ctrl+F8* по *Ctrl+F11*) – компилирует, начиная с Top Object File.
- Top Object File (Верхний объектный файл):
  - Отображается жирным шрифтом в имени вкладки и списке файлов.
  - Может быть установлен одним из способов (и откомпилирован операцией Compile Top):
    1) Правый клик на вкладке объекта и выбор “Top Object File,” либо
    2) Правый клик на объекте в перечне файлов, “Top Object File,” либо
    3) Меню File → Select Top Object File…, указать необходимый файл, либо
    4) Нажать *Ctrl+T* и выбрать необходимый файл из окна просмотра.
- Объекты не обязательно открывать, чтобы откомпилировать; они так же компилируются при компиляции других объектов или при команде Compile Top.

Объекты и Процессоры

Важно понимать, что между объектами и процессорами нет прямых зависимостей. Помните, в Упражнении 5 нами использовался всего один объект и два процессора, а в Упражнении 6 использовалось два объекта и два процессора, однако каждое из этих упражнений могло бы использовать всего один *Cog*, если бы было нужно выполнять процесс последовательно. Когда и как используются процессоры, полностью определяется приложением и разработчиком, который его написал.

Упражнение 7: Output.spin – Совершенствуем далее

Давайте добавим несколько серьезных доработок в наш объект Output. Сейчас для последовательного переключения вывода может быть вызван метод `Toggle`, а если необходимо запустить его как отдельный независимый процесс, вызывается метод `Start`. Но мы не обеспечили возможность для остановки процесса после его запуска или, по крайней мере, для начала, способ для определения, выполняется ли сам процесс. Кроме того, вдобавок к имеющемуся варианту с заданным количеством переключений, было бы хорошо иметь опцию для бесконечно долгого переключения вывода.

Давайте добавим метод `Stop` для остановки запущенного процесса и метод `Active` для проверки, выполняется ли в данный момент параллельный процесс. Вдобавок, мы усовершенствуем метод `Toggle`, как планировали выше.

Для объектов такого рода, считается обычной и правильной практикой использовать имя “Start” для метода, который запускает новый процессор, и имя “Stop” – для метода, который останавливает ранее запущенный таким образом процессор. При таком подходе, другим разработчикам проще понять, как использовать объект, когда при просмотре объекта в режиме документации или сжатом режиме они видят `Start` и `Stop` и могут предположить, что объект запускает/останавливает другой процессор. Для объектов, которые не запускают другого процессора, но все же нуждаются в некоторой инициализации, в качестве имени метода рекомендуется использовать имя “`Init`”.

Этот код был подвержен серьезным изменениям; будьте готовы, — понадобятся определенные усилия для его понимания, однако знания, которые Вы приобретете, окупят Ваши старания!

Вот код; откорректируйте свой в соответствии с ним:

```
{{ Output.spin }}

VAR
  long  Stack[9]                        'Stack space for new Cog
  byte  Cog                             'Hold ID of Cog in use, if any


PUB Start(Pin, Delay, Count): Success
{{Start new blinking process in new Cog; return TRUE if successful}}

  Stop
  Success := (Cog := Cognew(Toggle(Pin, Delay, Count), @Stack) + 1)


PUB Stop
{{Stop toggling process, if any.}}

  if Cog
    Cogstop(Cog~ - 1)


PUB Active: YesNo
{{Return TRUE if process is active, FALSE otherwise.}}

  YesNo := Cog > 0


PUB Toggle(Pin, Delay, Count)
{{Toggle Pin, Count times with Delay clock cycles in between.}}
  If Count = 0, toggle Pin forever.}}

  dira[Pin]~~                           'Set I/O pin to output…
  repeat Count                          'Repeat for Count iterations
  repeat                                'Repeat the following
    !outa[Pin]                          '  Toggle I/O Pin
    waitcnt(Delay + cnt)                '  Wait for Delay cycles
  while Count := --Count #> -1          'While not 0 (make min -1)
  Cog~                                  'Clear Cog ID variable
```

Блок переменных VAR

В блоке VAR мы добавили байтовую переменную Cog. Она будет использоваться для отслеживания номера *ID* процессора, запущенного методом Start, если таковой имеется. Переменные Stack и Cog являются глобальными по отношению к объекту, и они могут использоваться в любом из блоков PUB или PRI в объекте Output. Если они изменятся в одном методе, то другие методы будут видеть новые значения, когда будут на них ссылаться.

Метод Start

Как мы решили, удобно иметь возможность узнать, был ли успешным вызов метода Start или нет. Поскольку в ИМС Propeller ограниченное количество процессоров, есть вероятность, что метод Start не сможет в очередной раз запустить новый процессор. По этой причине мы добавим ему свойство возвращать значение (логическое TRUE или FALSE) как результат исхода его запуска: "`: Success`" в его объявлении значит, что значение, которое метод будет возвращать, мы назвали Success («Успешно», англ.). Каждый из методов PUB и PRI всегда возвращает величину *long* (4 байта), не зависимо от того, была ли она указана при объявлении метода. При написании метода со свойством возврата осмысленной величины, считается хорошей практикой объявлять возвращаемое значение так, как мы сделали в этом примере. Наша переменная Success стала копией встроенной в метод переменной RESULT, и теперь мы можем присвоить либо Success , либо RESULT значение, которое будет возвращено при выходе из метода.

Метод Start сейчас выполняет два действия: во-первых, он останавливает любой существующий процесс, и, во-вторых, запускает новый процесс. Сначала он вызывает метод Stop на случай того, если Start вызывался несколько раз без вызова Stop, извне объекта. Если этого не сделать, новый процессор при своем старте может повредить переменные, такие как Stack, уже запущенного.

Следующая строка похожа на нашу исходную, но выглядит несколько громоздко, поскольку является составным выражением. Сейчас мы проанализируем ее по частям, начиная с середины. Часть строки COGNEW осталась такой же, как и была ранее: `Cognew(Toggle(Pin, Delay, Count), @Stack)`. Она запускает новый процессор на выполнение метода Toggle. Чего, возможно, Вы еще не знаете – это то, что COGNEW всегда возвращает *ID* (идентификатор, номер) запущенного процессора, от 0 до 7, либо -1, если не было процессора, доступного для запуска. В предыдущей версии объекта Output мы попросту игнорировали возвращаемое значение. Однако в этот раз мы используем возвращаемое COGNEW значение в выражении и присваиваем результат переменной: `Cog := Cognew(Toggle(Pin, Delay, Count), @Stack) + 1`. Это выражение обозначает запустить COGNEW, взять возвращенное им значение и прибавить к нему 1, а

затем присвоить результат переменной с именем Cog. Символ ‘:=’ – это оператор присваивания, эквивалентный оператору равенства ‘=’ в некоторых других языках.

Мы будем использовать переменную Cog для запоминания *ID* запущенного процессора, и таким образом позднее мы сможем при необходимости его остановить. Для чего мы добавили к нему 1, станет ясно через минуту.

Мы еще не закончили с текущей строкой. Слева от части Cog := … находится выражение присваивания Success :=. Таким образом, после того, как *ID* нового *Cog* будет возвращен, увеличен на 1 и сохранен в Cog, результат также сохраняется и в переменной Success. Помните, что мы приняли Success как возвращаемую методом Start логическую величину? Логическое значение FALSE соответствует для нее численному значению 0, а TRUE соответствует -1, однако в операциях логического сравнения считается, что ноль (0) это FALSE, а любое ненулевое значение (≠ 0) это TRUE. Это очень удобно, и именно поэтому мы добавили 1 к возвращаемому COGNEW значению; диапазон от -1 до 7 преобразуется в таковой от 0 до 8. Тогда 0 (FALSE) обозначает, что ни один процессор не был запущен, в то время как значение от 1 до 8 (TRUE) значит, что процессор запущен и собственно представляет его номер.

Таким образом, в этой одной строке кода мы запустили (вероятно) новый *Cog*, передали ему ссылку на подпрограмму Toggle и область памяти под стек, сохранили в переменной Cog номер *ID* нового запущенного процессора, увеличенный на 1, и присвоили этот результат возвращаемой методом Start переменной Success! Эта строка демонстрирует одно из мощных свойств языка *Spin*: составные выражения с присваиванием промежуточных результатов.

Внешние скобки, заключающие часть Cog :=…, не являются необходимыми, но мы их добавили, чтобы помочь Вам отделить два различных присваивания переменных: сначала была обновлена переменная Cog, после чего результат был присвоен переменной Success. Для того чтобы помочь Вам в понимании структуры сложных выражений, подобных этому, программа *Propeller Tool* временно выделяет соответствующую пару скобок в текущей позиции курсора, жирным шрифтом. Установите курсор в произвольную позицию в строке, чтобы увидеть этот эффект. Приведенный ниже рисунок (Рис. 3-15) описывает это: звездочкой обозначена текущая позиция курсора, стрелки показывают утолщенные парные скобки, а затененная область – это выражение, заключенное в эти скобки (в реальности тень не отображается).

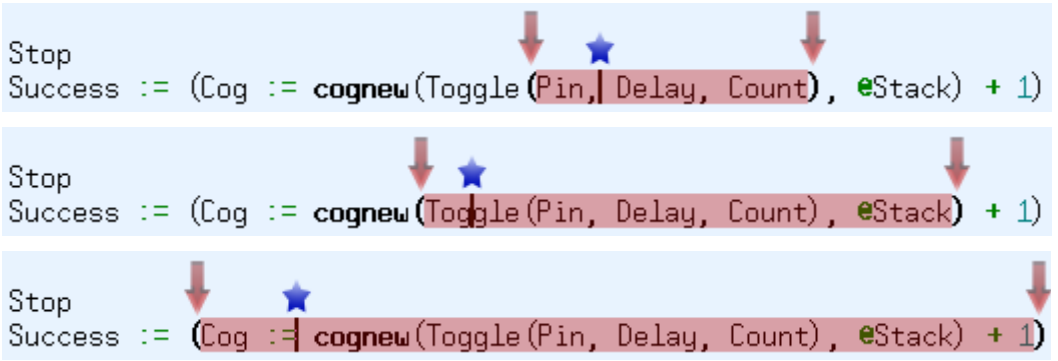


Рис. 3-16: Утолщенные парные скобки

Парные скобки, заключающие выражение, внутри которого находится текущая позиция курсора, временно отображаются утолщенным шрифтом. Пользуйтесь этим свойством для разбора сложных выражений.

Метод Stop

Наш метод `Stop` предназначен для остановки *Cog*, который был запущен методом `Start`. Выражение `if Cog` – это условная структура, обозначающая “если переменная *Cog* имеет значение `TRUE` , выполнить следующий форматированный отступом блок”. Помните, что `Cog` установлен в 0, если ни один из процессоров не был запущен, и установлен в значение от 1 до 8, если соответствующий *Cog* был запущен. Поскольку 0 значит `FALSE`, а не-0 значит `TRUE`, выражение `IF` дает «истину» только если мы действительно запустили процессор.

Выражение `COGSTOP` введено с отступом под условным выражением `IF`, поэтому оно выполняется только если условие `IF` дает «истину». Команда `COGSTOP` останавливает процессор, чей *ID* представлен в ее параметре: `Cog~ - 1`. Это еще одно хитрое, но мощное выражение языка *Spin*. Помните оператор пост-установки, «~~», из предыдущих упражнений? Аналогично, одиночный символ «~», идущий за переменной, является оператором пост-очистки; он очищает переменную, стоящую перед ним, в ноль (0). Эти операторы называются “пост-операторами”, поскольку они выполняют свое действие “после” того, как исходное значение переменной было использовано выражением, в которое она вовлечена. Таким образом, `Cog~ - 1` берет значение `Cog`, отнимает 1, передает это значение команде `COGSTOP`, после чего очищает в ноль (0). По сути, выражение `Cogstop(Cog~ - 1)` останавливает процессор с номером *ID*, равным `Cog-1`, а затем очищает переменную `Cog` в 0, и дальнейшие обращения к `Cog` покажут, что дополнительных запущенных процессоров сейчас нет.

Метод Active

Метод Active очень прост, он устанавливает свое возвращаемое значение, YesNo, в TRUE, если Cog больше 0, а иначе – в FALSE. Символ «>» - это оператор «Если Более, Чем». Отметьте, что мы также могли просто установить YesNo равным значению Cog, поскольку ноль рассматривается как FALSE, а неноль - как TRUE; это могло бы иметь дополнительные преимущества, будучи одновременно как возвращаемым значением истина/ложь, так и реальным *ID* процессора, используемого данным объектом.

Метод Toggle

Мы сделали пару небольших, но важных доработок в методе Toggle. Во-первых, давайте посмотрим на последнюю строку, Cog~. Помните, что если вызван метод Start, он запускает метод Toggle в другом процессоре и сохраняет *ID* этого процессора в переменной Cog. Когда Toggle завершается, его процессор также останавливается, но переменная Cog продолжает содержать его *ID*, предоставляя методам Active и Start недостоверную информацию о том, что якобы этот процессор еще активен. Мы добавили Cog~ в конце метода Toggle для очистки переменной Cog в ноль (0) для обеспечения верности кода.

Кроме того, мы хотели доработать Toggle для обеспечения возможности реализации бесконечных циклов наряду с конечными. Следующая наша доработка обеспечивает это искусным способом. Параметр Count – это количество переключений вывода. Это значит, что нет смысла устанавливать Count равным 0… Кому понадобится переключать вывод ноль раз? Поэтому мы сделаем значение 0 исключением, которое обозначает “переключать вывод бесконечно”.

Мы изменили цикл с repeat Count на repeat..while. Условие while находится в конце цикла, на три строки ниже repeat. Это – другая форма структуры цикла REPEAT, называемая “условный цикл Один-множество”. Он выполняет заключенный в него блок как минимум один раз, и повторяется еще и еще, пока условие “while” дает результат «истина». В нашем случае он повторяется, пока Count := --Count #> -1 дает TRUE (т.е.: не ноль). Это условие – еще одно составное выражение. Двойной минус, --, предшествующий Count – это оператор Пре-декремента; он декрементирует переменную Count на 1 перед тем, как ее значение будет использовано в выражении. Оператор #> - это оператор Ограничения по Минимуму; он берет значение слева от себя и возвращает либо это же значение, либо число справа, какое из них будет больше. Таким образом, каждый раз при вычислении этого выражения, Count декрементируется на 1, результат ограничивается до -1 или большего, и уже это финальное значение присваивается обратно переменной Count. Это выражение имеет интересный эффект, который мы опишем далее.

Если Toggle был вызван с Count равным 2, цикл будет выполняться дважды, именно так, как мы и хотели. После первой итерации, выражение while Count := --Count #> -1 декрементирует Count, сделав его равным 1, затем ограничит его до -1 или больше (так и останется 1) и сохранит результат в Count. Поскольку результат, 1 – это не ноль (TRUE), цикл повторится вновь. После второй итерации, оператор WHILE декрементирует Count, сделав ее 0, ограничит до -1 или более (так и останется 0) и сохранит результат в Count. Поскольку 0 - это FALSE, условие WHILE прервет выполнение цикла.

Так это работает для всех «нормальных» значений Count, но что, если Toggle будет вызван с параметром Count, равным 0? После первой итерации while Count := --Count #> -1 уменьшит Count, сделав ее -1, затем ограничит до значения -1 или более (при этом она так и останется -1) и сохранит значение в Count. Поскольку результат, -1, не равен нолю (TRUE), цикл повторяется далее. После второй итерации, оператор в WHILE уменьшает Count, сделав ее -2, ограничивает до значения -1 или большего (т.е. устанавливает значение -1) и сохраняет в Count. Опять, поскольку результат, -1, не равен нолю (TRUE), цикл продолжит выполнение.

Итак, если значение Count при старте равно 0, цикл повторяется бесконечно долго! Если же значение Count при старте больше 0, цикл повторяется лишь заданное количество раз!

Кратко: Упр. 7

- Объекты:
  - Не имеют прямых зависимостей с процессорами.
  - Должны вызывать интерфейсные методы “Start” и “Stop”, если они влияют на другие процессоры.
  - Должны вызывать интерфейсный метод “Init”, если им необходима инициализация.
- Язык *Spin*:
  - Переменные, определенные в блоках переменных, - глобальные для объекта, поэтому изменения, выполненные одним методом, видны в других методах. См VAR, стр. 364.
  - *Booleans*: (См. «Предопределенные Константы», стр. 237 и «Операторы Spin», стр. 291).
    - FALSE = 0
    - TRUE = -1; любое не равное нолю (≠0) значение является Истиной (True) для логических сравнений.
  - Составные выражения могут включать «Промежуточные присваивания», см. стр. 295.
  - Операторы:
    - *“Pre”/“Post”* операторы выполняют свои действия перед/после использования значения переменной в выражении.
    - «Присваивание ‘:=’» похоже на равенство ‘=’ в других языках, см. «Присваивание переменных ‘:=’», стр. 297.
    - «Пост-Очистка ‘~’» очищает переменную, предваряющую его, в ноль (0), см. «Распространение Знака 7 или Пост-Очистка ‘~’», стр.305.
    - «Пре-Декремент ‘--’» уменьшает переменную, следующую за ним, присваивая выражению результат, см. «Декремент, пре- или пост- ‘--’», стр. 299.
    - «Более Чем ‘>’» возвращает «Истину», если значение слева больше чем таковое справа, см. «Логическое Больше ‘>’, ‘>=’», стр. 322.
    - «Ограничение Минимума ‘#>’» возвращает большее из значений слева и справа, см. «Ограничение по минимуму ‘#>’, ‘#>=’», стр. 303.
  - Методы (См. PUB, стр. 334):
    - Всегда возвращают величину *long* (4 байта) не зависимо от того, была ли она указана в объявлении.

        - Содержат встроенную локальную переменную, `RESULT`, которая содержит возвращаемое значение.
        - Возвращаемые значения объявляются после имени метода и параметров с двоеточием (`:`) и именем возвращаемой величины.
    - `COGNEW` возвращает *ID* (от 0 до7) запущенного *Cog*; -1 если такового нет, см. `COGNEW`, стр. 221.
    - `COGSTOP` останавливает *Cog* с заданным *ID*, см. `COGSTOP`, стр. 227.
    - `IF` – это условная структура, которая выполняет принадлежащей ей следующий за ней блок кода, если условие - «истина», см `IF`, стр.257.
    - Условный цикл `REPEAT`, вид «Один-множество»: `REPEAT WHILE` *Condition* выполняется как минимум один раз и продолжается, пока *Condition* равно «истина». См. `REPEAT`, стр. 340.
- Программа *Propeller Tool* утолщает парные скобки, заключающие в себе текущую позицию курсора.

Упражнение 8: Blinker2.spin – Много объектов, много процессоров

Сейчас давайте создадим новый объект, который будет использовать преимущества усовершенствованного объекта Output для использования многих процессоров с целью выполнения многих параллельных процессов. Вот код:

Пример объекта: Blinker2.spin

```
{{ Blinker2.spin }}

CON
  MAXLEDS = 6                'Number of LED objects to use

OBJ
  LED[6] : "Output"


PUB Main
{Toggle pins at different rates, simultaneously}

  dira[16..23]~~                                  'Set pins to outputs
  LED[NextObject].Start(16, 3_000_000,   0)       'Blink LEDs
  LED[NextObject].Start(17, 2_000_000,   0)
  LED[NextObject].Start(18,   600_000, 300)
```

```
  LED[NextObject].Start(19, 6_000_000,  40)
  LED[NextObject].Start(20,   350_000, 300)
  LED[NextObject].Start(21, 1_250_000, 250)
  LED[NextObject].Start(22,   750_000, 200)      '<-Postponed
  LED[NextObject].Start(23,   400_000, 160)      '<-Postponed
  LED[0].Start(20, 12_000_000, 0)                'Restart object 0
  repeat                                         'Loop endlessly


PUB NextObject : Index
{Scan LED objects and return index of next available LED object.
 Scanning continues until one is available.}

  repeat
    repeat Index from 0 to MAXLEDS-1
      if not LED[Index].Active
        quit
  while Index == MAXLEDS
```

Откомпилируйте и загрузите Blinker2. Вы должны увидеть, как шесть светодиодов начнут мигать с различными, независимыми частотами и количеством раз. Присмотритесь: после примерно 8 секунд P20 перестанет мигать, а P22 - начнет. Несколько секунд позже, P18 остановится и P23 - начнет, затем P16 закончит, а P20 начнет снова, но с другой частотой. В конце концов, все, кроме P17 и P20, потухнут. Можете объяснить, почему они ведут себя именно так? Мы покажем это ниже.

Блок OBJ

В блоке объектов мы определили массив объектов Output, названный `LED`, с шестью элементами. Таким образом, мы можем получить шесть одновременно выполняемых независимых процессов.

Метод Main

Первая строка метода `Main`, `dira[16..23]~~`, устанавливает линии B/B с 16 по 23 на вывод. Регистры B/B **`DIRA`**, **`OUTA`**, и **`INA`**, могут использовать эту форму для воздействия на несколько смежных линий. Мы устанавливаем эту группу линий B/B на вывод лишь для того, чтобы предотвратить нежелательные эффекты, связанные с резисторами между парами линий B/B с 18 по 23 на плате Propeller Demo Board. Если какой-либо процессор один устанавливает определенную линию как выход, то при выключении эта линия опять станет входом, позволяя резистору между нею и соседней линией

повлиять на ее светодиод. Мы оставим процессор, выполняющий приложение, активным, поэтому наши результаты предсказуемы.

Следующие девять строк, `LED[...`, вызывают метод `Start` объекта Output для запуска нового процессора и переключения различных выводов с различными частотами. Строки в виде `LED[NextObject].Start...`, вызывают метод `NextObject` для получения следующего индекса в массиве. Мы вскоре опишем метод `NextObject` более детально, но в двух словах он возвращает индекс следующего доступного объекта Output в массиве `LED` (т.е. индекс первого свободного объекта) или ожидает, пока такового не будет.

Мы имеем лишь шесть объектов Output, определенных для массива `LED`, поэтому первые шесть вызовов `Start` должны пройти быстро, каждый доступаясь к объектам массива `LED` с индексами с 0 по 5 и запуская в общем 6 дополнительных процессоров. Первые два имеют параметр `Count`, равный 0, поэтому они будут переключать свои линии бесконечно; последние же четыре будут останавливаться по мере выполнения заданного количества переключений.

Седьмая строка, `LED[NextObject].Start(22, 750_000, 200)`, сначала вызовет `NextObject` для получения индекса следующего доступного объекта, но поскольку все шесть объектов заняты выполнением переключений своих выводов, `NextObject` будет ожидать и не вернется в `Main`, пока не дождется какого-либо одного, который завершил свою задачу. Как оказывается, объект с индексом 4 (линия B/B 20) завершает свою задачу первым и выключается. Тогда метод `NextObject` возвращает значение 4, позволяя выполниться методу `Start` объекта с этим индексом, который перезапустит свободный *Cog* на переключение вывода 22. похожий процесс происходит с восьмой линией, `LED[NextObject].Start(23, 400_000, 160)`; все объекты заняты, поэтому `NextObject` откладывает дальнейшее выполнение, пока один не освободится, в этом случае это объект с индексом 2.

Сразу же после выполнения восьмой строки, выполняется девятая, `LED[0].Start(20, 12_000_000, 0)`. Это выражение не похоже на предыдущее тем, что оно не вызывает `NextObject`, а вместо этого использует фиксированный индекс, равный 0. Это значит, что объект из массива `LED` с индексом 0, который сейчас занят бесконечной задачей переключения линии B/B 16, внезапно обнаруживает свой метод `Start` вызванным вновь. Это приводит к немедленной остановке *Cog*, занятого переключением P16 и его же дальнейшему запуску, но уже для задачи переключения вывода P20.

Последняя строка, `repeat`, находится здесь лишь для того, чтобы оставить *Cog*, выполняющий приложение, активным. Она создает бесконечный цикл, который не выполняет никакого кода, поскольку в его теле ничего нет. Если бы *Cog* приложения

остановился, линии B/B, которые он установил как выходы, могли переключиться назад и стать входами, порождая странные эффекты из-за резисторов между некоторыми парами светодиодов на плате Propeller Demo Board. Если Вы не используете Propeller Demo Board, первая и последняя строки в методе Main не обязательны.

Метод NextObject

У нас в коде есть шесть объектов LED и любое их количество может быть занято параллельно в любой момент времени. Задача метода NextObject – это сообщить нам, какой из них свободен либо отложить дальнейшее выполнение, пока какой-либо из объектов не освободится. Чтобы сделать это, он сканирует по всем шести объектам LED в поисках первого попавшегося, который не занят как параллельный процесс, и возвращает индекс в LED-массиве, представляющий этот свободный объект. Если все шесть в данный момент выполняют задачи, метод продолжает сканирование, пока один не освободится. Для того чтобы справиться с этой задачей, метод NextObject использует метод Active нашего объекта Output.

Мы имеем два вложенных цикла **REPEAT**. Внешний цикл, repeat..while Index == MAXLEDS, повторяется до тех пор, пока Index не станет равен MAXLEDS, 6 в нашем случае. В предыдущем упражнении мы изучили, как работает этот тип цикла.

Внутренний же цикл **REPEAT**, repeat Index from 0 to MAXLEDS-1, для нас новый. Он называется “перечислимый цикл” и повторяет свое тело заданное количество раз, но каждый раз – с новым значением переменной Index. На первой итерации Index установлен в 0, на второй – в 1, и т.д. до последней итерации, на которой Index равна MAXLEDS-1, или 5. Это превосходный способ для настройки выполнения в пределах цикла, основанное на том, сколько раз будет выполняться цикл.

Следующая строка, if not LED[Index].Active, - это условное выражение, которое выполняет свое тело, если объект LED с индексом Index “не активен”. Поскольку внутренний цикл меняет значение Index с 0 по 5, то, по мере выполнения, это условное выражение вызывает метод Active каждого из наших объектов LED по очереди.

Если условие дает «истину» (объект LED по Index не активен), выполняется следующая строка - quit. Команда **QUIT** – это специальная команда, предназначенная только для циклов **REPEAT**; она вызывает мгновенный выход из цикла **REPEAT**, в рамках которого она содержится. Когда это происходит, выполнение продолжается с конца внешнего цикла **REPEAT**, условия **WHILE** .

Если все объекты **LED** активны, внутренний цикл будет повторяться с Index от 0 до 5, затем Index станет 6 (**MAXLEDS**) и произойдет выход из этого цикла, что приведет к

следующей итерации внешнего цикла и весь процесс повторится снова. Если, однако, будет найден неактивный объект LED, значение Index будет меньше MAXLEDS, и внешний цикл будет прерван, приведя к возвращению методом NextObject индекса Index доступного объекта. Это значение используется в Main для выбора необходимого метода LED для запуска.

За кулисами

В блоке объектов мы создали массив из шести объектов Output. Каждый из объектов, используемых приложением, должен рассматриваться как его собственность, со своими данными, хранящимися отдельно от таковых для других объектов. Поэтому, так как нам были необходимы возможности шести объектов Output, мы объявили потребность в шести таковых в блоке объектов.

После компиляции Blinker2, Вид Объекта отображает, что в нашем приложении имеется шесть экземпляров объекта Output; это видно по символам “[6]”, указанным справа от иконки объекта Output. Это способ, с помощью которого панель Вид Объекта отображает структуру приложения, показанную на Рис. 3-17.

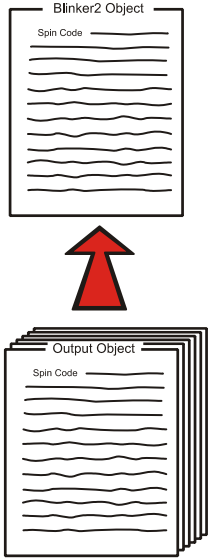


Рис. 3-17:
Приложение Blinker2

В приложении шесть экземпляров объектов Output. Приложение на самом деле использует лишь одну копию исполнимого кода объекта и шесть копий их областей глобальных переменных.

Значит ли это, что наше приложение увеличилось на величину шести размеров кода объекта Output? К счастью – нет. Программа *Propeller Tool* оптимизирует код приложения таким образом, что для всех экземпляров одного объекта включается лишь

одна копия кода, но создается несколько копий глобальных переменных. Это так, потому что код рассматривается как статический (неизменный) и абсолютно одинаковый для каждого экземпляра объекта. Однако, глобальные переменные объекта (определенные в его блоке VAR), не являются статическими; каждый экземпляр объекта требует свое собственное пространство переменных для обеспечения независимой работы без взаимного влияния экземпляров объекта друг на друга.

Окно Информации об Объекте

Мы можем увидеть этот эффект, используя свойство окна Информации об Объекте в *Propeller Tool*. Сначала измените блок OBJ в Blinker2, чтобы объявить лишь один экземпляр объекта Output - `LED[1] : "Output"`. Не запускайте код, сейчас он работать не будет, эти изменения мы делаем временно, для эксперимента.

Теперь нажмите клавишу *F8* (или выберите Run → Compile Current → View Info…) для компиляции приложения и отобразите окно Object Info. На Рис. 3-18 показано, как оно должно выглядеть.

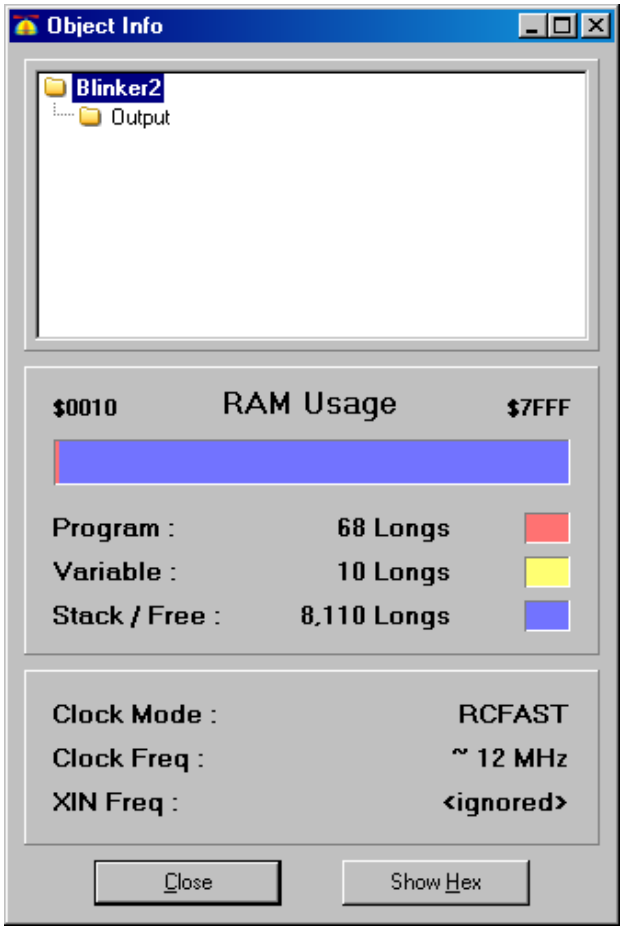


Рис. 3-18:
Окно Object Info для приложения Blinker2

Верхняя часть окна - это панель Info Object View, аналогичная Object View. Центральная часть окна отображает использование приложением ОЗУ. Отметьте, что сама программа “Program” (откомпилированный исходный код) использует 68 *long* ОЗУ, и переменные “Variable” (глобальные) требуют 10 *long* памяти.

Сейчас закройте это окно и измените блок `OBJ` в его прежнее состояние, объявляя шесть экземпляров объекта Output - `LED[6] : "Output"`. Откомпилируйте и посмотрите снова (F8 или Run → Compile Current → View Info…). Отметьте, что сейчас программа требует 73 *long* ОЗУ, а область переменных - 60 *long*. Программа увеличилась всего на 5 *long*, а область переменных – на 50 *long*. Дополнительная память под программу это издержки для возможности оперирования дополнительными объектами, но вот область

переменных увеличилась ровно в шесть раз по сравнению с предыдущей компиляцией; каждый экземпляр объекта имеет свою собственную область глобальных переменных. Наш объект Blinker2 не определяет сам ни одной глобальной переменной, но Output определяет девять *long* для стека `Stack` и один байт для переменной `Cog`, итого область из 10 *long*, поскольку область переменных выравнивается по размеру *long*.

Вы можете также кликнуть на объект Output в окне Object Info, чтобы увидеть, сколько места требует каждый из экземпляров этого объекта. Мы видим, что размер программы Output равен 21 *long*, а области переменных - 10 *long*.

Время жизни объекта

При компиляции приложений создается двоичный образ исполнимого кода. Этот двоичный образ – это собственно то, что загружается в ИМС Propeller, и именно его мы обычно имеем в виду, когда говорим “приложение ” или “приложение Propeller”.

Откомпилированный код каждого из используемых приложением объектов добавляется в этот двоичный образ вместе с областями переменных для каждого экземпляра каждого такого объекта.

Во время выполнения, приложение может использовать любой объект любое количество времени; одни могут использоваться постоянно, другие – по необходимости, но все они занимают неизменный объем памяти для своего кода и переменных.

Для разработчиков, привыкших к объектному программированию на компьютере, это - важная концепция для понимания. В ИМС Propeller время жизни объекта не меняется (static), не зависимо от того, активно ли он используется; он постоянно требует определенное количество памяти в двоичном образе приложения. В компьютерах объекты занимают изменяемый объем памяти (dynamic), поскольку они “создаются” и “уничтожаются ” при необходимости, во время выполнения приложения. В ИМС Propeller, объекты “создаются” во время компиляции и никогда не “создаются” и не “уничтожаются” во время выполнения, поскольку такие действия будут фрагментировать память, и приводить к неопределенностям в поведении встроенных систем реального времени.

Это значит, что каждый экземпляр объекта, который может понадобиться, должен быть объявлен до компиляции в блоке `OBJ`, так же, как мы делали в Упражнении 8 с массивом объектов Output.

Кратко: Упр. 8

- Приложения:
  - Используют уникальные идентификаторы или элементы массива для каждого отдельного используемого объекта.
  - Используют одну копию кода объекта и много копий его глобальных переменных.
- Объекты:
  - Массив объектов может быть создан в сегменте объектов так же, как массив переменных – в сегменте переменных.
  - Время жизни объекта не изменяется (static), объект занимает определенный неизменный объем памяти не зависимо от того, используется он, или нет. Это исключает возможность фрагментирования памяти при обычном использовании и обеспечивает однозначно устойчивое поведение систем реального времени.
- Язык *Spin*:
  - Команда `REPEAT`: (См. `REPEAT`, стр. 340).
    - Определенный, конечный цикл: `REPEAT` *Variable* `FROM` *Start* `TO` *Finish* где *Variable* – это переменная, используемая в качестве счетчика, а *Start* и *Finish* отображают диапазон.
    - Команда `QUIT` работает только внутри циклов `REPEAT` и приводит к мгновенному выходу из цикла, см. `QUIT`, стр. 338.
  - Регистры B/B (`DIR`×, `OUT`×, и `IN`×) могут использовать вид записи *reg*[*a*..*b*] для изменения нескольких смежных линий; здесь *reg* – это регистр (`DIR`×, `OUT`×, или `IN`×) а *a* и *b* – это номера линий B/B, см. `DIRA, DIRB` на стр. 249, `OUTA, OUTB` на стр. 326, и `INA, INB` на стр. 263.
- Линии B/B установлены как выходы только до тех пор, пока *Cog*, который выполнил эту настройку, активен, см. `DIRA, DIRB`, стр. 249.
- «Compile & View Info»: Клавиша *F8* (или выбрать Run → Compile Current → View Info…), см. «Информация об объекте (Object Info)», стр. 67.

Упражнение 9: Установки генератора

Внутренний генератор ИМС Propeller может работать на двух частотах, низкой (≈ 20 кГц) и высокой (≈ 12 МГц). Поскольку мы никогда не задавали каких-либо установок генератора в наших приложениях, все предыдущие упражнения использовали включенный по умолчанию внутренний RC-генератор в высокочастотном режиме.

Для указания установок генератора в приложении, в верхнем объектном файле необходимо задать одну или более специальных констант в сегменте **CON**. Этими константами являются: **_CLKMODE**, **_CLKFREQ** и **_XINFREQ**.

Начнем с **_CLKMODE**. В Табл. 4-3: Константы установки режимов генератора, на стр. 209, приведен перечень предустановленных идентификаторов для присваивания их значений константе **_CLKMODE**. Например, продолжая с нашим объектом Blinker2, изменение сегмента **CON**, как указано ниже, устанавливает режим генератора для работы внутреннего генератора на низкой частоте (показан только сегмент **CON**).

```
{{ Blinker2.spin }}

CON
  _CLKMODE = RCSLOW             'Set to internal slow clock
  MAXLEDS  = 6                  'Number of LED objects to use

<остальной код – без изменений>
```

Попробуйте откомпилировать и загрузить код Blinker2. Когда ИМС Propeller завершит процесс программирования и начальной загрузки, она переключит генератор в режим **RCSLOW** и запустит приложение. Поскольку приложение сейчас запущено с основной частотой, в сотни раз ниже, чем в предыдущих примерах, приложение будет выполняться намного медленнее, чем раньше, затрачивая более 20 секунд для первого переключения самого быстрого вывода, P20.

Вы можете заменить `_CLKMODE = RCSLOW` на `_CLKMODE = RCFAST`, чтобы приложение работало с внутренним генератором на высокой частоте (режим по умолчанию).

Если Вы хотите использовать внешний генератор, есть еще много опций для **_CLKMODE**. Допустим, Вы используете внешний резонатор на 5 МГц, такой, как в плате Propeller Demo Board.

Измените Ваш код в соответствии со следующим:

```
{{ Blinker2.spin }}

CON
  _CLKMODE = RCSLOW             'Set to internal slow clock
  _CLKMODE = XTAL1              'Set to ext. low-speed crystal
  _XINFREQ = 5_000_000          'Frequency on XIN pin is 5 MHz
  MAXLEDS  = 6                  'Number of LED objects to use

<остальной код – без изменений>
```

Здесь мы установили **_CLKMODE** в **XTAL1,** что конфигурирует генератор на работу с внешним низкочастотным резонатором и настраивает цепи усилителя внутреннего генератора для работы с кристаллом от 4 МГц до 16 МГц. Кроме самого кристалла (который должен быть подключен к выводам XI и XO), для данного режима работы генератора никакие внешние компоненты не нужны.

Когда бы ни использовались внешние резонаторы или генераторы, вдобавок к константе **_CLKMODE** необходимо задать либо **_XINFREQ**, либо **_CLKFREQ**. При этом константа **_XINFREQ** задает частоту на выводе XI (Вывод Входа Резонатора), а **_CLKFREQ** — Частоту Системного Генератора. Они обе зависят от установок ФАПЧ (PLL), которые мы обсудим позднее.

В этом примере мы задали значение **_XINFREQ** равное 5 миллионов; это значит, что частота на выводе XI равна 5 МГц, и резонатор, который мы подключили к выводам XI и XO — также на 5 МГц. При таких установках величина **_CLKFREQ** автоматически вычисляется и обновляется программой *Propeller Tool*.

Таким же образом Вы можете установить **_CLKFREQ** на 5 МГц (вместо **_XINFREQ**), и необходимое значение **_XINFREQ** также установится программой *Propeller Tool*. Однако принято задавать значение **_XINFREQ**, поскольку **_CLKFREQ** напрямую зависит от установок ФАПЧ (PLL). В нашем примере и **_XINFREQ**, и **_CLKFREQ** в конце имеют одно и то же значение, но следующий пример покажет, как они могут отличаться.

Если Вы сейчас откомпилируете и загрузите Blinker2, Вы увидите, что светодиоды мигают на частоте, немного меньшей, чем половина частоты в Упражнении 8. Это связано с тем, что мы настроили ИМС на работу с внешним 5 МГц кристаллом, вместо внутреннего 12 МГц генератора.

Зачем же кому-либо использовать внешний резонатор, который медленнее, чем внутренний генератор? Две причины: 1) Бо́льшая точность – внутренний генератор не очень стабилен, его частота различна от микросхемы к микросхеме и изменяется в

диапазоне изменения напряжения питания, а внешние резонаторы либо генераторы обычно весьма стабильны, и 2) цепи фазовой автоподстройки частоты ФАПЧ (PLL) можно использовать только с внешними источниками.

Попробуйте следующий пример:

```
{{ Blinker2.spin }}

CON
  _CLKMODE = XTAL1 + PLL4X      'Set to ext low-speed crystal, 4x PLL
  _XINFREQ = 5_000_000          'Frequency on XIN pin is 5 MHz
  MAXLEDS  = 6                  'Number of LED objects to use

<остальной код – без изменений>
```

Здесь мы немного изменили настройку **_CLKMODE**, добавив величину + PLL4X. Таким образом, мы конфигурируем режим генератора для использования внутренней ФАПЧ (PLL) для разгона частоты XIN в четыре раза, получая в результате Системную Частоту, равную 5 МГц * 4 = 20 МГц.

Попытайтесь откомпилировать и загрузить Blinker2 с этим настройками. Вы увидите мигание светодиодов с большей скоростью, чем Вы видели ранее.

Примечание: Поскольку здесь мы установили значение **_XINFREQ**, то значение **_CLKFREQ** автоматически рассчиталось для 20 МГц. Если бы мы вместо этого задали величину **_CLKFREQ** равную 5 МГц, добавление опции **PLL4X** подсчитало бы значение **_XINFREQ** 1.25 МГц, что не совпадает с частотой нашего внешнего резонатора. Вот почему обычно правильнее задавать частоту на XIN (**_XINFREQ**), вместо частоты генератора (**_CLKFREQ**).

Если цепь ФАПЧ включена, она всегда умножает входную частоту в 16 раз, но в качестве источника Системной Частоты Вы можете выбрать любой из её отводов на 1x, 2x, 4x, 8x либо 16x, используя установки **PLL1X**, **PLL2X**, **PLL4X**, **PLL8X** и **PLL16X**.

Попробуйте изменить **_CLKMODE** с XTAL1 + PLL4x на XTAL1 + PLL16x и загрузить вновь. Это настроит Системную Частоту на 5 МГц* 16 = 80 МГц! Большинство светодиодов мигает так быстро, что кажется, что они постоянно включены.

Упражнение 10: Временные соотношения

Последнее упражнение, возможно, помогло Вам кое-что осознать: наш объект Output подвержен влиянию величины тактовой частоты. Он имеет свою, жестко заданную временную базу, а для подчиняемых объектов (тех, которые не являются верхними

объектными файлами), это недопустимо, поскольку они никогда не могут предсказать, какие тактовые частоты будут использоваться во всем множестве их применений. Тем более что при своем выполнении приложение Propeller само может много раз менять величину тактовой частоты.

Предположим, нам необходимо создать объект Output, который переключает выводы на заданной частоте, которая не зависит от тактовой частоты. Это значит, что объект должен динамично реагировать на изменение значения Системной Частоты. Отредактируйте свой код в соответствии с приведенным ниже.

Example Object: Output.spin

```
{{ Output.spin }}

VAR
  long  Stack[9]                        'Stack space for new Cog
  byte  Cog                             'Hold ID of Cog in use, if any


PUB Start(Pin, DelayMS, Count): Success
{{Start new blinking process in new Cog; return True if successful.}}

  Stop
  Success := (Cog := Cognew(Toggle(Pin, DelayMS, Count), @Stack) + 1)


PUB Stop
{{Stop toggling process, if any.}}

  if Cog
    Cogstop(Cog~ - 1)


PUB Active: YesNo
{{Return TRUE if process is active, FALSE otherwise.}}

  YesNo := Cog > 0


PUB Toggle(Pin, DelayMS, Count)
{{Toggle Pin, Count times with DelayMS milliseconds ~~clock cycles~~
  in between.  If Count = 0, toggle Pin forever.}}

  dira[Pin]~~                                   'Set I/O pin to output…
  repeat                                        'Repeat the following
    !outa[Pin]                                  '  Toggle I/O Pin
    waitcnt(clkfreq / 1000 * DelayMS + cnt)     '  Wait for DelayMS…
  while Count := --Count #> -1                  'While not 0 (make min…
  Cog~                                          'Clear Cog ID variable
```

Мы модифицировали методы Start и Toggle, изменив параметр Delay на DelayMS, что значит “задержка в единицах миллисекунд”. Затем мы доработали оператор waitcnt… таким образом, чтобы он вместо ожидания фиксированного количества циклов, вычислял количество таких циклов в заданном параметре DelayMS миллисекунд времени. CLKFREQ - это команда, которая возвращает текущее значение Системной Частоты, в герцах (циклов в секунду). Это значение устанавливается программой *Propeller Tool* во время компиляции, а так же командой CLKSET во время выполнения; см. CLKSET на стр. 213. В секунде 1 000 миллисекунд, а CLKFREQ – это количество циклов частоты в секунду, поэтому clkfreq / 1000 * DelayMS – это количество циклов частоты в промежутке времени DelayMS миллисекунд.

С такой доработкой, независимо от частоты, на которой стартовало приложение, или от того, как часто приложение меняет частоту во время выполнения, объект Output будет пересчитывать правильную задержку при выполнении каждого своего цикла.

Теперь, конечно, нам необходимо изменить наш объект Blinker2 для правильной настройки параметров DelayMS. Добавьте изменения в код, как показано в листинге на стр. 159. Отметьте, что мы используем установки _CLKMODE и _XINFREQ лишь потому, что они остались у нас из предыдущего упражнения.

Example Object: Blinker2.spin

```
{{ Blinker2.spin }}

CON
  _CLKMODE = XTAL1 + PLL4X      'Set to ext low-speed crystal, 4x PLL
  _XINFREQ = 5_000_000          'Frequency on XIN pin is 5 MHz
  MAXLEDS = 6                   'Number of LED objects to use


OBJ
  LED[6] : "Output"


PUB Main
{Toggle pins at different rates, simultaneously}

  dira[16..23]~~                                'Set pins to outputs
  LED[NextObject].Start(16, 250,   0)           'Blink LEDs
  LED[NextObject].Start(17, 500,   0)
  LED[NextObject].Start(18,  50, 300)
  LED[NextObject].Start(19, 500,  40)
  LED[NextObject].Start(20,  29, 300)
  LED[NextObject].Start(21, 104, 250)
  LED[NextObject].Start(22,  63, 200)           '<-Postponed
  LED[NextObject].Start(23,  33, 160)           '<-Postponed
  LED[0].Start(20, 1000, 0)                     'Restart object 0
  repeat                                        'Loop endlessly

PUB NextObject : Index
{Scan LED objects and return index of next available LED object.
 Scanning continues until one is available.}

  repeat
    repeat Index from 0 to MAXLEDS-1
      if not LED[Index].Active
        quit
  while Index == MAXLEDS
```

В функции Main у всех вызовов Start мы изменили второй параметр с “задержка в циклах” на параметр “задержка в миллисекундах”. Теперь откомпилируйте и загрузите объект Blinker2. Отметьте, что частоты, с которыми мигает каждый светодиод, такие же, как и в упражнении, когда мы использовали внутренний высокочастотный генератор. Попробуйте увеличить тактовую частоту изменением _CLKMODE с XTAL1 + PLL4X на XTAL1 + PLL16X. Вы не должны увидеть никаких изменений в частотах мигания, даже притом, что мы увеличили тактовую частоту в четыре раза!

Имейте в виду, что точность внутреннего генератора Вашего конкретного образца ИМС Propeller может оказывать большое влияние на работу этого примера, особенно если использовать режим RCSLOW.

Есть два подхода в использовании команды WAITCNT, но мы показали только один из них. Для дальнейших разъяснений касательно временных отношений, см. WAITCNT на стр. 373.

Кратко: Упр. 9 и 10

- Генератор:
  - Внутренний генератор: медленный (≈20 кГц) либо быстрый (≈12 МГц).
  - Для настройки генератора, в верхнем файле задаем значения для одной или более специальных констант: _CLKMODE, _CLKFREQ или _XINFREQ.
  - Всегда, когда бы ни использовался внешний резонатор или частота, вдобавок к _CLKMODE необходимо задать либо _XINFREQ, либо _CLKFREQ.
  - _CLKMODE задает режим генератора: внутренний/внешний, усиление, установки ФАПЧ (PLL) и т.д. См. _CLKMODE, стр. 209.
  - _XINFREQ задает частоту, приходящую на вход XI (Вход Резонатора). См. _XINFREQ, стр. 392.
  - _CLKFREQ задает тактовую Системную Частоту. См. _CLKFREQ, стр. 206.
  - В приложениях, не требующих точность и стабильность, удобно использовать внутренний генератор. Используйте внешний источник, если необходима точность, стабильность, либо ФАПЧ (PLL).
- Тайминг:
  - Подчиненные объекты не должны использовать жестко заданной временной базы, поскольку приложения, которые их используют, могут менять тактовую частоту.
  - Для вычислений времени используйте команду CLKFREQ, возвращающую текущее значение Системной Частоты в герцах. См. CLKFREQ, стр. 204.

Упражнение 11: Библиотечные Объекты

Программа *Propeller Tool* поставляется с библиотекой объектов, созданной инженерами компании Parallax. Эти объекты выполняют множество полезных функций, таких как связь по последовательному каналу, математика с числами с плавающей запятой, преобразование число-строка и строка-число, генерация сигналов для TV-дисплея, подключение стандартных компьютерных клавиатуры, мыши, монитора и т.д.

Библиотека объектов Propeller – это просто папка, содержащая файлы объектов Propeller, которые автоматически создаются во время инсталляции пакета *Propeller Tool*. Вы можете попасть в папку библиотеки Propeller, выбрав “Propeller Library” из списка последних открытых файлов; см. Рис. 3-19. После выбора библиотеки Propeller список файлов отобразит все доступные объекты.

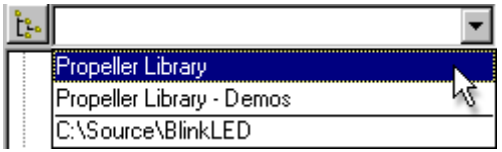


Рис. 3-19:
Просмотр библиотеки Propeller

Выберите “Propeller Library”из списка последних открытых файлов встроенного браузера, чтобы быстро попасть в папку библиотеки.

Давайте попробуем применить некоторые из них. Создайте новый файл и введите код, приведенный ниже. Подсвеченные элементы важны для дальнейшего обсуждения.

Example Object: Display.spin

```
{{ Display.spin }}

CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

OBJ
  Num   :       "Numbers"
  TV    :       "TV_Terminal"

PUB Main | Temp
  Num.Init                                      'Initialize Numbers
  TV.Start(12)                                  'Start TV Terminal

  Temp := 900 * 45 + 401                        'Evaluate expression
  TV.Str(string("900 * 45 + 401 = "))           'then display it and
  TV.Str(Num.ToStr(Temp, Num#DDEC))             'its result in decimal
  TV.Out(13)
  TV.Str(string("In hexadecimal it's  = "))     'and in hexadecimal
  TV.Str(Num.ToStr(Temp, Num#IHEX))
  TV.Out(13)
  TV.Out(13)

  TV.Str(string("Counting by fives:"))          'Now count by fives
  TV.Out(13)
  repeat Temp from 5 to 30 step 5
    TV.Str(Num.ToStr(Temp, Num#DEC))
    if Temp < 30
      TV.Out(",")
```

Сохраните этот объект как “Display.spin” в папке по Вашему выбору; в этом примере мы будем использовать папку “C:\Source\”.

В данном примере мы используем два объекта из библиотеки Propeller Library - Numbers и TV_Terminal, для преобразования численных значений в строки и отображения их на ТВ-дисплее. Откомпилируйте и загрузите этот пример объекта и подключите TV -дисплей (NTSC) к сигнальному выходу (джек RCA) на плате Propeller Demo Board. Дисплей должен показать следующий текст:

```
900 * 45 + 401 = 40,901
In hexadecimal it's = $9FC5
Counting by fives:
 5, 10, 15, 20, 25, 30
```

Посмотрите, чего мы добились! Используя всего несколько строк своего собственного кода, плюс два существующих библиотечных объекта и три резистора (на плате Propeller Demo Board), мы перевели численные величины в текстовые строки и синтезировали TV-совместимый сигнал, чтобы отобразить этот сигнал в реальном времени на стандартном TV! На самом деле, когда Вы читаете отображенную информацию, *Cog* постоянно занят генерацией сигнала NTSC с частотой 60 кадров в секунду, который может отобразить TV-дисплей.

Объект TV_Terminal предоставляет великолепный дисплей для задач отладки. Поскольку ИМС Propeller имеет много процессоров и довольно высокую производительность, то экран реального времени, такой как TV-монитор (CRT или LCD), используемый для отладки приложений, позволяет значительно упростить процесс создания оптимального кода. Мы рекомендуем использовать этот прием наряду с классическими для сокращения времени разработки.

Теперь давайте посмотрим на некоторые важные части нашего кода. Первый новый элемент в нашем коде – это `| Temp` , который мы видим в объявлении метода Main. Не ошибитесь – это выглядит как объявление переменной возврата, но на самом деле это не так. Символ вертикальной линии '|' означает, что далее мы объявляем локальные переменные. Так что `| Temp` означает, что `Temp` – это локальная для `Main` переменная размером *long*.

Далее мы имеем два очень важных оператора, `Num.Init` и `TV.Start(12)`. Эти два оператора инициализируют объект Numbers и запускают объект TV_Terminal (на выводах 12, 13 и 14) соответственно. Каждый из этих объектов требует некоторой инициализации перед их использованием. Объекту Numbers необходимо, чтобы был вызван его метод `Init` для инициализации некоторых внутренних регистров. Объекту TV_Terminal необходимо, чтобы был запущен его метод `Start` для задания необходимых выходов и запуска еще двух процессоров для генерации сигналов дисплея. Обычно такие требования указываются в документации на каждый объект, и обычно они включают методы `Init` или `Start`, если им необходима некоторая настройка перед использованием

Следующая строка выполняет некоторые арифметические действия и присваивает нашей локальной переменной `Temp` результат. Вскоре мы его будем использовать.

Следующие три оператора создают первую строку на ТВ-дисплее: `9 * 45 + 401 = 40,901`. Метод `TV.Str` выводит zero-terminated строку на дисплей. Его параметр, `string('900 * 45 + 401 = ')` для нас новый. **`STRING`** - это директива, которая создает *zero-terminated* строку символов (множество байтов данных, сопровождающихся нолем, иногда ее еще называют *z-string*) и возвращает адрес этой строки. Большинство методов, работающих со строками, требуют лишь адрес первого символа и чтобы строка заканчивалась байтом, равным нолю. Параметр метода `TV.Str` требует именно этого, адрес *zero-terminated* строки. Так что строка `TV.Str(string('900 * 45 + 401 = '))` приводит к отображению строки "900 * 45 + 401 = " на экране TV.

Следующий оператор, `TV.Str(Num.ToStr(Temp, Num#DDEC))` печатает такую часть строки: "40,901". Метод `Num.ToStr` преобразует численное значение из `Temp` в строку, используя формат с разделителями`,` и возвращает адрес этой строки. Конечно же, `Temp` содержит результат нашего более раннего выражения: 40901, размером *long*. Часть `Num#DDEC`, однако, для нас нова. Символ **`#`** , когда используется подобным образом, - это ссылка Объект-Константа (Object-Constant reference); она используется для обращения к константам, объявленным в других объектах. В этом случае, `Num#DDEC` ссылается на "константу формата" `DDEC`, которая объявлена в объекте Numbers. Как описано в Numbers, `DDEC` отвечает за десятичное разделение и содержит величину, которая показывает методу `ToStr` , что он должен форматировать число с разделением групп тысяч с разделителем тысяч – в этом случае запятой. Итак, `ToStr` создает *z-string,* содержащую "40,901" и возвращает ее адрес. Затем `TV.Str` выводит эту строку на дисплей. Прочтите Документацию на объект Numbers для более подробной информации об этой и других константах форматирования.

`TV.Out(13)` выводит один байт, 13, на дисплей. Символ 13 – это ASCII код возврата каретки (невидимый символ), который заставляет объект TV_Terminal переходить на следующую текстовую строку. Делается это в качестве подготовки для вывода следующей строки, которая будет напечатана далее.

Рабочие и Библиотечные папки

При компиляции объекта Display, панель Object View показывает изображенную ниже структуру. Она говорит нам, что наш объект Display использует объекты Numbers и TV_Terminal, а объект TV_Terminal использует объекты TV и Graphics.

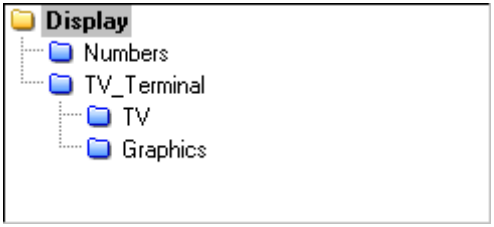


Рис. 3-20:
Панель Object View приложения Display

Желтые папки означают объект в «рабочей» папке. Синие папки означают объект в папке «библиотеки».

Иконки папок перед каждым объектом имеют различные цвета для отображения местоположения каждого из них. Желтые папки означают объект в «рабочей» папке. Синие папки означают объект в папке «библиотек». Из данных этой панели мы можем сделать вывод, что программа *Propeller Tool* нашла объекты Numbers, TV_Terminal, TV и Graphics в библиотечной папке, а объект Display – в рабочей папке.

Вы помните, что мы сохранили наш объект Display в папке C:\Source? Когда приложение компилируется, папка, в которой сохранен верхний объектный файл, становится рабочей папкой. Если этот файл ссылается на другие объекты, то рабочая папка – это первое место, в котором программа *Propeller Tool* их ищет. Если запрашиваемый объект не находится в рабочей папке, следующая папка, в которой производится поиск – это папка библиотеки. Если объект в библиотечной папке ссылается на другой объект, он ищется в библиотечной папке. Если необходимые объекты не найдены ни в одной из этих папок, возникает ошибка.

Исходя из такого принципа, можно сказать, что любое приложение полностью состоит из файлов, взятых максимум из двух папок: рабочей и/или библиотечной. Учитывайте это при построении своих собственных приложений.

Вы можете определить расположение каждого объекта, а так же рабочей и библиотечной папок, указав мышью на каждый из объектов в панели Object View. На рисунках ниже мы видим, что Display находится в C:\Source (“рабочая” папка), а Numbers – в C:\Program Files\Parallax Inc\Propeller Tool (“библиотечная” папка).

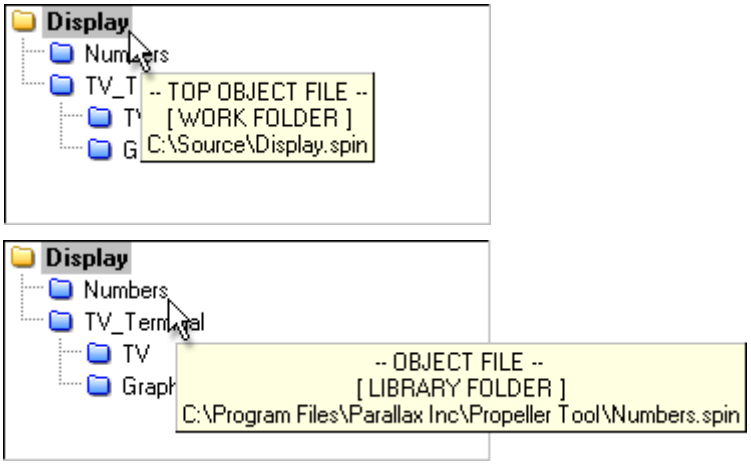


Рис. 3-21:

Подсказки в панели Object View для приложения Display

Пути к рабочей и библиотечной папкам можно увидеть в сообщениях подсказок

Упражнение 12: Целые и вещественные числа

ИМС Propeller – это 32-х разрядный контроллер, для которого «родной» формат, как в константах, так и в математических выражениях – это целые числа со знаком (от -2 147 483 648 до 2 147 483 647). Однако, для работы с вещественными числами (с целой и дробной частью) компилятор поддерживает формат с плавающей точкой (с одинарной точностью, согласно IEEE-754), для констант напрямую, а для выполнимых математических операций – при помощи специальных библиотечных объектов.

Псевдо-вещественные числа

Существует много возможных способов реализации операций с вещественными числами. Один из способов – это использовать целочисленную математику таким путем, который приспосабливает вещественные величины и выполнимые выражения к целочисленной математике. Мы называем такие числа псевдо-реальными.

Обладая 32-х разрядной архитектурой, ИМС Propeller обеспечивает нас большим «пространством для разворота» – динамическим диапазоном – для вычислений. Например, пусть у нас есть выражение, в котором необходимо умножать и делить десятичные величины, которые имеют 2-х разрядную дробную часть, то есть следующее:

A = B * C / D

Пусть в нашем примере будет A = 7.6 * 38.75 / 12.5, что в результате даст 23.56.

Для вычисления этого выражения во время выполнения, мы можем сдвинуть десятичную запятую вправо на 2 знака, чтобы сделать все числа целыми, выполнить целочисленные вычисления, а затем просто считать последние две цифры результата дробной частью. Мы добились этого, умножив каждое число на 100. Вот доказательство:

A = (B* 100) * (C * 100) / (D * 100)

A = (7.6 * 100) * (38.75 * 100) / (12.5 * 100)

A = 760 * 3875 / 1250

A = 2356

Поскольку мы умножили все исходные значения на 100, мы понимаем, что на самом деле результат будет 2356 / 100 = 23.56, но для большинства задач мы можем просто оставить его целым числом, имея в виду, что правые две цифры – это дробная часть.

Приведенное выше решение работает до тех пор, пока каждая из исходных величин и каждое из промежуточных значений не выходят за границы диапазона от -2 147 483 648 до 2 147 483 647.

В следующем примере приведен код, использующий как операции с псевдо-вещественными числами, так и с числами с плавающей запятой.

Формат с плавающей запятой

Выражения, включающие вещественные числа, о многих случаях могут быть решены без использования величин в формате с плавающей точкой и методов работы с ними; так происходит, например, при использовании метода с псевдо-вещественным представлением чисел. Поскольку решения, подобные приведенному выше, имеют тенденцию выполняться намного быстрее и требовать намного меньше памяти, то перед тем, как использовать поддержку чисел в формате с плавающей запятой, рекомендуем Вам серьезно подумать – нужно ли это в действительности делать. Если же у Вас нет дефицита в памяти и времени выполнения, поддержка чисел с плавающей запятой может представлять собой наилучшее решение.

Программа *Propeller Tool* имеет прямую поддержку констант в формате с плавающей точкой. В то же время ИМС Propeller поддерживает операции с числами в формате с плавающей точкой лишь путем использования объектов; это значит, что во время выполнения Интерпретатор языка *Spin* может оперировать только с целочисленными выражениями.

Объект RealNumbers.spin, в следующем примере, демонстрирует использование целочисленных констант (`iB`, `iC`, и `iD`), приведенных к псевдо-вещественным числам; констант с плавающей точкой (`B`, `C`, и `D`), используемых в их первоначальном виде через библиотечные объекты FloatMath и FloatString; а также этих же констант, приводимых к псевдо-вещественному представлению во время компиляции.

Example Object: RealNumbers.spin

```
{{ RealNumbers.spin}}
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  iB    =  760                  'Integer constants
  iC    = 3875
  iD    = 1250

  B     =  7.6                  'Floating-point constants
  C     = 38.75
  D     = 12.5

  K     = 100.0                 'Real-to-Pseudo-Real multiplier

OBJ
  Term  :       "TV_Terminal"
  F     :       "FloatMath"
  FS    :       "FloatString"

PUB Math
  Term.Start(12)

  {Integer constants (real numbers * 100) to do fast integer math}
  Term.Str(string("Pseudo-Real Number Result: "))
  Term.Dec(iB*iC/iD)

  {Floating-point constants using FloatMath and FloatString objects}
  Term.Out(13)
  Term.Str(string("Floating-Point Number Result: "))
  Term.Str(FS.FloatToString(F.FDiv(F.FMul(B, C), D)))

  {Floating-point constants translated to pseudo-real for fast math}
  Term.Out(13)
  Term.Str(string("Another Pseudo-Real Number Result: "))
  Term.Dec(trunc(B*K)*trunc(C*K)/trunc(D*K))
```

Откомпилируйте и загрузите RealNumbers.spin. На TV-дисплее будет отображено следующее:

```
Pseudo-Real Number Result: 2356
Floating-Point Number Result: 23.56
Another Pseudo-Real Number Result: 2356
```

Каждый из псевдо-вещественных результатов, конечно, представляет одно и то же значение 23.56, но вся величина смещена на два порядка вверх для обеспечения совместимости с целочисленной математикой. С некоторой доработкой кода мы могли бы вывести его на дисплей в корректной форме, «23.56».

Как мы уже увидели ранее, константы `iB`, `iC`, и `iD` – это стандартные целые константы, но их значения в действительности являются псевдо-вещественными числами, представляющими значения нашего выражения из примера.

Константы `B`, `C`, `D`, и `K` – это константы в формате с плавающей точкой (вещественные числа). Компилятор автоматически распознает их как таковые, и сохраняет их в 32-х битном формате с плавающей точкой, одинарной точности. Они могут быть напрямую использованы в других выражениях, решаемых при компиляции, однако в процессе выполнения приложения они могут быть использованы только с помощью специальных методов работы, таких как в объектах FloatMath и FloatString.

Оператор `Term.Dec(iB*iC/iD)` использует приведенные псевдо-реальные константы как принято в способе Псевдо-Вещественных Чисел, выше. Он выполняется примерно в 1.6 раз быстрее, чем по способу плавающей точки и требует намного меньше кода.

Оператор `Term.Str(FS.FloatToString(F.FDiv(F.FMul(B, C), D)))` вызывает метод `FMul` объекта FloatMath для умножения чисел с плавающей точкой `B` и `C`, затем вызывает метод `FDiv` объекта FloatMath для деления результата на число с плавающей точкой `D`, преобразовывает результат в строку, используя метод `FloatToString` объекта FloatString, и отображает его на TV.

Оператор `Term.Dec(trunc(B*K)*trunc(C*K)/trunc(D*K))` использует выражения, рассчитываемые при компиляции внутри директив `TRUNC` для смещения констант с плавающей точкой `B`, `C`, и `D` вверх на два порядка и отсечения их до целочисленных величин. Результирующее выражение эквивалентно таковому в первом случае представления псевдо-вещественных значений `Term.Dec(iB*iC/iD)`, но имеет то преимущество, что оно позволяет значениям своих компонентов быть числами в формате с плавающей точкой.

`TRUNC` во время компиляции отсекает результат выражения до целого значения, – константы с плавающей точкой не могут быть использованы в run-time выражениях.

Контекстно-зависимая информация компиляции

После того, как объект был откомпилирован, программа *Propeller Tool* в строке статуса (панель 5) отображает контекстно-зависимую информацию о компиляции элемента исходного кода, возле или внутри которого сейчас находится курсор. Это очень полезно при проверке и анализе значений констант, объявленных в объекте. Например, откомпилируйте наш объект, нажав *F9* (или выбрав в меню Run → Compile Current → Update Status), а затем поместите курсор на константу `iB` в блоке `CON`. Строка статуса временно подсветит контекстную информацию и должна выглядеть так:

Рис. 3-22:
Строка статуса с информацией о компиляции

После компиляции, строка состояния (панель 5)отображает информацию о ближнем к курсору элементе кода.

Это говорит нам, что наша константа `iB` определена в блоке `CON` как десятичное 760, или шестнадцатеричное $2F8.

Поместите курсор на константу `B`. Информация теперь должна сообщать “CON B = 7.6 ($40F3_3333) Floating Point”,т.е. что это вещественное число, в форме с плавающей точкой, равное 7.6 в десятичной и $40F3_333 — в шестнадцатеричной форме. Это подтверждает, что величины с плавающей точкой кодируются в формате, не совместимом с форматом целых чисел.

Вдобавок к отображению значений идентификаторов в блоках `CON` и `DAT`, в панели информации о компиляции отображается размер (в байтах) блоков `PUB`/`PRI`/`DAT`, когда курсор находится внутри каждого из блоков. В нашем случае, метод `Math` имеет размер 196 байтов. Это очень полезное свойство, помогающее при оптимизации размера кода - сделали небольшие изменения в коде, нажали F9, проверили размер в сравнении с предыдущим разом, и так далее.

Кратко: Упр.11 и 12

- Библиотека Propeller:
  - Это папка, автоматически созданная инсталлятором *Propeller Tool*.
  - Содержит созданные в Parallax объекты, с полезными функциями.
  - Для быстрого доступа используется пункт “Propeller Library” в списке последних открытых папок.
- Язык *Spin*:
  - Символ ‘|’ в строках объявления метода объявляет локальные переменные метода; см. Параметры и Локальные переменные, стр. 336.
  - Директива `STRING` создает *zero-terminated* сроку и возвращает ее адрес; см. `STRING`, стр. 358.
  - Символ `#` создает ссылку объект-константа для доступа к константам, определенным в других объектах; см. Область видимости , стр. 234.
  - Директива `TRUNC` отсекает константы с плавающей точкой в целые; см. `TRUNC`, стр. 363.
- Рабочие и библиотечные папки:
  - Иконки папок в панели Object View показывают, где находится объект.
    - С желтыми папками – объекты в «рабочей» папке.
    - С синими папками – объекты в «библиотечной» папке.
  - Каждое приложение полностью состоит из файлов, расположенных максимум в двух папках: рабочей папки и/или библиотечной папки.
- Целые и вещественные числа: (См. `CON`, стр.228, или «Операторы Spin»,стр.291)
  - Целые числа напрямую поддерживаются как в константах, так и в процессе выполнения приложения.
  - Вещественные числа, в формате с плавающей точкой, поддерживаются напрямую в константах, и косвенно – в run-time, при использовании специальных библиотечных объектов.
  - Во многих случаях, выражения с вещественными числами могут быть решены без использования арифметики с плавающей точкой.
- Строка статуса отображает информацию компиляции об элементе исходного кода рядом с курсором, размер/адрес блока `CON`/`DAT` и размер блока `PUB`/`PRI`/`DAT`.

Что делать далее...

Вы уже должны накопить знания, достаточные для самостоятельного использования ИМС Propeller. Используйте остальную часть руководства как справочник по языкам *Spin* и ассемблер, изучите каждый интересный Вам библиотечный объект, и участвуйте в форуме Propeller для продолжения обучения и общения с другими активными пользователями ИМС Propeller!

Глава 4: Справочник по языку Spin

Эта глава описывает все элементы языка *Spin* для ИМС Propeller и рассчитана на использование в качестве справочного пособия. Предполагается, что Вы уже ознакомились с материалом, который представляет Глава 3: Программирование ИМС Propeller . Совместно с программой *Propeller Tool* поставляются англоязычные файлы онлайн-помощи, отражающие материалы Главы 2 и Главы 3, и в которых учитываются все последние изменения и доработки, периодически вносимые в программное обеспечение.

Справочник по языку *Spin* состоит из трех секций:

1) **Структура Объектов Propeller**. Объекты Propeller состоят из кода *Spin* и, возможно, кода ассемблера и данных. Структура объекта формируется кодом *Spin* при помощи специальных блоков. В этой секции приводятся такие блоки, а также элементы, которые могут использоваться в каждом из них. Каждый перечисленный элемент имеет справочную страницу с подробной информацией.

2) **Перечень элементов языка Propeller *Spin* по категориям**. Все элементы, включая операторы и символы синтаксиса, группируются в соответствии с выполняемыми функциями. Это удобный способ для быстрого осознания возможностей языка и того, какие функции необходимы для конкретной задачи. Каждый перечисленный элемент имеет справочную страницу с подробной информацией. Некоторые элементы обозначены индексом “a”, означающим, что они также доступны в ассемблере, хотя их синтаксис может и отличаться. Отмеченные таким образом элементы включены также и в Главу 5: Справочник по языку ассемблера.

3) **Элементы языка *Spin***. Большинство элементов имеют собственные, отведенные им, подсекции, сгруппированные по алфавиту для облегчения поиска. Элементы, которым не отведено подсекции, – такие как Операторы, Идентификаторы и некоторые константы, – сгруппированы в рамках групп, организованных по другому признаку, но все так же могут быть легко найдены путем поиска их справочной страницы из перечня по категориям.

Структура Объектов Propeller

Каждый объект Propeller имеет внутреннюю структуру, включающую до шести специальных блоков: CON, VAR, OBJ, PUB, PRI, и DAT. Эти блоки показаны ниже (в порядке, в котором они обычно приводятся в объектах), совместно с элементами, обычно используемыми с каждым из них.

Для подробных примеров структур объектов и их использования см. Глава 3: Программирование ИМС Propeller , которая начинается со стр. 101.

CON: Блоки констант определяют глобальные константы (стр. 228).

_CLKFREQ	стр. 206	NEGX	стр. 237	PLL16X	стр. 209	XINPUT	стр. 209
_CLKMODE	стр. 209	**Операторы***	стр. 291	POSX	стр. 237	XTAL1	стр. 209
_FREE	стр. 255	PI	стр. 237	RCFAST	стр. 209	XTAL2	стр. 209
_STACK	стр. 355	PLL1X	стр. 209	RCSLOW	стр. 209	XTAL3	стр. 209
_XINFREQ	стр. 392	PLL2X	стр. 209	ROUND	стр. 351		
FALSE	стр. 237	PLL4X	стр. 209	TRUE	стр. 237		
FLOAT	стр. 253	PLL8X	стр. 209	TRUNC	стр. 363		

* **Кроме операторов присваивания.**

VAR: Блоки переменных, определяют глобальные переменные (стр. 364).

BYTE	стр. 192	LONG	стр. 274	ROUND	стр. 351	TRUNC	стр. 363
FLOAT	стр. 253	**Операторы***	стр. 291	WORD	стр. 382		

* **Кроме операторов присваивания.**

OBJ: Блоки объектов, определяют используемые объекты (стр. 288).

FLOAT	стр. 253	**Операторы***	стр. 291	ROUND	стр. 351	TRUNC	стр. 363

* **Кроме операторов присваивания.**

PUB/PRI: Блоки методов *Public* и *Private* определяют процедуры Spin (стр. 334/333).

ABORT	стр. 187	FLOAT	стр. 253	**Операторы**	стр. 291	ROUND	стр. 351
BYTE	стр. 192	FRQA	стр. 256	OUTA	стр. 326	SPR	стр. 353
BYTEFILL	стр. 198	FRQB	стр. 256	OUTB	стр. 326	STRCOMP	стр. 356
BYTEMOVE	стр. 199	IF	стр. 257	PAR	стр. 330	STRING	стр. 358
CASE	стр. 200	IFNOT	стр. 263	PHSA	стр. 332	STRSIZE	стр. 359
CHIPVER	стр. 203	INA	стр. 263	PHSB	стр. 332	TRUE	стр. 237
CLKFREQ	стр. 204	INB	стр. 263	PI	стр. 237	TRUNC	стр. 363
CLKMODE	стр. 208	LOCKCLR	стр. 266	PLL1X	стр. 209	VCFG	стр. 368
CLKSET	стр. 213	LOCKNEW	стр. 268	PLL2X	стр. 209	VSCL	стр. 371
CNT	стр. 215	LOCKRET	стр. 271	PLL4X	стр. 209	WAITCNT	стр. 373
COGID	стр. 217	LOCKSET	стр. 272	PLL8X	стр. 209	WAITPEQ	стр. 377
COGINIT	стр. 218	LONG	стр. 274	PLL16X	стр. 209	WAITPNE	стр. 379
COGNEW	стр. 221	LONGFILL	стр. 281	POSX	стр. 237	WAITVID	стр. 380
COGSTOP	стр. 227	LONGMOVE	стр. 282	QUIT	стр. 338	WORD	стр. 382
CONSTANT	стр. 235	LOOKDOWN	стр. 283	RCFAST	стр. 209	WORDFILL	стр. 390
CTRA	стр. 239	LOOKDOWNZ	стр. 283	RCSLOW	стр. 209	WORDMOVE	стр. 391
CTRB	стр. 239	LOOKUP	стр. 285	REBOOT	стр. 339	XINPUT	стр. 209
DIRA	стр. 249	LOOKUPZ	стр. 285	REPEAT	стр. 340	XTAL1	стр. 209
DIRB	стр. 249	NEGX	стр. 237	RESULT	стр. 347	XTAL2	стр. 209
FALSE	стр. 237	NEXT	стр. 287	RETURN	стр. 349	XTAL3	стр. 209

DAT: Блоки данных определяют данные и код ассемблера (стр. 243).

Assembly	стр. 394	FRQB	стр. 256	PI	стр. 237	TRUNC	стр. 363
BYTE	стр. 192	INA	стр. 263	PLL1X	стр. 209	VCFG	стр. 368
CNT	стр. 215	INB	стр. 263	PLL2X	стр. 209	VSCL	стр. 371
CTRA	стр. 239	LONG	стр. 274	PLL4X	стр. 209	WORD	стр. 382
CTRB	стр. 239	NEGX	стр. 237	PLL8X	стр. 209	XINPUT	стр. 209
DIRA	стр. 249	**Операторы***	стр. 291	PLL16X	стр. 209	XTAL1	стр. 209
DIRB	стр. 249	OUTA	стр. 326	POSX	стр. 237	XTAL2	стр. 209
FALSE	стр. 237	OUTB	стр. 326	RCFAST	стр. 209	XTAL3	стр. 209
FILE	стр. 252	PAR	стр. 330	RCSLOW	стр. 209		
FLOAT	стр. 253	PHSA	стр. 332	ROUND	стр. 351		
FRQA	стр. 256	PHSB	стр. 332	TRUE	стр. 237		

*** Кроме операторов присваивания.**

Перечень элементов языка Propeller Spin по категориям

Элементы, обозначенные индексом “a”, также доступны в ассемблере Propeller.

Указатели блоков

CON	Объявляет блок констант; стр. 228.
VAR	Объявляет блок переменных; стр. 364.
OBJ	Объявляет блок ссылок на объекты; стр. 288.
PUB	Объявляет блок методов *Public*; стр. 334.
PRI	Объявляет блок методов *Private*; стр. 333.
DAT	Объявляет блок данных; стр. 243.

Конфигурация

CHIPVER	Номер версии чипа Propeller; стр. 203.
CLKMODE	Текущий режим генератора; стр. 208.
_CLKMODE	Режим генератора, заданный приложением (т.чтение); стр. 209.
CLKFREQ	Текущая частота генератора; стр. 204.
_CLKFREQ	Частота генератора, определяемая приложением (т.чтение); с. 206.
CLKSET[a]	Устанавливает режим и частоту генератора; стр. 213.
_XINFREQ	Заданная приложением внешняя частота (только чтение); стр. 392.
_STACK	Заданная приложением область под стек (только чтение); стр. 355.
_FREE	Заданная приложением свободная область (только чтение); с. 255.
RCFAST	Константа для _CLKMODE: внутренний быстрый генератор; стр. 209.
RCSLOW	Константа для _CLKMODE: внутренний медленный генератор; с. 209.
XINPUT	Константа для _CLKMODE: внешний кварц/генератор (пин XI); с. 209.
XTAL1	Константа для _CLKMODE: внешний низкочастотный кварц; стр. 209.
XTAL2	Константа для _CLKMODE: внешний среднечастотный кварц; стр. 209.
XTAL3	Константа для _CLKMODE: внешний высокочастотный кварц; стр. 209.
PLL1X	Константа для _CLKMODE: множитель внешней частоты на 1; стр. 209.
PLL2X	Константа для _CLKMODE: множитель внешней частоты на 2; стр. 209.
PLL4X	Константа для _CLKMODE: множитель внешней частоты на 4; стр. 209.

PLL8X — Константа для _CLKMODE: множитель внешней частоты на 8; с. 209.
PLL16X — Константа для _CLKMODE: множитель внешней частоты на 16; с. 209.

Управление Процессорами

COGID[a] — *ID* текущего *Cog* (0-7); стр. 217.
COGNEW — Запуск следующего доступного *Cog*; стр. 221.
COGINIT[a] — Запуск или перезапуск *Cog* по *ID*; стр. 218.
COGSTOP[a] — Остановить *Cog* по *ID*; стр. 227.
REBOOT — Сброс ИМС Propeller; стр. 339.

Управление Процессом

LOCKNEW[a] — Проверить бит защиты; стр. 268.
LOCKRET[a] — Отменить бит защиты; стр. 271.
LOCKCLR[a] — Снять бит защиты по *ID*; стр. 266.
LOCKSET[a] — Установить бит защиты по *ID*; стр. 272.
WAITCNT[a] — Ожидать пока Системный Счетчик достигнет значения; стр. 373.
WAITPEQ[a] — Ожидать до получения заданной комбинации на линиях B/B; стр. 377.
WAITPNE[a] — Ожидать, пока зад. комбинация присутствует на линиях B/B; стр. 379.
WAITVID[a] — Ожидать видеосинхронизацию и освободить следующую группу цвет/пиксель; стр. 380.

Управление потоками

IF — Условное выполнение одного или более блоков кода; стр. 257.
...ELSEIF
...ELSEIFNOT
...ELSE

IFNOT — Условное выполнение одного или более блоков кода; стр. 263.
...ELSEIF
...ELSEIFNOT
...ELSE

CASE ...OTHER	Вычислить выражение и выполнить блок кода, удовлетворяющий условию; стр. 200.
REPEAT ...FROM ...TO ...STEP ...UNTIL ...WHILE	Выполнять блок кода циклично, неопределенное либо определенное количество раз с возможностью применения счетчика циклов, интервалов, условий выхода и продолжения; стр. 340.
NEXT	Пропустить остальной код блока REPEAT и перейти на следующую итерацию цикла; стр. 287.
QUIT	Выйти из цикла REPEAT; стр. 338.
RETURN	Выйти из PUB/PRI с нормальным статусом и опционально значением возврата; стр. 349.
ABORT	Выйти из PUB/PRI со статусом abort и опционально значением возврата; стр. 187.

Память

BYTE	Объявляет идентификатор размера *byte* либо производит доступ к байту основной памяти; стр. 192.
WORD	Объявляет идентификатор размера *word* либо производит доступ к слову основной памяти; стр. 382.
LONG	Объявляет идентификатор размера *long* либо производит доступ к двойному слову основной памяти; стр. 274.
BYTEFILL	Заполнить байты основной памяти значением; стр. 198.
WORDFILL	Заполнить слова основной памяти значением; стр. 390.
LONGFILL	Заполнить двойные слова основной памяти значением; стр. 281.
BYTEMOVE	Копировать байты из одной области основной памяти в другую; стр. 199.
WORDMOVE	Копировать слова из одной области основной памяти в другую; стр. 391.
LONGMOVE	Копировать двойные слова из одной области основной памяти в другую; стр. 282.

LOOKUP	Получить значение из списка по индексу (1..N); стр. 285.
LOOKUPZ	Получить значение из нуль-базового списка по индексу (0..N−1); стр. 285.
LOOKDOWN	Получить индекс (1..N) совпадающего значения из списка; с. 283.
LOOKDOWNZ	Получить нуль-базовый индекс (0..N−1) совпадающего значения из списка; стр. 283.
STRSIZE	Получить размер строки в байтах; стр. 359.
STRCOMP	Сравнить строку из байтов с другой строкой из байтов; стр. 356.

Директивы

STRING	Объявляет in-line строковое выражение; вычисляется во время компиляции; стр. 358.
CONSTANT	Объявляет in-line выражение-константу; вычисляется во время компиляции; стр. 235.
FLOAT	Объявляет выражение-константу с плавающей точкой; вычисляется во время компиляции; стр. 253.
ROUND	Округлить при компиляции выражение-константу с плавающей точкой до целого; стр. 351.
TRUNC	Отсечь при компиляции выражение-константу с плавающей точкой до целого; стр. 363.
FILE	Импортировать данные из внешнего файла; стр. 252.

Регистры

DIRA[a]	Регистр Направления для 32-битного порта A; стр. 249.
DIRB[a]	Регистр Направления для 32-битного порта B (резерв); стр. 249.
INA[a]	Входной Регистр для 32-битного порта A (чтение); стр. 263.
INB[a]	Входной Регистр для 32-битного порта B (чтен., резерв); стр. 264.
OUTA[a]	Выходной Регистр для 32-битного порта A; стр. 326.
OUTB[a]	Выходной Регистр для 32-битного порта B(резерв); стр. 329.
CNT[a]	Регистр 32-битного Системного Счетчика (чтение); стр. 215.
CTRA[a]	Регистр Управления Счетчика A; стр. 239.
CTRB[a]	Регистр Управления Счетчика B; стр. 239.
FRQA[a]	Регистр Частоты Счетчика A; стр. 256.

FRQB[a]	Регистр Частоты Счетчика B; стр. 256.
PHSA[a]	Регистр ФАПЧ (PLL) Счетчика A; стр. 332.
PHSB[a]	Регистр ФАПЧ (PLL) Счетчика B; стр. 332.
VCFG[a]	Регистр Конфигурации Видео; стр. 368.
VSCL[a]	Регистр Масштаба Видео; стр. 371.
PAR[a]	Регистр Параметров Загрузки *Cog* (чтение); стр. 330.
SPR	Массив регистров специального назначения (PCH); косвенный доступ к *Cog*; стр. 353.

Константы

TRUE[a]	Логическое «истина»: -1 ($FFFFFFFF); стр. 237.
FALSE[a]	Логическое «ложь»: 0 ($00000000) ; стр. 237.
POSX[a]	Максимальное положит. целое: 2147483647 ($7FFFFFFF); с. 237.
NEGX[a]	Максимальное отрицат. целое: -2147483648 ($80000000); с. 237.
PI[a]	Вещественное значение PI: ~3.141593 ($40490FDB); стр. 237.

Переменные

RESULT	Переменная результата по умолчанию для методов PUB/PRI; с. 347.

Унарные операции

+	Положительное (+X); унарное от Add; стр. 298.
-	Отрицание (-X); унарное от Subtract; стр. 299.
--	Pre-декрементировать (--X) или post-декрементировать (X--) и присвоить; стр. 299.
++	Pre-инкрементировать (++X) или post-инкрементировать (X++) и присвоить; стр. 300.
^^	Квадратный корень; стр. 304.
||	Абсолютное значение; стр. 305.
~	Распространить знак с бита 7 (~X) или post-очистить в значение 0 (X~); стр. 305.
~~	Распространить знак с бита 15 (~~X) или post-установить в значение -1(X~~); стр. 306.

?	Случайное число вперед (?X) либо назад (X?); стр. 308.
|<	Дешифровать значение (модули 32; 0-31); стр. 309.
>|	Шифровать *long* по модулю (0 - 32); стр. 310.
!	Побитовое: NOT; стр. 317.
NOT	Логическое: NOT (переводит «не-0» в -1); стр. 319.
@	Символ взятия адреса; стр. 324.
@@	Адрес объекта плюс символическое значение; стр. 325.

Бинарные операции

ПРИМЕЧАНИЕ: Все операторы справа - операторы присваивания.

=	--и--	=	Присваивание константе (блоки CON); стр. 296.
:=	--и--	:=	Присваивание переменной (блоки PUB/PRI); стр. 297.
+	--или--	+=	Сложить; стр. 298.
-	--или--	-=	Вычесть; стр. 299.
*	--или--	*=	Умножить и вернуть младшие 32 бита (знаковое);с. 301.
**	--или--	**=	Умножить и вернуть старшие 32 бита (знаковое); с. 302.
/	--или--	/=	Деление (знаковое); стр. 302.
//	--или--	//=	Модуль (знаковое); стр. 302.
#>	--или--	#>=	Ограничение по минимуму (знаковое); стр. 303.
<#	--или--	<#=	Ограничение по максимуму (знаковое); стр. 304.
~>	--или--	~>=	Арифметический сдвиг вправо; стр. 307.
<<	--или--	<<=	Побитно: Сдвиг влево; стр. 310.
>>	--или--	>>=	Побитно: Сдвиг вправо; стр. 311.
<-	--или--	<-=	Побитно: Циклический сдвиг влево; стр. 312.
->	--или--	->=	Побитно: Циклический сдвиг вправо; стр. 312.
><	--или--	><=	Побитно: Обратное значение; стр. 313.
&	--или--	&=	Побитно: AND; стр. 314.
|	--или--	|=	Побитно: OR; стр. 315.
^	--или--	^=	Побитно: XOR; стр. 316.
AND	--или--	AND=	Логическое: AND (переводит «не-0» в -1); стр. 317.
OR	--или--	OR=	Логическое: OR (переводит «не-0» в -1); стр. 318.

==	--или--	===	Логическое: Равно; стр. 320.
<>	--или--	<>=	Логическое: Не равно; стр. 321.
<	--или--	<=	Логическое: Менее чем (знаковое); стр. 321.
>	--или--	>=	Логическое: Более чем (знаковое); стр. 322.
=<	--или--	=<=	Логическое: Равно или менее (знаковое); стр. 322.
=>	--или--	=>=	Логическое: Равно или более (знаковое); стр. 323.

Символы синтаксиса

%	Признак двоичного числа, как в %1010; стр. 361.
%%	Признак четверичного числа, как в %%2130; стр. 361.
$	Признак шестнадцатеричного числа, как в $1AF, либо ассемблерный символ-указатель текущего адреса «здесь»; стр. 361.
''	Указатель начала строки: ''Hello''; стр. 361.
_	Разделитель групп в константах, либо подчеркивание в идентификаторах; стр. 361.
#	Ссылка объект-константа: obj#constant; стр. 361.
.	Ссылка объект-метод: obj.method(param) или децимальная точка; с. 361.
..	Индикатор диапазона, как в 0..7; стр. 361.
:	Разделитель для возвратной части: PUB method : sym, либо присваивание объекта, и т.д.; стр. 361.
|	Разделитель локальных переменных: PUB method | temp, str; стр. 362.
\	Прерывание задачи, как в \method(parameters); стр. 362.
,	Разделитель списка, как в method(param1, param2, param3); стр. 362.
( )	Указатели списка параметров, как в method(parameters); стр. 362.
[ ]	Указатели индекса массива, как в INA[2]; стр. 362.
{ }	Указатели одно/много- строчных комментариев кода; стр. 362.
{{ }}	Указатели одно/много- строчных комментариев документации; стр. 362.
'	Указатель начала комментария кода; стр. 362.
''	Указатель начала комментария документации; стр. 362.

Элементы языка Spin

В оставшейся части этой главы описываются приведенные выше элементы языка *Spin* в алфавитном порядке. Несколько элементов, для ясности, будут показаны в контексте других. Чтобы найти необходимое описание, используйте номера страниц из приведенного выше перечня по категориям. Многие элементы имеются как в языке *Spin*, так и в ассемблере ИМС Propeller. Такие элементы, детально описаные в этой секции, приводятся со сносками на своих двойников в секциях, которые описывает Глава 5: Справочник по языку ассемблера, начинающаяся со стр. 394.

Правила Идентификаторов

Идентификаторы – это нечувствительные к регистру, буквенно-цифровые имена, созданные либо компилятором (зарезервированные слова), либо разработчиком (слова, определенные пользователем). Они заменяют величины (константы либо переменные), чтобы сделать код более легко читаемым и поддерживаемым. Идентификаторы должны подчиняться следующим правилам:

1) Начинаться с буквы (a – z) либо подчеркивания ‘_’.
2) Содержать только буквы, числа и подчеркивания (a – z, 0 – 9, _); пробелы не допускаются.
3) Должны быть не длиннее 30 символов.
4) Уникальны в рамках объекта; не являются зарезервированным словом (стр. 539) либо идентификатором, ранее уже определенным пользователем.

Представление величин

Величины могут быть введены в двоичном (основание 2), четверичном (основание 4), десятичном (основание 10), шестнадцатеричном (основание 16), либо символьном формате. Численные значения могут также включать подчеркивания, ‘_’, как разделители групп, для более легкого восприятия больших чисел. Далее приведены примеры этих форматов.

Табл. 4-1: Представление величин		
Основание	**Тип величины**	**Примеры**
2	Двоичный	%1010 –или– %11110000_10101100
4	Четверичный	%%2130_3311 – или – %%3311_2301_1012
10	Десятичный (целое)	1024 – или – 2_147_483_647 – или – -25
10	Десятичный (плавающая точка)	1e6 – или – 1.000_005 – или– -.70712
16	Шестнадцатеричный	$1AF – или – $FFAF_126D_8755
n/a	Символьный	"A"

Разделители могут быть использованы на месте запятых (в десятичных величинах) или для формирования логических групп, таких как тетрады, байты, слова и т.д.

Правила Синтаксиса

Вдобавок к детальному описанию, в главе также содержится описание синтаксиса для многих элементов, кратко отображающее все варианты использования конкретного элемента. Определения синтаксиса используют специальные символы для индикации того, когда и как должна быть использована конкретная опция для конкретного элемента.

BOLDCAPS	Символы утолщенного шрифта и верхнего регистра должны вводиться, где показаны.
Bold Italics	Символы с утолщенным шрифтом и курсивом должны заменяться текстом пользователя: идентификаторами, операторами, выражениями и т.д.
. .. : , # | \ [ ] ( )	Точки, двойные точки, двоеточия, запятые, специальные символы, квадратные и круглые скобки должны вводиться, где показаны.
〈 〉	Угловые скобки заключают опциональные элементы. Вводите их, когда необходимо. Скобки не вводите.
((¦))	Двойные круглые скобки заключают взаимоисключающие элементы, разделенные тире. Вводите один, и только один, из таких элементов. Не вводите сами двойные скобки или тире.
…	Символ повторения показывает, что предыдущий элемент или группа может повторяться несколько раз. Дублируйте последний(-е) элемент(ы) при необходимости. Сам этот символ не вводите.
↳	Символ новой строки/отступа показывает, что следующие элементы должны быть введены на новой строке, с как минимум одним отступом (пробелом).
→|	Символ отступа показывает, что следующие элементы должны быть введены с как минимум одним отступом (пробелом).
Одинарная линия	Отделяет различные по структуре опции.
Двойная линия	Отделяет инструкцию от ее значения возврата.

Поскольку элементы ограничиваются специфичными блоками *Spin*, все синтаксические определения начинаются с описания типа необходимого блока. Например, следующий пример синтаксиса показывает, что команда BYTEFILL и ее параметры должна быть либо в блоке PUB, либо PRI, и она может быть одной из многих команд в этом блоке.

((PUB ¦ PRI))
 BYTEFILL (*StartAddress*, *Value*, *Count*)

ABORT

Команда: Выход из метода PUB/PRI со статусом аварийного завершения и с возможным возвратом значения *Value*.

((PUB ¦ PRI))
  ABORT ⟨*Value*⟩

Возвращает: Либо текущее значение RESULT, либо *Value*, если оно задано.

- ***Value*** – это опциональное выражение, чье значение должно быть возвращено из метода PUB или PRI со статусом аварийного завершения.

Описание

ABORT - это одна из двух команд (ABORT и RETURN), которые прекращают выполнение метода PUB или PRI.

ABORT приводит к возврату из метода PUB или PRI со статусом аварийного завершения, что значит, что он постоянно выбирает стек вызовов до тех пор, пока стек либо станет пустым, либо достигнет вызывающего с ловушкой Abort Trap (\), и предоставляет значение процессу.

Команда ABORT полезна в случаях, когда методу необходимо прекратиться и показать ненормальное завершение непосредственному или одному из предыдущих вызвавших его методов. Например, приложение может иметь сложную цепь событий, где любое из этих событий может вывести к различным частям цепи или к решению о выполнении заключительных действий. Проще говоря, каждое приложение, использующее маленькие, специализированные методы, вызываемые во вложенной структуре, должно иметь дело со специфическими «подсобытиями» в цепи. Когда один из простых методов определяет направление действий, это может привести к аварийному завершению, которое полностью завершает вложенный вызов и отменяет выполнение всех промежуточных методов.

Когда ABORT применяется без опционального *Value*, она возвращает текущее значение встроенной переменной RESULT метода PUB/PRI. Если же поле *Value* было заполнено, метод PUB или PRI при аварийном завершении возвращает значение *Value*.

Стек Вызовов

При вызове одного метода из другого, для сохранения адреса возврата после завершения вызванного метода должен быть предусмотрен специальный механизм. Такой механизм называется “стек”, либо “стек вызовов”. Стек представляет собой область памяти ОЗУ, используемую для хранения адресов возврата, значений возврата, параметров и промежуточных результатов. По мере поступления вызовов, стек вызовов растет. По мере возврата из методов (по команде RETURN или при достижении конца метода), стек вызовов уменьшается. Это называется соответственно “заталкиванием” в стек и “выталкиванием” из него.

Команда RETURN выталкивает самые последние данные из стека вызовов для обеспечения возврата к непосредственно вызвавшему его методу – тому, который вызвал только что завершенный метод. Команда же ABORT выталкивает данные из стека до тех пор, пока он не достигнет адреса вызывающего с ловушкой Abort Trap (см. ниже), возвращая выполнение вызывающему методу более высокого уровня через всю промежуточную цепь вложенных вызовов. Все точки возврата между методом, инициирующим аварийное завершение, и методом-ловушкой, игнорируются, и, соответственно, выполнение промежуточных методов отменяется. Таким образом, ABORT предоставляет возможность обратного пути из, возможно, очень глубоких и потенциально сложных цепочек вызовов для обслуживания серьезных проблем на верхнем уровне.

Использование команды ABORT

Любой метод может использовать для выхода команду ABORT. Проверять и обрабатывать статус аварийного завершения методов – дело метода верхнего уровня. Этот метод мог вызвать проблемный метод либо непосредственно, либо через некоторые другие методы. Для выхода по команде ABORT, используйте подобную конструкцию:

```
if <bad condition>
  abort               'If bad condition detected, abort
```

—или—

```
if <bad condition>
  abort <value>       'If bad condition detected, abort with value
```

...где <bad condition> – это условие, определяющее, что метод должен быть аварийно завершен, а <value> – это значение возврата при аварийном завершении.

Ловушка Abort Trap (\ )

Для отлавливания ситуаций с аварийным завершением по команде ABORT, вызов метода или цепи методов, которые потенциально могут быть аварийно завершены, должен предваряться символом ловушки – Abort Trap (обратной наклонной чертой \). Например, если вызываемый метод, с именем MayAbort, может быть аварийно завершен, либо он вызывает другие методы, которые могут быть аварийно завершены, вызывающий метод может отловить эту ситуацию следующим образом:

```
if \MayAbort        'Call MayAbort with abort trap
  abort <value>     'Process abort
```

Какой тип выхода использовал метод MayAbort на самом деле – ABORT или RETURN, вызывающему не известно; здесь вполне возможен возврат по команде RETURN. Поэтому код должен быть написан таким образом, чтобы определить, какой из типов выхода был использован. Некоторые из возможных способов – это: 1) писать код таким образом, чтобы единственным местом, отлавливающим аварийное завершение, был высокоуровневый метод, а остальной код промежуточного уровня обрабатывал события обычным образом, не допуская продвижения RETURN-ов на верхний уровень; либо 2) возвращать из аварийно завершаемых методов определенное значение, которое не может иметь место при нормальном завершении; либо 3) перед выходом из аварийно завершаемого метода взводить глобальный флаг.

Пример использования команды Abort

Далее приведен пример приложения для простейшего робота, в котором робот должен удаляться от объекта, который он определяет своими четырьмя датчиками (Left, Right, Front и Back). Допустим, что методы CheckSensors, Beep, и MotorStuck определены ранее в другом месте.

```
CON
  #0, None, Left, Right, Front, Back  'Direction Enumerations

PUB Main | Direction
  Direction := None
  repeat
    case CheckSensors                 'Get active sensor
      Left  : Direction := Right      'Object on left? Let's go right
      Right : Direction := Left       'Object on right? Let's go left
      Front : Direction := Back       'Object in front? Let's go back
      Back  : Direction := Front      'Object in back? Let's go front
      other : Direction := None       'Otherwise, stay still
    if not \Move(Direction)           'Move robot
      Beep                            'We're stuck? Beep

PUB Move(Direction)
  result := TRUE                      'Assume success
  if Direction == None
    return                            'Return if no direction
  repeat 1000
    DriveMotors(Direction)            'Drive motor 1000 times

PUB DriveMotors(Direction)
  <code to drive motors>
  if MotorStuck
    abort FALSE                       'If motor is stuck, abort
 <more code>
```

В приведенном примере показаны три метода, занимающие различные логические уровни: `Main` (“верхний”), `Move` (“промежуточный”) и `DriveMotors` (“нижний”). Метод верхнего уровня, `Main` – это место принятия решений в приложении, здесь принимаются решения о том, как реагировать на события, такие как активация датчиков и движение моторов. Метод среднего уровня, `Move`, отвечает за перемещение робота на короткое расстояние. Метод нижнего уровня, `DriveMotors`, занимается деталями правильного привода моторов и проверкой успешности результата.

В коде нижнего уровня подобного приложения могут возникать критические события, которые соответственно должны быть обработаны кодом верхнего уровня. Команда **ABORT** может стать инструментом для передачи сообщения коду верхнего уровня без необходимости внедрения сложной системы передачи сообщений между всеми

промежуточными методами. Здесь мы имеем только один метод среднего уровня, но в общем случае на этом месте может быть множество вложенных методов, логически расположенных между верхним и нижним уровнями иерархии.

Метод Main получает сигналы от датчиков, и в условии CASE принимает решение, в каком направлении двигаться роботу. Затем он специальным образом вызывает Move, с использованием символа ловушки Abort Trap, \ , предваряющим его. Метод Move устанавливает свой RESULT в TRUE и затем в цикле вызывает DriveMotors. Если выполнение проходит успешно, Move возвращает TRUE. Метод DriveMotors управляет моторами робота для достижения заданной позиции, и если он определяет, что моторы блокированы и нет возможности их вращать, то он производит аварийное завершение со значением FALSE. Иначе он просто выполняет обычный выход.

Если все в порядке и метод DriveMotors нормально завершается, метод Move также завершается нормально и, в конце концов, возвращает TRUE, то метод Main продолжает нормальное выполнение. Если же DriveMotors обнаруживает проблему, он выполняет выход ABORT, который заставляет ИМС Propeller вытолкнуть из стека весь путь от текущего метода через метод Move до метода Main, где будет найдена ловушка Abort Trap. Метод Move не чувствует этого и полностью завершается. Метод Main проверяет значение, возвращенное после его вызова Move (которое сейчас FALSE, возвращенное прерванным по ABORT глубоко внизу стека методом DriveMotors) и принимает решение выполнить Beep как результат обнаруженного отказа.

Если бы мы не поставили ловушку Abort Trap (\ ) перед вызовом Move, то когда DriveMotors вышел бы по ABORT, стек вызовов выталкивался бы до опустошения и это приложение немедленно бы прекратилось.

BYTE

Объявление: Объявляет однобайтовую переменную, однобайтовые/выровненные по границе байта данные либо выполняет чтение/запись байта основной памяти.

```
VAR
   BYTE Symbol {[Count]}
DAT
   {Symbol} BYTE Data {[Count]}
((PUB | PRI))
   BYTE [BaseAddress] {[Offset]}
((PUB | PRI))
   Symbol.BYTE {[Offset]}
```

- ***Symbol*** – желаемое имя переменной (синтаксис 1) или блока данных (синтаксис 2) или существующее имя переменной (синтаксис 4).
- ***Count*** – опциональное выражение, отображающее количество байт-размерных элементов для *Symbol* (синтаксис 1), либо количество байтовых членов ***Data*** для сохранения в таблице данных.
- ***Data*** – константа либо разделенный запятыми список констант. Также допустимы строки символов в кавычках; они рассматриваются как список разделенных запятыми символов.
- ***BaseAddress*** – выражение, представляющее адрес в основной памяти для чтения или записи. Если параметр *Offset* опущен, *BaseAddress* – это реальный адрес, с которым будет проводиться операция. Если же параметр *Offset* указан, реальным адресом для операции будет *BaseAddress* + *Offset*.
- ***Offset*** – опциональное выражение, указывающее смещение от адреса *BaseAddress* для проведения операции, либо смещение от 0-го байта Symbol.

Описание

BYTE является одним из трех объявлений (BYTE, WORD, и LONG), которые объявляют переменные либо оперируют с памятью. Объявление BYTE используется для:

1) объявления однобайтовой переменной либо массива однобайтовых переменных в блоке VAR, или
2) объявления байт-выровненных и, возможно, байт-размерных, данных в блоке DAT,
3) чтения или записи байта основной памяти по базовому адресу со смещением, или
4) доступа к отдельным байтам в переменных размером слово или двойное слово.

Диапазон значений Byte

Ячейка памяти размером в один байт (8 битов) может содержать значение, равное одной из 2^8 возможных комбинаций битов (то есть одной из 256 комбинаций). Таким образом, диапазон чисел для однобайтовых величин составляет от 0 до 255. Поскольку язык *Spin* выполняет все математические операции, используя 32-битную арифметику со знаком, то любые однобайтовые значения имеют внутреннее представление в виде положительной величины размером 4 байта (1 *long*). Однако число, которое содержится в однобайтовой ячейке, в известной степени зависит от представления компьютера и пользователя о нем. Например, в *Spin*–выражении Вы можете использовать оператор «Распространения Знака 7» (~), стр. 305, для преобразования однобайтовой величины, которую Вы рассматриваете «со знаком» (от -128 до +127) в 4-х байтную long-величину со знаком.

Синтаксис объявления однобайтовой переменной (Синтаксис 1)

Синтаксис 1 объявления BYTE используется в блоках VAR для объявления глобальных однобайтовых переменных, которые могут быть как одиночными, так и массивами.

Например:

```
VAR
  byte Temp           'Temp is a byte
  byte Str[25]        'Str is a byte array
```

В приведенном выше примере объявляется две переменные (идентификатора), Temp и Str. Здесь Temp – это просто одиночная однобайтовая переменная. В строке под объявлением Temp используется опциональное поле *Count* для создания массива из 25 однобайтовых переменных, названного Str. И Temp, и Str доступны из любого метода PUB или PRI в рамках того же объекта, в блоке VAR которого они объявлены; они глобальные по отношению к объекту. Пример обращения к ним приводится ниже.

```
PUB SomeMethod
  Temp := 250         'Set Temp to 250
  Str[0] := "A"       'Set first element of Str to "A"
  Str[1] := "B"       'Set second element of Str to "B"
  Str[24] := "C"      'Set last element of Str to "C"
```

Более детальная информация о подобном использовании BYTE приведена на стр. 364, Объявление переменных (Синтаксис 1) в секции VAR, где BYTE используется в поле *Size*.

Объявление однобайтовых данных (Синтаксис 2)

Синтаксис 2 объявления BYTE используется в блоках DAT для объявления байт-выровненных и/или байт-размерных данных, которые компилируются как константы, расположенные в основной памяти. В блоках DAT такому объявлению позволяется иметь опциональный идентификатор, предваряющий его, который будет использован в дальнейшем (См. DAT, стр. 243). Например:

```
DAT
  MyData byte 64, $AA, 55     'Byte-aligned and byte-sized data
  MyString byte "Hello",0     'A string of bytes (characters)
```

В приведенном выше примере объявляется два идентификатора данных, MyData и MyString. Каждый идентификатор данных указывает на начало байт-выровненных и байт-размерных данных в основной памяти. Значения идентификатора MyData в основной памяти – это соответственно 64, $AA и 55. Значениями MyString в основной памяти являются соответственно “H”, “e”, “l”, “l”, “o”, и 0. Эти данные компилируются в объекте и конечном приложении как часть секции исполнимого кода, и доступны при использовании синтаксиса 3 объявления BYTE для чтения/записи (см. ниже). Для более детальной информации о таком использовании идентификатора BYTE, обратитесь к стр. 244, «Объявление Данных (Синтаксис 1)», секция DAT, и помните, что для поля *Size* в том описании используется BYTE.

При использовании опционального поля *Count,* элементы данных могут повторяться. Например:

```
DAT
  MyData byte 64, $AA[8], 55
```

В приведенном выше примере объявляется таблица однобайтовых байт-выровненных данных с именем MyData, состоящая из следующих десяти величин: 64, $AA, $AA, $AA, $AA, $AA, $AA, $AA,$AA, 55. Согласно полю [8] сразу после объявления величины $AA, она встречается в таблице восемь раз.

Чтение/Запись Байтов Основной Памяти (Синтаксис 3)

Для записи либо чтения байтовых величин основной памяти в блоках PUB и PRI используется Синтаксис 3 объявления BYTE. Такие действия выполняются при использовании инструкций, обращающихся к основной памяти в виде: byte[*BaseAddress*][*Offset*]. Вот пример.

```
PUB MemTest | Temp
  Temp := byte[@MyData][1]            'Read byte value
  byte[@MyStr][0] := "M"              'Write byte value

DAT
  MyData   byte  64, $AA, 55          'Byte-sized/aligned data
  MyStr    byte  "Hello", 0           'A string of bytes
(characters)
```

Из нижней части примера видно, что блок **DAT** размещает свои данные в памяти согласно Рис. 4-1. Первый элемент данных MyData расположен в памяти по адресу $18. Последний его элемент расположен по адресу $1A, сопровождаемый непосредственно за ним, — по адресу $1B, — первым элементом MyStr. Отметьте, что начальный адрес ($18) является произвольным, и изменяется компилятором при изменении кода объекта, либо при включении самого объекта в другое приложение.

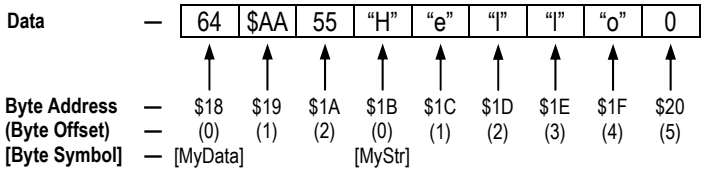


Рисунок 4-1: Структура и адресация однобайтовых данных в Основной Памяти

В верхней части примера, в первой строке метода MemTest, Temp := BYTE[@MyData][1], производится чтение однобайтового значения из основной памяти. Здесь локальная переменная Temp устанавливается в $AA, т.е. в значение, прочитанное по адресу $19. Адрес $19 был определен из адреса идентификатора MyData ($18) плюс смещение в 1 байт. Далее приводится упрощенная диаграмма процесса.

BYTE[@MyData][1] → BYTE[$18][1] → BYTE[$18 + 1] → BYTE[$19]

В следующей строке, BYTE[@MyStr][0] := "M", производится запись однобайтовой переменной в основную память. В ней значение по адресу $1B в основной памяти устанавливается равным символу “M.” Адрес $1B был определен из адреса идентификатора MyStr ($1B) плюс смещение в 0 байт.

BYTE[@MyStr][0] → BYTE[$1B][0] → BYTE[$1B + 0] → BYTE[$1B]

Адресация Основной Памяти

Как показано на Рис. 4-1, основная память — это на самом деле последовательность следующих друг за другом байтов, и адреса доступа к ней вычисляются по позиции байтов. Такой подход действителен для всех команд, использующих адресацию.

Адресация основной памяти всегда байтовая, независимо от размера переменной, к которой осуществляется доступ, — будь она однобайтовая, одинарное либо двойное слово. Это упрощает понимание взаимного расположения байтов, слов и двойных слов, но в то же время вносит некоторые сложности при рассмотрении нескольких элементов одинакового размера, таких как одинарные либо двойные слова. Примеры доступа к элементам с размером одинарное и двойное слово приведены в описании синтаксиса 3: для WORD на стр. 382, и для LONG — на стр. 274.

Для дальнейшего рассмотрения того, как данные распределяются в памяти, обратитесь к описанию Синтаксиса 1 «Объявление Данных» секции DAT на стр. 243.

Альтернативное обращение к памяти

Существует еще один метод доступа к данным, альтернативный использованному в предыдущем примере; в нем можно обращаться к данным напрямую по имени. К примеру, в следующем выражении производится чтение первого байта идентификатора MyData:

```
Temp := MyData[0]
```

А далее читаются его второй и третий байты:

```
Temp := MyData[1]
Temp := MyData[2]
```

Другие особенности доступа

Оба приведенных метода доступа к памяти, — как директива BYTE, так и прямое обращение по имени, — могут использоваться для доступа к любому байту в основной памяти, независимо от его принадлежности к конкретным структурам данных. Несколько примеров:

```
Temp := byte[@MyStr][-1]   'Read last byte of MyData (before MyStr)
Temp := byte[@MyData][3]   'Read first byte of MyStr (after MyData)
Temp := MyStr[-3]          'Read first byte of MyData
Temp := MyData[-2]         'Read byte that is two bytes before MyData
```

В этих примерах производится чтение байтов за рамками логических границ (начальной и конечной точек) структуры объявленных данных. Это может оказаться и

полезным трюком, однако чаще такое происходит по ошибке; будьте внимательны при адресации памяти, особенно когда вы выполняете операции записи.

Доступ к отдельным байтам переменных большего размера (Синтаксис 4)

Синтаксис 4 объявления BYTE используется в блоках PUB и PRI для чтения или записи байтовых компонентов переменных с размером в слово и двойное слово. Например:

```
VAR
  word WordVar
  long LongVar

PUB Main
   WordVar.byte := 0      'Set first byte of WordVar to 0
   WordVar.byte[0] := 0   'Same as above
   WordVar.byte[1] := 100 'Set second byte of WordVar to 100
   LongVar.byte := 25     'Set first byte of LongVar to 25
   LongVar.byte[0] := 25  'Same as above
   LongVar.byte[1] := 50  'Set second byte of LongVar to 50
   LongVar.byte[2] := 75  'Set third byte of LongVar to 75
   LongVar.byte[3] := 100 'Set fourth byte of LongVar to 100
```

В этом примере производится доступ к отдельным байтовым компонентам переменных WordVar и LongVar. Из комментариев ясно, что делает каждая из строк. В конце метода Main переменная WordVar будет равна 25 600, а LongVar будет равна 1 682 649 625.

Аналогичный подход может использоваться при обращении к байтовым составляющим *данных* с размером в одинарное либо двойное слово.

```
PUB Main | Temp
  Temp := MyData.byte[0]            'Read low byte of MyData word 0
  Temp := MyData.byte[1]            'Read high byte of MyData word 0
  MyList.byte[3] := $12             'Write high byte of MyList long 0
  MyList.byte[4] := $FF             'Write low byte of MyList long 1

DAT
  MyData  word  $ABCD, $9988        'Word-sized/aligned data
  MyList  long  $FF998877, $EEEE    'Long-sized/aligned data
```

В первой и второй строках метода Main производится чтение значений из MyData, соответственно $CD и $AB. В третьей строке производится запись значения $12 в старший байт двойного слова элемента 0 массива MyList, изменяющая значение

этого элемента на $12998877. В четвертой строке в байт с индексом 4 массива MyList (младший байт двойного слова элемента номер 1 массива MyList), записывается $FF, изменяя значение элемента на $EEFF.

BYTEFILL

Команда: Заполняет байты основной памяти заданным значением.

((PUB ¦ PRI))
BYTEFILL (*StartAddress*, *Value*, *Count*)

- ***StartAddress*** – адрес первого байта памяти для заполнения значением *Value*.
- ***Value*** – величина, которой будет заполняться память.
- ***Count*** – количество байтов для заполнения, начиная с адреса *StartAddress*.

Описание

BYTEFILL — одна из трех команд (BYTEFILL, WORDFILL, и LONGFILL), которые используются для заполнения блоков основной памяти заданной величиной. BYTEFILL заполняет *Count* байтов основной памяти, начиная с адреса *StartAddress*, значением *Value*.

Использование BYTEFILL

BYTEFILL предоставляет мощное средство для очистки больших блоков байт-размерной памяти. Например:

```
VAR
  byte Buff[100]

PUB Main
  bytefill(@Buff, 0, 100)                  'Clear Buff to 0
```

Первая строка метода Main очищает весь 100-байтовый массив Buff, устанавливая значения всех его элементов в ноли. Для этой задачи BYTEFILL работает быстрее, чем цикл REPEAT.

BYTEMOVE

Команда: Копирует байты основной памяти из одной области в другую.

((PUB ¦ PRI))
  BYTEMOVE (*DestAddress*, *SrcAddress*, *Count*)

- ***DestAddress*** – адрес области основной памяти, куда будет скопирован первый байт из источника.
- ***SrcAddress*** – адрес области основной памяти, где находится первый копируемый байт.
- ***Count*** – количество байт в области источника для копирования в область приемника.

Описание

BYTEMOVE — одна из трех команд (BYTEMOVE, WORDMOVE, и LONGMOVE), используемых для копирования блоков основной памяти из одной области в другую. BYTEMOVE копирует *Count* байтов основной памяти, начиная с адреса *SrcAddress* в основную память, начиная с адреса *DestAddress*.

Использование BYTEMOVE

BYTEMOVE предоставляет мощное средство для копирования больших блоков байт-размерной памяти. Например:

```
VAR
  byte Buff1[100]
  byte Buff2[100]

PUB Main
  bytemove(@Buff2, @Buff1, 100)             'Copy Buff1 to Buff2
```

В первой строке метода Main выполняется копирование всего 100-байтного массива Buff1 в массив Buff2. Для этой задачи BYTEMOVE работает быстрее, чем цикл REPEAT.

CASE

Команда: Проверяет выражение на совпадение выражению(ям) и выполняет соответствующий блок кода, если совпадение было найдено.

```
((PUB ¦ PRI))
   CASE CaseExpression
     →¦ MatchExpression :
       →¦ Statement(s)
   〈→¦ MatchExpression :
       →¦ Statement(s) 〉
   〈→¦ OTHER :
       →¦ Statement(s) 〉
```

- ***CaseExpression*** – выражение для сравнения.
- ***MatchExpression*** – одиночное значение либо разделенный запятыми набор значений и/или выражений, с которыми производится сравнение выражения *CaseExpression*. Каждое *MatchExpression* сопровождается двоеточием (:).
- ***Statement(s)*** – блок из одной или более строк кода, который выполняется, когда *CaseExpression* совпадает с соответствующим *MatchExpression*. Первое или единственное выражение из *Statement(s)* может указываться справа от двоеточия в строке *MatchExpression*, либо под нею и с небольшим отступом от самого *MatchExpression*.

Описание

CASE — одна из трех условных команд (IF, IFNOT, и CASE), которые производят условное выполнение блока кода. Использование структуры CASE является предпочтительным по сравнению с IF..ELSEIF..ELSE, если необходимо проверить выражение *CaseExpression* на равенство многим различным значениям.

CASE сравнивает *CaseExpression* со значениями каждого из *MatchExpression*, по порядку, и, если совпадение найдено, выполняет соответствующий код *Statement(s)*. Если совпадений не найдено, выполняются *Statement(s)* в блоке, ассоциированном с опциональной командой OTHER.

Отступы важны

ВАЖНО: Отступы важны! Язык *Spin* чувствителен к отступам (даже на один пробел) в строках, сопровождающих команды условного выполнения, что используется для

определения, принадлежат ли они блоку данной команды, или нет. Чтобы указать программе *Propeller Tool* индицировать такие логически сгруппированные блоки кода на экране, Вы можете нажать *Ctrl+I* для включения индикаторов блок-групп. Повторное нажатие *Ctrl+I* отключит эту функцию. См. «Отступы и Выступы», стр. 84, и «Индикаторы Блок-Групп», стр. 89.

Использование CASE

CASE удобно использовать, когда в зависимости от результата выражения должно быть выполнено одно из многих действий. В следующем примере предполагается, что переменные A, X и Y уже определены ранее.

```
case X+Y                'Test X+Y
  10, 15: !outa[0]      'X+Y = 10 or 15? Toggle P0
  A*2   : !outa[1]      'X+Y = A*2? Toggle P1
  30..40: !outa[2]      'X+Y in 30 to 40? Toggle P2
X += 5                  'Add 5 to X
```

Поскольку строки *MatchExpression* введены с отступом от строки CASE, они принадлежат структуре CASE и выполняются в зависимости от результатов сравнения выражения *CaseExpression*. Следующая строка, X += 5, введена без отступа от CASE, поэтому она выполняется независимо от результатов CASE.

В этом примере значение выражения X + Y сравнивается с 10 или 15, A*2 и диапазоном от 30 до 40. Если X + Y равно 10 или 15, то P0 переключается. Если X + Y равно A*2, то P1 не переключается. Если же X + Y находится в диапазоне от 30 до 40, включительно, то переключается P2. Не зависимо от того, найдены совпадения или нет, далее будет выполняться строка X += 5.

Использование опции OTHER

Опциональный компонент OTHER в конструкции CASE похож на опциональный компонент ELSE для структуры IF. Например:

```
case X+Y                'Test X+Y
  10, 15: !outa[0]      'X+Y = 10 or 15? Toggle P0
  25    : !outa[1]      'X+Y = 25? Toggle P1
  20..30: !outa[2]      'X+Y in 20 to 30? Toggle P2
  OTHER : !outa[3]      'Othewise toggle P3
X += 5                  'Add 5 to X
```

Отличие этого примера от предыдущего — в наличии третьего условия *MatchStatement*, проверяющего диапазон от 20 до 30, а также опционального компонента OTHER. Если X

+ Y не равно 10, 15, 25, или не находится в диапазоне от 20 до 30, выполняется блок кода *Statement(s)* в секции OTHER. Затем, как и ранее, выполняется строка X += 5.

В этом примере нужно понять важную концепцию. Переключается линия P0, если результат X + Y равен 10 или 15, либо переключается P1, если X + Y равно 25, либо переключается P2, если X + Y находится в диапазоне от 20 до 30, и т.д. Так происходит, потому что *MatchExpressions* проверяются один за другим, в порядке, в котором они приведены в списке, и выполняться будет лишь один блок кода первого совпавшего выражения, ни одно из оставшихся выражений после этого не проверяется. Это значит, что если бы мы перегруппировали строки 25 и 20..30, так, что сначала бы выполнялась проверка на диапазон 20..30, мы бы получили ошибку:

```
  case X+Y                  'Test X+Y
    10, 15: !outa[0]        'X+Y = 10 or 15? Toggle P0
    20..30: !outa[2]        'X+Y in 20 to 30? Toggle P2
    25    : !outa[1]        'X+Y = 25? Toggle P1 <-- THIS NEVER RUNS
```

Этот пример содержит ошибку, поскольку даже при равенстве X + Y значению 25, это условие совпадения никогда не будет проверяться, так как предыдущее условие 20..30 проверится раньше и даст истину, после чего его блок кода выполнится и дальнейшие проверки производиться не будут.

Варианты Условий

Примеры, приведенные выше, используют лишь одну строку на каждый блок *Statement*, но, на самом деле, каждый из таких боков может состоять из множества строк кода. Блок *Statement(s)* может вводиться снизу с отступом от самого условия *MatchExpression*. Такие варианты показаны в следующих двух примерах.

```
case A                      'Test A
  4     : !outa[0]          'A = 4? Toggle P0
  Z+1   : !outa[1]          'A = Z+1? Toggle P1
          !outa[2]          'And toggle P2
  10..15: !outa[3]          'A in 10 to 15? Toggle P3
case A                      'Test A
  4:                        'A = 4?
    !outa[0]                'Toggle P0
  Z+1:                      'A = Z+1?
    !outa[1]                'Toggle P1
    !outa[2]                'And toggle P2
  10..15:                   'A in 10 to 15?
    !outa[3]                'Toggle P3
```

CHIPVER

Команда: Получить номер версии ИМС Propeller .

((PUB ¦ PRI))
  CHIPVER

Возвращает: Номер версии ИМС Propeller.

Объяснение

Команда CHIPVER читает и возвращает номер версии ИМС Propeller. Например:

```
V := chipver
```

В этом примере переменной V присваивается номер версии ИМС Propeller, в этом случае 1. Будущие приложения Propeller могут использовать эту функцию для определения версии и типа ИМС Propeller, на которой они выполняются, и, при необходимости, вносить изменения в свою работу.

CLKFREQ

Команда: Текущая частота Системного Генератора – тактовая частота, с которой работает *Cog*.

((PUB ¦ PRI))
  CLKFREQ

Возвращает: Текущую частоту Системного Генератора, в Гц.

Описание

Величина, возвращаемая CLKFREQ - это значение системной тактовой частоты, определяемое текущим режимом работы генератора (тип генератора, усиление, и установки ФАПЧ) и значением частоты внешнего сигнала на входе XI, если он используется. Объекты используют CLKFREQ для обеспечения правильной временной базы при формировании задержек в чувствительных ко времени приложениях. Например:

```
waitcnt(clkfreq / 10 + cnt)  'wait for .1 seconds (100 msec)
```

Здесь значение CLKFREQ делится на 10 и результат прибавляется к CNT (текущему значению Системного Счетчика), затем ожидается (WAITCNT), пока Системный Счетчик не достигнет заданного значения. Поскольку CLKFREQ – это количество циклов в секунду, то деление на 10 дает количество циклов в 0.1 секунду (или в 100 мсек). Таким образом, не зависимо от того, сколько времени необходимо на вычисление, это выражение останавливает выполнение программы процессора на 100 мсек. В таблице снизу приведено еще несколько примеров вычисления количества циклов тактовой системной частоты в зависимости от необходимого времени задержки.

Табл. 4-2: Количество циклов Системной Частоты для заданного времени

Выражение	Результат
clkfreq / 10	Циклов для 0.1 секунды (100 мсек)
clkfreq / 100	Циклов для 0.01 секунды (10 мсек)
clkfreq / 1_000	Циклов для 0.001 секунды (1 мсек)
clkfreq / 10_000	Циклов для 0.0001 секунды (100 мксек)
clkfreq / 100_000	Циклов для 0.00001 секунды (10 мксек)
clkfreq / 9600	Циклов для периода на 9600 бод (~ 104 мксек)
clkfreq / 19200	Циклов для периода на 19200 бод (~ 52 мксек)

Значение, которое возвращает CLKFREQ, на самом деле читается из двойного слова по адресу 0 (первая позиция) в ОЗУ, и при изменении приложением режима генератора — хоть напрямую, хоть через команду CLKSET, величина его изменяется. Объекты с жесткими требованиями по временной базе должны использовать CLKFREQ для автоматической подстройки под новые установки.

CLKFREQ и _CLKFREQ

Хотя CLKFREQ и тесно связана с _CLKFREQ, но все же это не то же самое. CLKFREQ – это команда, которая возвращает текущую тактовую частоту, в то время как _CLKFREQ – константа, определяемая приложением и содержащая значение тактовой частоты Системного Генератора при старте приложения. Другими словами, CLKFREQ – это текущая частота генератора, а _CLKFREQ – его исходная частота; они обе могут иметь одинаковые значения, хотя, конечно же, могут быть и разными.

_CLKFREQ

Константа: Предустановленная, устанавливаемая один раз константа для задания тактовой частоты Системного Генератора.

```
CON
  _CLKFREQ = Expression
```

- ***Expression*** – целое выражение, отображающее частоту Системного Генератора при старте приложения.

Описание

_CLKFREQ задает тактовую частоту Системного Генератора при старте приложения. Это предустановленная константа, значение которой задается верхним объектным файлом приложения. Приложение может установить _CLKFREQ прямо, либо косвенно, как результат установок _CLKMODE и _XINFREQ.

Верхний объектный файл приложения (тот, с которого начинается компиляция) в своем блоке CON может установить величину _CLKFREQ. Это определит исходную тактовую частоту выполнения приложения, то есть частоту, на которую переключится Системный Генератор, как только приложение будет загружено и запущено на выполнение.

Приложение в блоке CON может задать либо _CLKFREQ, либо _XINFREQ; эти константы взаимно вычисляемые, и одна из них автоматически определяется при задании другой.

В приводимых далее примерах предполагается, что эти константы уже содержатся в верхнем объектном файле. Любые установки _CLKFREQ в объектах-потомках компилятором игнорируются.

Например:

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _CLKFREQ = 32_000_000
```

Первое объявление в приведенном выше блоке CON устанавливает режим генератора на внешний низкочастотный резонатор и множитель ФАПЧ, равный 8. Второе объявление устанавливает частоту Системного Генератора, равную 32 МГц, что означает, что частота внешнего резонатора должна быть 4 МГц, потому как 4 МГц* 8 = 32 МГц. В

результате этих объявлений значение _XINFREQ автоматически устанавливается равным 4 МГц.

```
CON
  _CLKMODE = XTAL2
  _CLKFREQ = 10_000_000
```

Эти два объявления устанавливают режим генератора на внешний среднечастотный резонатор, без умножения частоты при помощи ФАПЧ, и тактовую частоту Системного Генератора, равную 10 МГц. В результате таких объявлений, значение _XINFREQ также автоматически устанавливается равным 10 МГц.

_CLKFREQ и CLKFREQ

Хотя CLKFREQ и тесно связана с _CLKFREQ, однако это не то же самое. CLKFREQ – это команда, которая возвращает текущую тактовую частоту, в то время как _CLKFREQ – константа, определяемая приложением и содержащая значение тактовой частоты Системного Генератора при старте приложения. Другими словами, CLKFREQ – это текущая частота генератора, а _CLKFREQ – это его исходная частота; обе они могут иметь одинаковые значения, хотя, конечно же, могут быть и разными.

CLKMODE

Команда: Установка текущего режима работы генератора.

((PUB ¦ PRI))
  CLKMODE

Возвращает: Текущий режим работы генератора .

Описание

Режим работы генератора – это байтовая переменная, определяемая приложением во время компиляции по значению из регистра CLK. Подробное описание возможных режимов приведено в секции Регистр CLK, на стр. 33. Например:

```
Mode := clkmode
```

Это выражение используется для присваивания переменной Mode значения текущего режима работы генератора. Многие приложения поддерживают установки генератора неизменными, однако, некоторые приложения могут изменять эти установки во время исполнения для подстройки временных соотношений, введения режимов энергосбережения и т.д. Некоторые объекты должны учитывать возможные изменения режимов работы генератора для поддержания временных соотношений и правильного функционирования.

CLKMODE и _CLKMODE

Хотя CLKMODE и тесно связана с _CLKMODE, однако это не одно и то же. CLKMODE – это команда, которая возвращает текущий режим работы генератора (в формате битов регистра CLK), в то время как _CLKMODE – это константа, определяемая приложением и содержащая требуемый режим работы генератора при старте приложения (в формате констант, устанавливающих режим генератора, просуммированных по ИЛИ). Оба идентификатора могут описывать один и тот же режим работы, но при этом их значения не будут одинаковыми.

_CLKMODE

Константа: Предопределенная, устанавливаемая один раз на уровне приложения константа для задания режима работы генератора .

```
CON
  _CLKMODE = Expression
```

- ***Expression*** – целое выражение, составленное из одной или двух констант установки режима генератора, представленных в Табл. 4-3. Таким образом устанавливается режим работы генератора при старте приложения.

Описание

_CLKMODE используется для настройки Системного Генератора. Это предопределенная константа, значение которой задается верхним объектным файлом приложения. Режим работы генератора – это байт, величина которого определяется во время компиляции комбинацией констант RCxxxx, XINPUT, XTALx и PLLxx. Табл. 4-3 описывает константы установки режимов генератора. Обратите внимание, что не любая комбинация корректна; в Табл. 4-4 показаны все корректные комбинации.

Табл. 4-3: Константы установки режимов генератора

Константа установки режима[1]	XO Сопротивление[2]	XI/XO Емкость[2]	Описание
RCFAST	Не определено	н/д	Внутр. быстрый генератор (~12 МГц). От 8 до 20 МГц. (Умолчан.)
RCSLOW	Не определено	н/д	Внутренний медленный генератор (~20 кГц). От 13 кГц to 33 кГц.
XINPUT	Не определено	6 pF (вывода)	Внешняя частота/генератор (от 0Гц до 80 МГц); только пин XI
XTAL1	2 kΩ	36 pF	Внешний медленный резонатор (от 4 МГц до 16 МГц)
XTAL2	1 kΩ	26 pF	Внешний среднечастотный резонатор (от 8 МГц до 32 МГц)
XTAL3	500 Ω	16 pF	Внешний высокочастотный резонатор (от 20 МГц до 80 МГц)
PLL1X	н/д	н/д	Множитель внешней частоты x1
PLL2X	н/д	н/д	Множитель внешней частоты x2
PLL4X	н/д	н/д	Множитель внешней частоты x4
PLL8X	н/д	н/д	Множитель внешней частоты x8
PLL16X	н/д	н/д	Множитель внешней частоты x16

1. Все константы так же доступны в ассемблере Propeller.

2. Все необходимые резисторы/конденсаторы встроены внутри ИМС Propeller.

Табл. 4-4: Корректные комбинации констант и значений регистра CLK	
Корр. комбинация Значение CLK	**Корр. комбинация Значение CLK**
RCFAST 0_0_0_00_000	XTAL1 + PLL1X 0_1_1_01_011 XTAL1 + PLL2X 0_1_1_01_100 XTAL1 + PLL4X 0_1_1_01_101 XTAL1 + PLL8X 0_1_1_01_110 XTAL1 + PLL16X 0_1_1_01_111
RCSLOW 0_0_0_00_001	
XINPUT 0_0_1_00_010	
XTAL1 0_0_1_01_010 XTAL2 0_0_1_10_010 XTAL3 0_0_1_11_010	XTAL2 + PLL1X 0_1_1_10_011 XTAL2 + PLL2X 0_1_1_10_100 XTAL2 + PLL4X 0_1_1_10_101 XTAL2 + PLL8X 0_1_1_10_110 XTAL2 + PLL16X 0_1_1_10_111
XINPUT + PLL1X 0_1_1_00_011 XINPUT + PLL2X 0_1_1_00_100 XINPUT + PLL4X 0_1_1_00_101 XINPUT + PLL8X 0_1_1_00_110 XINPUT + PLL16X 0_1_1_00_111	XTAL3 + PLL1X 0_1_1_11_011 XTAL3 + PLL2X 0_1_1_11_100 XTAL3 + PLL4X 0_1_1_11_101 XTAL3 + PLL8X 0_1_1_11_110 XTAL3 + PLL16X 0_1_1_11_111

Верхний объектный файл приложения (файл, с которого начинается компиляция) в своем блоке **CON** может задать значение **_CLKMODE**. Это определит исходную тактовую частоту приложения, то есть режим, в который переключится Системный Генератор после загрузки и запуска приложения на выполнение. В приводимых далее примерах предполагается, что такие установки уже входят в состав верхнего объектного файла. Любые установки **_CLKMODE** в дочерних объектах компилятором игнорируются. Например:

```
CON
  _CLKMODE = RCFAST
```

Здесь задается режим работы с быстрым внутренним RC-генератором. Системный генератор с такой настройкой будет работать на частоте примерно 12 МГц. Установка **RCFAST** – это установка по умолчанию, поэтому если даже константа **_CLKMODE** не была задана, будет использоваться это значение. Отметьте, что совместно с внутренним RC генератором не может использоваться цепь ФАПЧ. Вот пример с использованием внешнего источника:

```
CON
  _CLKMODE = XTAL1 + PLL8X
```

Здесь режим работы генератора устанавливается на использование внешнего низкочастотного резонатора (XTAL1), цепь ФАПЧ включена и установлен отвод умножителя частоты 8x (PLL8X). Если, к примеру, на XI и XO подключен внешний резонатор 4 МГц, частота сигнала умножается на 16 (генератор ФАПЧ всегда умножает на 16), но используется отвод с множителя 8x; системная частота при этом будет составлять 4МГц * 8 = 32МГц.

```
CON
  _CLKMODE = XINPUT + PLL2X
```

В этом примере режим генератора устанавливается на использование сигнала внешней частоты/генератора, присоединенного только к пину XI, цепь ФАПЧ включена и настроена на множитель 2x. Если на XI подан сигнал от внешнего осциллятора 8МГц, то системный генератор будет работать на частоте 16 МГц: 8 МГц * 2.

Отметьте, что если нет необходимости в использовании цепи ФАПЧ, она может быть отключена путем отсутствия инициализации любого из множителей, например:

```
CON
  _CLKMODE = XTAL1
```

Здесь генератор настраивается на работу с внешним низкочастотным кварцевым резонатором, но оставляет цепь ФАПЧ отключенной; системная частота будет равна частоте внешнего резонатора.

Параметры настройки _CLKFREQ и _XINFREQ

Для упрощения, в приведенных выше примерах показаны лишь установки _CLKMODE, но за ними должны следовать установки либо _CLKFREQ, либо _XINFREQ для того, чтобы объекты могли определить реальную частоту системного генератора. Далее приведем второй пример с внешним резонатором на частоту 4 МГц (_XINFREQ).

```
CON
  _CLKMODE = XTAL1 + PLL8X     'low-speed crystal x 8
  _XINFREQ = 4_000_000         'external crystal of 4
```

Этот пример полностью повторяет приведенный ранее, но здесь _XINFREQ указывает, что частота внешнего резонатора равна 4 МГц. ИМС Propeller использует это значение вместе с установкой _CLKMODE для определения тактовой частоты Системного Генератора (по команде CLKFREQ) с тем, чтобы объекты могли правильно подстроить свои временные соотношения. См _XINFREQ, стр. 392.

_CLKMODE и CLKMODE

CLKMODE связана с _CLKMODE, однако это не одно и то же. CLKMODE – это команда, которая возвращает текущий режим работы генератора (в формате битов регистра CLK), в то время как _CLKMODE – это константа, определяемая приложением и содержащая требуемый режим работы генератора при старте приложения (в формате констант, устанавливающих режим генератора, просуммированных по ИЛИ). Оба идентификатора могут описывать один и тот же режим работы, но при этом их значения не будут одинаковыми.

CLKSET

Команда: Устанавливает режим работы и тактовую частоту Системного Генератора во время выполнения приложения.

((PUB ¦ PRI))
  CLKSET (*Mode*, *Frequency*)

- ***Mode*** – целое выражение, которое будет записано в регистр CLK для изменения режима работы генератора.
- ***Frequency*** – целое выражение, определяющее системную тактовую частоту.

Описание

Одним из мощных свойств архитектуры ИМС Propeller является возможность изменять режим работы генератора в процессе выполнения приложения. К примеру, приложению может понадобиться переключаться вперед-назад между работой на медленной скорости (для малого потребления) и на быстрой (для высокоскоростных задач). Команда CLKSET используется для изменения режима работы и частоты системного генератора во время выполнения приложения. Это эквивалент констант _CLKMODE и _CLKFREQ, определяемых приложением во время исполнения. Например:

```
clkset(%01101100, 4_000_000)                    'Set to XTAL1 + PLL2x
```

Здесь режим работы устанавливается на низкочастотный внешний резонатор и цепь ФАПЧ с умножителем 2, в результате чего получаем частоту системного генератора (CLKFREQ), равную 4МГц. После выполнения этой команды, команды CLKMODE и CLKFREQ выдадут использующим их объектам измененные значения.

В общем случае, для переключения между режимами генератора достаточно одной команды CLKSET. Однако, в случае переключения с внутреннего источника частоты на внешний кварц (установки бита OSCENA), процесс необходимо выполнять в три этапа:

1) Установить биты PLLENA,OSCENA,OSCM1 и OSCM0 регистра CLK как необходимо (см. «Регистр CLK» на стр. 33 для подробной информации);
2) Подождать 10 мсек, чтобы дать внешнему кварцу время для стабилизации;
3) Установить биты CLKSELx регистра CLK соответствующим образом для переключения Системного Генератора на новый источник.

Приведенный процесс переключения необходим только в случае включения цепи внешнего кварцевого резонатора. В случае, когда цепь внешнего кварцевого резонатора не изменяет своего режима работы – остается либо включена, либо выключена – вносить дополнительные изменения в процесс переключения режима работы генератора нет необходимости. Для информации о методах переключения частоты см. объект «Clock» библиотеки *Propeller Library*.

ПРИМЕЧАНИЕ: Процесс переключения источника частоты занимает у ИМС Propeller примерно 75 мксек времени.

CNT

Регистр: Регистр Системного Счетчика.

((PUB ¦ PRI))
  CNT

Возвращает: Текущее значение 32-битного Системного Счетчика.

Описание

Регистр CNT содержит текущее значение глобального 32-битного Системного Счетчика. Системный счетчик служит единой временной базой для всех процессоров; он инкрементирует свое 32-битное значение каждый период тактовой системной частоты.

При включении питания/сбросе, системный счетчик стартует со случайной величины и считает от нее вверх, инкрементируя ее каждый период системной частоты. Поскольку Системный Счетчик является ресурсом только для чтения, каждый из *Cog* может читать его одновременно с другими и может использовать возвращенное значение для синхронизации событий, подсчета циклов и измерения времени.

Использование CNT

Чтобы получить текущее значение Системного Счетчика, необходимо прочесть регистр CNT. Само это реальное значение не имеет никакой практической пользы, но разность между последующими чтениями очень важна. Наиболее часто регистр CNT используется для задержки выполнения на заданный интервал времени либо для синхронизации события с началом заданного окна времени. В следующих примерах показано использование инструкции WAITCNT для достижения таких целей.

```
waitcnt(3_000_000 + cnt)       'Wait for 3 million clock cycles
```

Приведенный выше код – это пример “фиксированной задержки.” Он задерживает работу *Cog*-а на 3 миллиона периодов тактовой частоты (около ¼ секунды при работе от внутреннего быстрого генератора).

В *Spin*-коде, при использовании CNT внутри команды WAITCNT, как указано выше, убедитесь, что выражение записано в виде “offset + cnt”, а не “cnt + offset”, а также что величина offset не меньше 381 — для учета затрат работы *Spin*-интерпретатора и исключения неожиданно длинных задержек. Для детальной информации см. секцию «Фиксированные задержки» в описании команды WAITCNT на стр. 373.

Далее приведен пример “синхронизированной задержки”. В этом примере сначала запоминается текущее значение счетчика, а затем каждую миллисекунду выполняется действие (переключение линии), с точностью генератора, от которого работает ИМС Propeller.

```
PUB Toggle | TimeBase, OneMS
  dira[0]~~                     'Set P0 to output
  OneMS := clkfreq / 1000       'Calculate cycles per 1 millisecond
  TimeBase := cnt               'Get current count
  repeat                        'Loop endlessly
    waitcnt(TimeBase += OneMS)   ' Wait to start of next millisecond
    !outa[0]                     ' Toggle P0
```

Здесь сначала линия В/В P0 была установлена как выход. Затем локальной переменной `OneMS` было присвоено значение текущей системной частоты, деленной на 1000, т.е. количество циклов частоты на 1 миллисекунду времени. Далее локальной переменной `TimeBase` присвоено текущее значение Системного Счетчика. В завершение, две последние строки кода повторяются в бесконечном цикле, каждый раз ожидая начала следующей миллисекунды и затем переключая состояние линии P0.

Для дополнительной информации см. описание команды **WAITCNT,** секции «Фиксированные задержки», стр. 373, и «Синхронизированные задержки», стр. 374.

Регистр **CNT** является регистром только для чтения, поэтому в коде *Spin* ему не должно присваиваться никакое значение (т.е. он не должен присутствовать слева от оператора := либо другого оператора присваивания); в коде на Propeller-ассемблере этот регистр должен использоваться только как источник (*source*, *s*-поле) (т.е.: **mov** *dest*, *source*).

COGID

Команда: Возвращает *ID* номер текущего процессора (0-7).

((PUB ¦ PRI))
  COGID

Возвращает: *ID* текущего процессора (0-7).

Описание

Значение, возвращаемое **COGID** – это *ID*-номер процессора, выполнившего эту команду. Обычно не имеет значения, какой именно процессор выполнил данную команду, однако, для некоторых объектов может быть важно следить за этим. Например:

```
PUB StopMyself
  'Stop Cog this code is running in
  Cogstop(Cogid)
```

Метод StopMyself в этом примере имеет всего одну строку кода, которая просто вызывает **COGSTOP** с параметром **COGID**. Поскольку **COGID** возвращает *ID* номер процессора, выполняющего этот код, эта подпрограмма приводит к тому, что *Cog* останавливает сам себя.

COGINIT

Команда: Запуск или перезапуск процессора по *ID*-номеру, для выполнения *Spin*-кода либо кода ассемблера Propeller.

((PUB ¦ PRI))
COGINIT (*CogID*, *SpinMethod* 〈 (*ParameterList*) 〉, *StackPointer*)

((PUB ¦ PRI))
COGINIT (*CogID*, *AsmAddress*, *Parameter*)

- ***CogID*** – *ID*-номер (0–7) процессора для запуска или перезапуска. При *CogID*, большем 7 будет запущен следующий доступный процессор (при наличии).
- ***SpinMethod*** – PUB либо PRI метод *Spin*, который должен запуститься в стартующем *Cog-е*. За ним в скобках может идти список его параметров.
- ***ParameterList*** – опциональный, разделенный запятыми список из одного или более параметров для *SpinMethod*. Может иметь место, лишь когда метод *SpinMethod* нуждается в параметрах.
- ***StackPointer*** – указатель на память (массив *long*-ов), зарезервированную для области стека используемого *Cog*. Этот *Cog* использует данную память для хранения временных данных при последующих вызовах и вычислениях выражений. При резервировании недостаточного объема памяти приложение либо не запустится, либо это приведет к неопределенным результатам.
- ***AsmAddress*** – адрес процедуры на ассемблере Propeller в блоке DAT.
- ***Parameter*** используется для передачи значения в новый *Cog* (опционально). Эта величина располагается в регистре только для чтения *Cog* Boot Parameter (PAR) нового процессора. *Parameter* может использоваться для передачи либо простого 14-битного значения, либо адреса блока памяти для использования ассемблерной процедурой. *Parameter* необходим для COGINIT, но если он не нужен Вашей подпрограмме, просто установите его в какое-нибудь значение (например, ноль).

Описание

Метод COGINIT работает точно так же, как и COGNEW (стр. 189), с двумя исключениями: 1) он запускает код в указанном процессоре, чей *ID* номер – это *CogID*, и 2) он не возвращает значений. Поскольку COGINIT работает с процессором, указанным в параметре *CogID*, он может использоваться для остановки и перезапуска активного процессора за один шаг. Это касается и текущего *Cog*, т.е.: *Cog* может использовать

COGINIT для остановки и перезапуска самого себя, для выполнения, возможно, совершенно другого кода.

Имейте в виду, что каждый *cog*, исполняющий *Spin*-код, должен располагать своим собственным стэком; процессоры не могут использовать один и тот же стэк.

Кроме того, необходимо соблюдать осторожность при перезапуске *cog*-а для выполнения *Spin*-кода с указанием стэка, уже используемого этим процессором в данный момент времени. ИМС Propeller всегда создает изначальный образ стэка в заданном стэковом пространстве перед самим запуском *cog*-а. Перезапуск процессора командой COGINIT с указанием под стэк той же области памяти, что используется в текущий момент, приведет к перекрытию областей и вероятному разрушению нового стэка перед тем, как *cog* будет перезапущен. Для исключения такого эффекта, перед выполнением команды COGINIT, для выбранного процессора необходимо выполнить команду COGSTOP.

Код Spin (Синтаксис 1)

Для выполнения заданным процессором указанного *Spin*-метода, команде COGINIT необходимо указать *ID*-номер процессора, имя метода, его параметры и указатель на некоторую область памяти для стека. Например:

```
Coginit(1, Square(@X), @SqStack) 'Launch Square in Cog 1
```

В этом примере выполняется запуск метода Square в процессоре *Cog*1, с передачей в COGINIT адреса X в рамках метода Square и адреса памяти SqStack как указателя на область стека. Для детальной информации см. COGNEW, стр. 221.

Код ассемблера Propeller (Синтаксис 2)

Для запуска ассемблер-кода на выполнение в заданном процессоре, команде COGINIT необходимо указать *ID*-номер процессора, адрес ассемблерной процедуры и величину, которая опционально может быть использована процедурой. Например:

```
coginit(2, @Toggle, 0)
```

В этом примере происходит запуск ассемблерной процедуры Toggle в процессоре *Cog*2 с параметром PAR, равным 0. Для более детальной информации см. пример в описании синтаксиса ассемблерной команды COGNEW, стр. 221, имея в виду, что в том примере вместо COGNEW может быть использована описанная выше команда COGINIT.

Поле параметра

Важно отметить, что поле *параметр* предназначено для передачи адреса на *long*, и поэтому в регистр PAR процессора передаётся лишь 14-битное значение (биты со 2 по 15); младшие два бита всегда сброшены в ноль, чтобы обеспечить выравнивание адреса памяти по двойному слову. Значение, отличное от адреса *long* также может быть передано посредством поля *параметр*, однако оно должно быть предварительно ограничено до 14 бит и сдвинуто на два бита влево (для команд COGNEW/COGINIT), а затем должно быть сдвинуто на 2 бита вправо в самой ассемблерной подпрограмме, получившей это значение.

COGNEW

Команда: Запускает следующий доступный процессор для выполнения кода *Spin* или кода ассемблера Propeller.

((PUB ¦ PRI))
COGNEW (*SpinMethod* 〈(*ParameterList*)〉, *StackPointer*)

((PUB ¦ PRI))
COGNEW (*AsmAddress*, *Parameter*)

Возвращает: При успехе – *ID*-номер запущенного им процессора (0-7), иначе – -1.

- ***SpinMethod*** – PUB или PRI метод *Spin*, запускаемый на выполнение новым *Cog*. Сопровождается списком параметров в скобках (опционально).
- ***ParameterList*** – опциональный, разделенный запятыми список из одного или более параметров для метода *SpinMethod*. Указывается, только если метод *SpinMethod* требует параметры.
- ***StackPointer*** – указатель на память (массив *long*-ов), зарезервированную для области стека запускаемого *Cog*. Этот *Cog* использует данную память для хранения временных данных при последующих вызовах и вычислениях выражений. При резервировании недостаточного объема памяти приложение либо не запустится, либо приведет к неопределенным результатам.
- ***AsmAddress*** – адрес процедуры на ассемблере Propeller, обычно в блоке DAT.
- ***Parameter*** используется для передачи значения в новый *Cog* (опционально). Эта величина располагается в регистре только для чтения *Cog* Boot Parameter (PAR) нового процессора. *Parameter* может использоваться для передачи либо простого 14-битного значения, либо адреса блока памяти для использования ассемблерной процедурой. *Parameter* необходим для COGNEW, но если он не нужен Вашей подпрограмме, просто установите его в какое-нибудь значение (например, ноль).

Описание

COGNEW запускает *Spin*-код или код ассемблера Propeller на выполнение следующим доступным процессором. В случае успешного запуска COGNEW возвращает *ID*-номер запущенного процессора. Если больше не было доступных свободных процессоров, COGNEW возвращает -1. Метод COGNEW работает точно так же, как COGINIT (стр. 187) с двумя отличиями: 1) он запускает код на выполнение в следующем доступном процессоре и 2) он возвращает ID успешно запущенного процессора.

Код Spin (Синтаксис 1)

Для выполнения *Spin*-метода другим процессором, команде **COGNEW** необходимо указать имя метода, его параметры и указатель на некоторую область памяти для стека. Например:

```
VAR
  long SqStack[6]               'Stack space for Square Cog


PUB Main | X
  X := 2                        'Initialize X
  Cognew(Square(@X), @SqStack)  'Launch square Cog
  <check X here>                'Loop here and check X


PUB Square(XAddr)
  'Square the value at XAddr
  repeat                        'Repeat the following endlessly
    long[XAddr] *= long[XAddr]  ' Square value, store back
    waitcnt(2_000_000 + cnt)    ' Wait 2 million cycles
```

В этом примере показаны два метода, `Main` и `Square`. `Main` запускает другой *Cog*, который начинает бесконечно выполнять `Square`, после чего `Main` может контролировать результат в переменной `X`. Метод `Square`, выполняемый другим процессором, берет значение по `XAddr`, возводит его в квадрат и сохраняет результат назад по этому же адресу `XAddr`, после чего ожидает 2 миллиона циклов перед следующим выполнением этой операции. Далее объясним подробнее, но сейчас просто отметим, что `X` стартовал со значения 2, а второй *Cog*, выполняющий `Square`, циклично устанавливал `X` в 4, 16, 256, 65536 и затем – в 0 (переполнил 32 бита), причем независимо от первого процессора, который в это время мог просто проверять значение `X` либо выполнять другую задачу.

Метод `Main` объявляет локальную переменную `X`, которая устанавливается в 2 в первой строке. Затем `Main` запускает новый *Cog* с помощью **COGNEW**, для выполнения метода `Square` в отдельном процессоре. Первый параметр **COGNEW,** `Square(@X)` – это метод *Spin*, который нужно выполнить, и необходимый ему параметр; в этом случае мы передаем ему адрес переменной `X`. Второй параметр **COGNEW**, `@SqStack` – это адрес области памяти под стек, зарезервированной для нового процессора. Когда *Cog* запускается для выполнения кода *Spin*, ему необходима некоторая область под стек, где он может сохранять временные данные, такие, как адреса возврата, параметры и промежуточные

результаты вычислений. Этот пример для правильной работы требует всего 6 *long* в области стека (для подробной информации см. секцию «Необходимость в Пространстве Стека», приведенную ниже).

После того, как команда **COGNEW** выполнена, работают уже два процессора; первый все еще выполняет метод `Main`, а второй запущен для выполнения метода `Square`. Не смотря на то, что они используют код из одного и того же *Spin*-объекта, код они выполняют независимо. Строка "`<check X here>`" может быть заменена на код, который использует переменную `X`.

Необходимость в пространстве стэка

При выполнении *Spin*-кода, в отличие от кода Propeller-ассемблера, процессор требует наличия некоторой временной области памяти, называемой «пространство стэка», для хранения оперативных данных, таких как очереди вызовов, параметры и промежуточные результаты вычислений. Без области стэка на такие комплекстные операции, как вызовы методов, сохранение результатов и сложные вычисления, накладывались бы жесткие ограничения.

В самом начале выполнения кода Propeller-приложения, с самых первых его *Spin*-операторов, компилятор автоматически выделяет некоторое пространство стэка. Для этой цели используется «свободное пространство» основной памяти, следующее за образом приложения. Однако компилятор не в состоянии каждый раз автоматически выделять различные блоки памяти под стэк для *Spin*-кода, который приложение может запускать по своему усмотрению; поэтому приложение должно выделять такое пространство стэка самостоятельно.

Обычно такое пространство стэка выделяется посредством специально предназначенной глобальной переменной, — такой, как переменная `SqStack` в приведенном выше примере. К сожалению, сложно определить, какой именно размер области памяти под стэк должен быть выделен в каждом отдельном случае, поэтому рекомендуется изначально задавать заведомо больший объем памяти (к примеру, 128 *long*-ов или более), а когда объект считается уже завершенным, использовать специальный объект для определения оптимального размера стэка, такой как объект «*Stack Length*» библиотеки *Propeller Library*. Для более подробного описания смотрите объект «*Stack Length*».

Запущен может быть только Spin-код своего объекта

В языке *Spin*, по задумке, объекты должны разумно распоряжаться своими данными, методами, оперирующими этими данными, процессорами, выполняющими эти методы и интерфейсом, используемым другими объектами для воздействия на них. Все эти аспекты служат для поддержания целостности объекта, его удобства и надежности.

Исходя из этого объект и его разработчик вооружены всем необходимым для выделения достаточного пространства стэка, требуемого для запуска *Spin*-кода на выполнение в другом процессоре.

Для осуществления этого принципа, команды `COGNEW` и `COGINIT` не могут запускать *Spin*-код извне содержащего их объекта. Это значит, что приведенное ниже выражение не будет работать так, как могло ожидаться.

```
cognew(SomeObject.Method, @StackSpace)
```

Вместо того, чтобы запустить метод SomeObject.Method на выполнение в новом процессоре, ИМС Propeller, напротив, выполнит сам метод SomeObject.Method в текущем *cog*-е, и если этот метод возвратит значение, оно будет использовано как адрес начала кода, который должен запуститься командой cognew. Эти действия не приведут к первоначально задуманному результату.

Если действительно очень важно запустить Method в другом процессоре, то вместо примера, приведенного выше, объект SomeObject должен быть переписан согласно примеру внизу.

```
VAR
 long StackSpace[8]                   'Stack space for new cog
 byte CogID                           'Stores the ID of new cog

PUB Start
 Stop                                 'Prevent multiple starts
 CogID := cognew(Method, @StackSpace)'Launch method in another cog

PUB Stop
  if CogID > -1
    cogstop(CogID)                    'Stop previously launched cog

PRI Method
  <some code here>
```

В приведенном выше объекте имеется два интерфейсных *public*-метода, — Start и Stop, которые могут быть использованы извне объекта для корректного запуска кода самого объекта в другом процессоре. Важным принципом приведенного является то, что объект сам обеспечивает такую возможность, и при этом сам распоряжается памятью под стэк, необходимой для корректной работы. Отметьте также, что статус метода Method был изменен на *private* (PRI) для исключения прямых вызовов извне.

Код Propeller Ассемблер (Синтаксис 2)

Для запуска кода ассемблера Propeller в другом процессоре, команде COGNEW нужен адрес ассемблерной процедуры и величина, которая опционально может быть использована ассемблерным кодом. Например:

```
PUB Main
  cognew(@Toggle, 0)                            'Launch Toggle code

DAT
                  org 0                         'Reset assembly
pointer
Toggle            rdlong   Delay, #0            'Get clock frequency
                  shr      Delay, #2            'Divide by 4
                  mov      Time,  cnt           'Get current time
                  add      Time,  Delay         'Adjust by 1/4 second
                  mov      dira,  #1            'set pin 0 to output
Loop              waitcnt  Time,  Delay         'Wait for 1/4 second
                  xor      outa,  #1            'toggle pin
                  jmp      #Loop                'loop back

Delay     res     1
Time      res     1
```

Инструкция **COGNEW** в приведенном выше методе `Main`, указывает ИМС Propeller запустить ассемблерный код `Toggle` на выполнение в новом *cog*. В результате Propeller находит следующий доступный *cog* и копирует 496 *long*-ов из содержимого блока **DAT**, начиная с `Toggle`, в ОЗУ этого процессора. После этого инициализируется регистр **PAR** процессора, а его остальные регистры специальных функций обнуляются, и *cog* начинает выполнение ассемблерного кода, стартуя с позиции 0 в своем ОЗУ.

Поле параметр

Важно отметить, что поле *параметр* предназначено для передачи адреса на *long*, и поэтому в регистр **PAR** процессора передаётся лишь 14-битное значение (биты со 2 по 15); младшие два бита всегда сброшены в ноль, чтобы обеспечить выравнивание адреса памяти по двойному слову. Значение, отличное от адреса *long* также может быть передано посредством поля *параметр*, однако оно должно быть предварительно ограничено до 14 бит и сдвинуто на два бита влево (для команд **COGNEW/COGINIT**), а затем должно быть сдвинуто на 2 бита вправо в самой ассемблерной подпрограмме, получившей это значение.

COGSTOP

Команда: Останавливает процессор по его *ID*-номеру.

((PUB ¦ PRI))
 COGSTOP (***CogID***)

- ***CogID*** – *ID*-номер (0 – 7) процессора, который необходимо остановить.

Описание

COGSTOP останавливает cog, чей *ID*-номер равен *CogID*, и переводит его в режим останова. В этом режиме процессор прекращает получать импульсы тактовой частоты Системного Генератора, и поэтому потребление энергии значительно уменьшается.

Для остановки процессора выполните команду COGSTOP с *ID*-номером процессора, который необходимо остановить. Например:

```
VAR
  byte Cog 'Used to store ID of newly started Cog


PUB Start(Pos) : Pass
  'Start a new Cog to run Update with Pos,
  'return TRUE if successful
  Pass := (Cog := Cognew(@Update, Pos) + 1) > 0


PUB Stop
  'Stop the Cog we started earlier, if any.
  if Cog
    Cogstop(Cog~ - 1)
```

В этом примере, взятом из описания к COGNEW, используется COGSTOP в *Public*-методе Stop для остановки процессора, запущенного перед этим методом Start. Для более детальной информации об этом примере см. COGNEW, стр. 221,.

CON

Объявление: Объявляет блок констант .

```
CON
  Symbol = Expression ⟨(( , ¦ ↵)) Symbol = Expression⟩...
```

```
CON
  #Expression (( , ¦ ↵)) Symbol ) ⟨[Offset ] ⟩ ⟨(( , ¦ ↵)) Symbol ⟨[Offset ] ⟩ ⟩...
CON
  Symbol ⟨[Offset ] ⟩ ⟨(( , ¦ ↵)) Symbol ⟨[Offset ] ⟩ ⟩...
```

- ***Symbol*** – желаемое имя константы.
- ***Expression*** – любое корректное целое либо вещественное, алгебраическое выражение-константа. Выражение может также включать другие константы, которые были объявлены ранее.
- ***Offset*** – опциональное выражение, по которому осуществляется установка величины смещения для константы ***Symbol***, следующей за ним. Если ***Offset*** не используется, по умолчанию используется значение смещения 1. Используйте ***Offset*** для задания следующего значения перечислимой константы ***Symbol*** отличного от величины этого ***Symbol*** плюс один

Описание

Блок констант – это секция исходного кода, в которой объявляются глобальные идентификаторы констант и глобальные установки конфигурации ИМС Propeller. Это один из шести специальных блоков объявлений (CON, VAR, OBJ, PUB, PRI, и DAT), которые обеспечивают четкую структуру языка *Spin*.

Константы – это численные величины, которые не изменяются во время выполнения. Они могут быть определены посредством простых величин (1, $F, 65000, %1010, %%2310, “A”, и т.д.) или выражений, называемых константными выражениями (25 + 16 / 2, 1000 * 5, и т.д.), которые всегда при решении дают определенную величину.

Блок констант – это область кода, отдельно используемая для присваивания идентификаторов (ассоциированных имен) значениям констант таким образом, что эти идентификаторы могут использоваться везде, где в коде необходима ассоциированная с ними константная величина. Такой подход делает код более читабельным и легким для поддержки, в случае, если Вам станет необходимо поменять величину константы, встречающуюся во многих местах. Эти константы являются глобальными по

отношению ко всему объекту, поэтому использовать их может любой его метод. Существует множество способов определять константы, все они описаны ниже.

Общая форма объявления констант (Синтаксис 1)

Самые общие формы объявления констант начинаются с идентификатора **CON** на строке, под которой расположены одно или более объявлений. Идентификатор **CON** должен начинаться с первого (самого левого) столбца строки, и мы рекомендуем все дальнейшие строки блока вводить с отступом как минимум в один пробел. Выражения могут состоять из комбинаций чисел, операторов, круглых скобок и символов в кавычках. См. «Операторы Spin», стр. 291, где показаны примеры выражений.

Пример:

```
CON
  Delay = 500
  Baud = 9600
  AChar = "A"
```

—или—

```
CON
  Delay = 500, Baud = 9600, AChar = "A"
```

Оба этих примера создают идентификаторы `Delay`, значение которого 500, `Baud` со значением 9600, и `Achar`, который имеет значение символа “A”. Для объявления `Delay`, например, мы могли бы также использовать и алгебраическое выражение, такое как:

```
  Delay = 250 * 2
```

В приведенном выражении результат тот же, равен 500, но применение выражения может сделать код более понятным, если результат – не просто произвольная величина.

Блок **CON** также используется для задания глобальных установок, таких как установки системного генератора. В примере, приведенном ниже, показано, как установить режим генератора на работу с низкочастотным резонатором, умножителем ФАПЧ 8x, и указать, что частота на пине XIN равна 4 МГц.

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _XINFREQ = 4_000_000
```

Для подробного описания этих установок, см. **_CLKMODE**, стр. 209, и **_XINFREQ**, стр. 392.

Величины в формате с плавающей точкой также могут быть определены как константы. Этот формат используется для представления вещественных чисел (с дробными частями), и его 32-битная кодировка отличается от целых констант. Для задания константы с плавающей точкой, Вы должны четко указать, что величина – вещественная; выражение должно быть либо просто вещественной величиной, либо должно полностью состоять из вещественных величин (без целых).

Величины в формате с плавающей точкой должны записываться как:

1) Десятичные цифры, сопровождаемые десятичной точкой и как минимум еще одной десятичной цифрой, или,
2) Десятичные цифры, сопровождаемые “e” (экспонента) и целая величина показателя экспоненты, или,
3) Комбинация 1 и 2.

Далее приведены примеры констант:

`0.5`	величина с плавающей точкой
`1.0`	величина с плавающей точкой
`3.14`	величина с плавающей точкой
`1e16`	величина с плавающей точкой
`51.025e5`	величина с плавающей точкой
`3 + 4`	целая величина
`3.0 + 4.0`	выражение с плавающей точкой
`3.0 + 4`	неверное выражение, приведет к ошибке компиляции
`3.0 + FLOAT(4)`	выражение с плавающей точкой

Приведем пример, в котором объявляется одна целая и две вещественные константы.

```
CON
  Num1 = 20
  Num2 = 127.38
  Num3 = 32.05 * 18.1 - Num2 / float(Num1)
```

В приведенном примере константы `Num1`, `Num2` и `Num3` устанавливаются соответственно в 20, 127.38 и 573.736. Отметьте, что в последнем выражении константа `Num1` должна быть заключена в объявлении **`FLOAT`** с тем, чтобы компилятор трактовал ее как величину в формате с плавающей точкой.

Компилятор Propeller рассматривает константы с плавающей точкой как вещественные числа одинарной точности, как описано в стандарте IEEE-754. Вещественные числа одинарной точности хранятся в 32 битах, с 1 битом на знак, 8-битной экспонентой и 23-битной мантиссой (дробная часть). Это обеспечивает примерно 7.2 значащих десятичных разряда.

Для выполнения операций с float-числами, объекты FloatMath, FloatString, Float32 и Float32Full предоставляют математические функции, совместимые с числами одинарной точности.

Для детальной информации см. FLOAT на стр. 253, ROUND на стр. 351, TRUNC на стр. 363, а так же объекты FloatMath, FloatString, Float32 и Float32Full.

Перечисления (Синтаксис 2 и 3)

Блоки констант могут также объявлять перечисляемые константны. Перечисления – это логически сгруппированные идентификаторы, которые имеют постоянное целое значение инкремента, сопоставленное группе и уникальное для каждой из групп. Например, объекту могут быть необходимы определенные режимы работы. Каждый из этих режимов может быть идентифицирован номером, например 0, 1, 2 и 3. Сами значения чисел на самом деле для нашей задачи не важны, они просто должны быть уникальными в рамках каждого режима работы. Поскольку номера, как таковые, не несут описательной нагрузки, нам тяжело вспомнить, что делает, к примеру, режим 3, и было бы намного легче, если бы он имел описательное имя. Посмотрите на следующий пример.

```
CON
  'Declare modes of operation
  RunTest    = 0
  RunVerbose = 1
  RunBrief   = 2
  RunFull    = 3
```

Приведенный пример решает нашу задачу; теперь для задания необходимого режима работы пользователи нашего объекта смогут указать “RunFull” вместо “3”. Однако проблема в том, что определение логической группы элементов таким способом может привести к ошибкам и сложностям в сопровождении, потому как при изменении любого значения (преднамеренно или же случайно) без соответствующего изменения остальных, может привести к неверной работе приложения. Представьте также случай, когда было бы необходимо 20 режимов работы. Это привело бы к намного более

длинному набору констант, а также к еще большей вероятности получить ошибки при сопровождении.

Перечисления решают эти проблемы путем автоматического инкрементирования значений констант. Мы можем переписать предыдущий пример с использованием перечислений таким образом:

```
CON 'Declare modes of operation
  #0, RunTest, RunVerbose, RunBrief, RunFull
```

Здесь `#0` указывает компилятору начинать счет с числа 0, и он устанавливает следующую константу в это значение. Далее любые дополнительные константы, у которых явно не указано значение (посредством '= выражение'), автоматически устанавливаются равными предыдущему значению плюс 1. В результате `RunTest` равна 0, `RunVerbose` равна 1, `RunBrief` равна 2 и `RunFull` равна 3. Для большинства случаев сами величины не важны; важно то, что каждый из идентификаторов имеет уникальное значение. Задание перечислимых величин подобным образом имеет то преимущество, что присвоенные значения уникальны и последовательны в рамках группы.

Используя приведенный выше пример, использующие его методы могут выполнять следующие действия (считаем, что `Mode` установлен вызывающим объектом):

```
 case Mode
   RunTest    : <test code here>
   RunVerbose : <verbose code here>
   RunBrief   : <brief code here>
   RunFull    : <full code here>
```

—или—

```
 if Mode > RunVerbose
   <brief and run mode code here>
```

Отметьте, что эти подпрограммы не зависят от конкретного значения переменной `Mode`, скорее они зависят от положения самого перечислимого идентификатора режима по отношению к остальным элементам этой группы. Важно писать код именно таким образом, чтобы уменьшить потенциальную возможность возникновения ошибки при введении изменений в будущем.

Также нужно отметить, что перечисления не обязательно должны состоять из разделенного запятыми списка элементов. Следующий далее пример также рабочий, но в его коде справа оставлено место для ввода комментариев о каждом режиме работы.

```
CON 'Declare modes of operation
  #0
  RunTest 'Run in test mode
  RunVerbose 'Run in verbose mode
  RunBrief 'Run with brief prompts
  RunFull 'Run in full production mode
```

Приведенный пример выполняет те же действия, что и предыдущий однострочковый вариант, однако сейчас мы имеем достаточно места для описания каждого из режимов без потери преимущества в автоматическом инкременте. Позже, если возникнет необходимость добавить пятый режим, просто добавьте его в список в любом удобном для Вас месте. Если возникнет необходимость в задании определенного значения, с которого должен начинаться список, просто измените `#0` на нужное Вам число: `#1`, `#20`, и т.д.

Существует даже возможность изменить значение перечислимых элементов в середине списка.

```
CON
  'Declare modes of operation
  #1, RunTest, RunVerbose, #5, RunBrief, RunFull
```

Здесь `RunTest` и `RunVerbose` равны соответственно 1 и 2, а `RunBrief` и `RunFull` равны соответственно 5 и 6. Хотя этот метод и может быть удобным, но для поддержания хорошего стиля программирования он должен использоваться в исключительно редких случаях.

Более правильным путем для получения результата, аналогичного приведенному примеру, является включение опционального поля *Offset*. Код предыдущего примера мог быть записан следующим образом:

```
CON
'Declare modes of operation
#1, RunTest, RunVerbose[3], RunBrief, RunFull
```

Также, как и ранее, `RunTest` и `RunVerbose` равны соответственно 1 и 2. Символы ‘`[3]`’, следующие сразу после `RunVerbose` приводят к увеличению текущего значения перечислимой константы (2) на величину 3, перед следующей перечислимой константой. Результат этого – такой же, как и ранее: `RunBrief` и `RunFull` равны соответственно 5 и 6. Однако преимуществом этого метода является то, что в нем

устанавливается зависимость перечислимых констант друг от друга. Изменение начального значения в строке приведет к их всех соответствующему изменению. Например, изменение #1 на #4 приведет к изменению RunTest и RunVerbose на соответственно 4 и 5, а RunBrief и RunFull – на соответственно 8 и 9. В сравнении с этим, если бы в исходном примере #1 было изменено на #4, то как RunVerbose, так и RunBrief установятся в 5, возможно приводя к неверному выполнению кода, использующего эти константы.
Величина *Offset* может быть любой со знаком, но она влияет только на величину, следующую непосредственно за ней; перечисляемое значение после идентификатора *Symbol*, возле которого не указано поле *Offset*, всегда инкрементируется на 1. Если желательно иметь пересекающиеся значения, этого можно достичь указанием значения *Offset* равным 0 или менее.
Синтаксис 3 – это вариант синтаксиса перечислений. В нем не указывается никакого начального значения. Любые идентификаторы, определенные таким способом, будут всегда начинаться либо с ноля 0 (для нового блока **CON**), либо со следующего перечисляемого значения по отношению к предыдущему (в рамках существующего блока **CON**).

Область видимости констант

Символьные константы, определенные в Блоках Констант, являются глобальными для объекта, в котором они определены, но не за его границами. Это означает, что константы могут быть напрямую доступны из любого места в рамках своего объекта, и, в то же время, их имена не будут конфликтовать с такими же, объявленными в родительском либо дочернем объекте.

Однако символьные константы могут также быть косвенно доступны и родительским объектам, при использовании синтаксиса ссылок на константы. Пример:

```
OBJ
  Num : "Numbers"
PUB SomeRoutine
  Format := Num#DEC 'Set Format to Number's Decimal constant
```

В этом примере объект “Numbers” ассоциирован с идентификатором Num. Далее метод обращается к константе DEC этого объекта как Num#DEC. Здесь Num – это ссылка на объект, ‘#’ — указывает, что нам необходим доступ к константам этого объекта, а DEC – это константа в рамках необходимого нам объекта. Это свойство позволяет объектам определять константы для их собственного использования, а также предоставляет свободный доступ к ним родительских объектов, исключая необходимость объявления родительскими объектами собственных идентификаторов для связи с ними.

CONSTANT

Директива: Объявляет однострочное выражение-константу, вычисляемое при компиляции.

((PUB ¦ PRI))
CONSTANT (*ConstantExpression*)

Возвращает: Результат решения выражения-константы.

- ***ConstantExpression*** – желаемое выражение-константа.

Описание

Блок CON может использоваться для задания выражений, на которые ссылаются из различных мест кода, в виде констант; однако существуют ситуации, когда выражение-константа необходимо для временных, одноразовых целей. Директива CONSTANT используется для полного вычисления константного однострокового выражения в методе при компиляции. Без использования директивы CONSTANT, однострочковые выражения метода всегда решаются во время выполнения, даже если выражение – это постоянная величина.

Использование CONSTANT

Директива CONSTANT создает выражения-константы одноразового использования, которые позволяют сэкономить в размере кода и выиграть в скорости выполнения. Обратите внимание на два примера, приведенные ниже:

Пример 1, использующий стандартные, вычисляемые при выполнении, выражения:

```
CON
  X = 500
  Y = 2500

PUB Blink
  !outa[0]
  waitcnt(X+200 + cnt)                'Standard run-time expression
  !outa[0]
  waitcnt((X+Y)/2 + cnt)              'Standard run-time expression
```

Пример 2, такой же, как и выше, но с директивой CONSTANT, заключающей константные, вычисляемые при компиляции, выражения:

```
CON
  X = 500
  Y = 2500

PUB Blink
  !outa[0]
  waitcnt(constant(X+200) + cnt)     'exp w/compile & run-time parts
  !outa[0]
  waitcnt(constant((X+Y)/2) + cnt)   'exp w/compile & run-time parts
```

Эти два примера выполняют абсолютно одинаковые действия: их методы `Blink` переключают `P0`, ожидают `X+200` циклов, вновь переключают `P0` и ожидают еще `(X+Y)/2` циклов перед возвратом. В то время, как идентификаторы `X` и `Y` блока **CON** могут использоваться во многих местах объекта, команды **WAITCNT**, использованные в методе `Blink` каждого примера, возможно, используются только в этом одном месте. По этой причине, скорее всего, нет смысла создавать дополнительные константы в блоке **CON** для таких выражений, как `X+200` и `(X+Y)/2`. Хотя и нет ничего страшного в использовании выражений непосредственно в исполнимом виде, как в примере 1, однако тогда все это выражение будет вычисляться в процессе выполнения приложения, требуя дополнительного времени и объема памяти.

Директива **CONSTANT** прекрасно подходит для данной ситуации, потому как она полностью вычисляет каждое однократно используемое константное выражение в простое, статическое значение, сохраняя память программ и убыстряя выполнение. В примере 1, метод `Blink` требует 33 байта кода, в то время как этот же метод примера 2, с добавленными директивами **CONSTANT** требует всего 23 байта памяти. Отметьте, что части выражений "`+ cnt`" не включены в скобки директивы **CONSTANT**; так сделано из-за того, что `cnt` – это переменная (регистр Системного Счетчика; см. **CNT**, стр. 215), поэтому ее значение не может быть использовано во время компиляции.

Если константа должна использоваться более, чем в одном месте кода, лучше определить ее в блоке **CON**, тогда она будет определена всего однажды, а идентификатор, представляющий ее значение, может быть использован много раз.

Предопределенные Константы

Далее приведен список предопределенных в компиляторе констант:

TRUE	Логическая истина:	-1	($FFFFFFFF)
FALSE	Логическая ложь:	0	($00000000)
POSX	Макс. положительное целое:	2147483647	($7FFFFFFF)
NEGX	Макс. отрицательное целое:	-2147483648	($80000000)
PI	Float-величина PI:	≈ 3.141593	($40490FDB)
RCFAST	Внутр. быстрый генератор:	$00000001	(%00000000001)
RCSLOW	Внутр. медленный генератор:	$00000002	(%00000000010)
XINPUT	Внешн. частота/генератор:	$00000004	(%00000000100)
XTAL1	External low-speed crystal:	$00000008	(%00000001000)
XTAL2	Внешн. среднечастотный резонатор:	$00000010	(%00000010000)
XTAL3	Внешн. высокочастотный резонатор:	$00000020	(%00000100000)
PLL1X	Множитель внешн. частоты 1:	$00000040	(%00001000000)
PLL2X	Множитель внешн. частоты 2:	$00000080	(%00010000000)
PLL4X	Множитель внешн. частоты 4:	$00000100	(%00100000000)
PLL8X	Множитель внешн. частоты 8:	$00000200	(%01000000000)
PLL16X	Множитель внешн. частоты 16:	$00000400	(%10000000000)

(Все эти константы также доступны в ассемблере Propeller.)

TRUE и FALSE

TRUE и FALSE обычно используются для логического сравнения величин:

```
if (X = TRUE) or (Y = FALSE)
 <code to execute if total condition is true>
```

POSX и NEGX

POSX и NEGX обычно используются в целях сравнения или как флаг особого события:

```
if Z > NEGX
 <code to execute if Z hasn't reached smallest negative>
```

—или—

```
PUB FindListItem(Item) : Index
  Index := NEGX 'Default to "not found" response
  <code to find Item in list>
  if <item found>
  Index := <items index>
```

PI

PI может использоваться при вычислениях с вещественными числами, как float-констант, так и float-переменных, с применением объектов FloatMath и FloatString.

Константы от RCFAST до PLL16X

Константы от RCFAST до PLL16X – это константы установок режима работы системного генератора. Они подробно описаны в секции _CLKMODE , начинающейся со стр. 209.

Отметьте, что они являются перечисляемыми константами и не эквивалентны соответствующему значению регистра CLK. См. «Регистр CLK», стр. 33, для получения информации касательно того, как каждая из констант соотносится с битами регистра CLK.

CTRA, CTRB

Регистр: Регистры управления Счетчика A и Счетчика B.

((PUB ¦ PRI))
  CTRA

((PUB ¦ PRI))
  CTRB

Возвращает: Текущее значение регистров управления Счетчика A или Счетчика B, если используется как переменная-источник.

Описание

CTRA и CTRB - это два из шести регистров (CTRA, CTRB, FRQA, FRQB, PHSA, и PHSB), которые влияют на работу Модулей Счетчиков каждого из процессоров. Каждый *Cog* имеет два идентичных модуля счетчиков (A и B), которые могут выполнять множество повторяющихся задач. Регистры CTRA и CTRB содержат установки конфигурации для соответственно модулей Счетчика A и Счетчика B.

В дальнейшем при обсуждении мы будем использовать идентификаторы CTRx, FRQx и PHSx для обращения к обоим (A и B) парам каждого из регистров.

Каждый из двух модулей счетчиков может управлять или контролировать до двух линий B/B и выполнять условное 32-битное накопление значения регистра FRQx в регистре PHSx по каждому такту системной частоты. Каждый Модуль Счетчика имеет свою собственную ФАПЧ (PLLx), которая может быть использована для синтеза частот от 64 МГц до 128 МГц.

С простыми настройками и, в некоторых случаях, небольшим участием процессора, модули счетчиков могут использоваться для:

- Синтез частоты
- Измерение частоты
- Подсчет импульсов
- Измерение ширины импульсов
- Измерение состояния неск. линий
- Широтно-имп. модуляция (ШИМ)
- Измерение duty-cycle
- Цифро-аналог. преобразов. (ЦАП)
- Аналог-цифр. преобразов. (АЦП)
- И другое.

Для некоторых из этих режимов *Cog* устанавливает конфигурацию счетчиков посредством CTRA или CTRB, и они выполняют свою задачу полностью независимо. Для

других режимов *Cog* может использовать WAITCNT для временной синхронизации данных чтения и записи со счетчиков в цикле, создавая более сложный автомат состояний. Поскольку период обновления счетчика может быть коротким (12.5 нс при 80 МГц), возможны генерация и измерение сигналов с большой динамикой.

Поля Регистров Управления

Каждый из регистров CTRA и CTRB содержит четыре поля, представленных в Табл. 4.5.

Табл. 4-5: Регистры CTRA и CTRB

31	30..26	25..23	22..15	14..9	8..6	5..0
-	CTRMODE	PLLDIV	-	BPIN	-	APIN

APIN
Поле APIN регистра CTRx выбирает основную линию B/B для этого счетчика. Это поле может быть игнорировано, если линия не используется. %0xxxxx = Порт A, %1xxxxx = Порт B (зарезервировано). В ассемблере Propeller, поле APIN удобно записывать с использованием инструкции MOVS.

Отметьте, что запись ноля в CTRA моментально выключает Счетчик A, а также останавливает весь связанный с ним вывод и аккумуляцию PHSA.

BPIN
Поле BPIN регистра CTRx выбирает дополнительную линию B/B для этого счетчика. Это поле может быть игнорировано, если линия не используется. %0xxxxx = Порт A , %1xxxxx = Порт B (зарезервировано). В ассемблере Propeller, поле BPIN удобно записывать с использованием инструкции MOVD.

PLLDIV
Поле PLLDIV регистра CTRx выбирает отвод множителя ФАПЧ, см. таблицу ниже. Этим определяется, на какую степень двойки делится частота VCO для использования в качестве выходной частоты PLLx (диапазон от 500 кГц до 128 МГц). Это поле может быть игнорировано, если не используется. В ассемблере Propeller поле PLLDIV удобно записывать вместе с CTRMODE, используя инструкцию MOVI.

Табл. 4-6: Поле PLLDIV

PLLDIV	%000	%001	%010	%011	%100	%101	%110	%111
Выход	VCO ÷ 128	VCO ÷ 64	VCO ÷ 32	VCO ÷ 16	VCO ÷ 8	VCO ÷ 4	VCO ÷ 2	VCO ÷ 1

CTRMODE
Поле CTRMODE регистров CTRA и CTRB выбирает один из 32 рабочих режима, показанных в Табл. 4-7, соответственно для Счетчика A или Счетчика B. В ассемблере Propeller, поле CTRMODE удобно записывать вместе с PLLDIV, используя инструкцию MOVI.

В режимах от %00001 до %00011 происходит аккумуляция FRQx-в-PHSx каждый цикл системной частоты. Таким образом получается цифро-задающий генератор (NCO) в PHSx[31], который запитывает опорный вход PLLx. ФАПЧ (PLLx) умножает эту частоту на 16, используя свой генератор, управляемый напряжением (VCO).

Для стабильной работы рекомендуется держать частоту VCO в границах от 64 МГц до 128 МГц. Это даст частоту NCO от 4 МГц до 8 МГц.

Использование CTRA и CTRB

В языке *Spin* регистры CTRx могут быть прочитаны/записаны так же, как и любые другие регистры или предопределенные переменные. Как только этот регистр записан, счетчик переходит в новый режим работы. Например:

```
CTRA := %00100 << 26
```

Этот код устанавливает поле CTRMODE регистра CTRA в режим NCO (%00100), а все остальные биты – в ноль.

Табл. 4-7: Режимы работы счетчика (значения поля CTRMODE)				
CTRMODE	**Описание**	**Накопление FRQx в PHSx**	**Выход APIN***	**Выход BPIN***
%00000	Счетчик отключен (off)	0 (никогда)	0 (нет)	0 (нет)
%00001 %00010 %00011	PLL внутренн. (видео режим) PLL одноканальный PLL дифференциальный	1 (always) 1 1	0 PLLx PLLx	0 0 !PLLx
%00100 %00101	NCO одноканальный NCO дифференциальный	1 1	PHSx[31] PHSx[31]	0 !PHSx[31]
%00110 %00111	DUTY одноканальный DUTY дифференциальный	1 1	PHSx-Carry PHSx-Carry	0 !PHSx-Carry
%01000 %01001 %01010 %01011	POS детектор POS детектор с обратной связью (OC) POSEDGE детектор POSEDGE детектор с OC	A^1 A^1 A^1 & $!A^2$ A^1 & $!A^2$	0 0 0 0	0 $!A^1$ 0 $!A^1$
%01100 %01101 %01110 %01111	NEG детектор NEG детектор с OC NEGEDGE детектор NEGEDGE детектор с OC	$!A^1$ $!A^1$ $!A^1$ & A^2 $!A^1$ & A^2	0 0 0 0	0 $!A^1$ 0 $!A^1$
%10000 %10001 %10010 %10011 %10100 %10101 %10110 %10111 %11000 %11001 %11010 %11011 %11100 %11101 %11110 %11111	ЛОГИЧЕСКОЕ никогда ЛОГИЧЕСКОЕ !A & !B ЛОГИЧЕСКОЕ A & !B ЛОГИЧЕСКОЕ !B ЛОГИЧЕСКОЕ !A & B ЛОГИЧЕСКОЕ !A ЛОГИЧЕСКОЕ A <> B ЛОГИЧЕСКОЕ !A | !B ЛОГИЧЕСКОЕ A & B ЛОГИЧЕСКОЕ A == B ЛОГИЧЕСКОЕ A ЛОГИЧЕСКОЕ A | !B ЛОГИЧЕСКОЕ B ЛОГИЧЕСКОЕ !A | B ЛОГИЧЕСКОЕ A | B ЛОГИЧЕСКОЕ всегда	0 $!A^1$ & $!B^1$ A^1 & $!B^1$ $!B^1$ $!A^1$ & B^1 $!A^1$ A^1 <> B^1 $!A^1$ | $!B^1$ A^1 & B^1 A^1 == B^1 A^1 A^1 | $!B^1$ B^1 $!A^1$ | B^1 A^1 | B^1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

***Должен установить соответствующий бит DIR для влияния на пин**
A^1 = вход APIN, задержанный на 1 цикл
A^2 = вход APIN задержанный на 2 цикла
B^1 = вход BPIN задержанный на 1 цикл

DAT

Объявление: Объявляет блок данных .

DAT

⟨***Symbol***⟩ ***Alignment*** ⟨***Size***⟩ ⟨***Data***⟩ ⟨[***Count***]⟩ ⟨**,** ⟨***Size***⟩ ***Data*** ⟨[***Count***]⟩⟩...

DAT

⟨***Symbol***⟩ ⟨***Condition***⟩ ***Instruction Operands*** ⟨***Effect(s)***⟩

- ***Symbol*** – опциональное имя для данных, зарезервированной области, либо следующей далее инструкции.
- ***Alignment*** – желаемое выравнивание и размер (BYTE, WORD, или LONG) следующих далее элементов данных.
- ***Size*** – желаемый размер (BYTE, WORD, или LONG) следующих далее элементов данных; выравнивание не изменяется.
- ***Data*** – константа, либо разделенный запятыми список констант. Также допускаются заключенные в кавычки строки символов; они рассматриваются как список символов, разделенный запятыми.
- ***Count*** – опциональное выражение, отображающее количество экземпляров *byte*-, *word*-, либо *long*-данных *Data*, сохраняемых в таблице.
- ***Condition*** – условный оператор языка ассемблера: IF_C, IF_NC, IF_Z, и т.д.
- ***Instruction*** – инструкция ассемблера: ADD, SUB, MOV, и т.д., и все ее операнды.
- ***Operands*** – ноль, один либо два операнда, как указано в *Instruction*
- ***Effect(s)*** – один, два или три эффекта языка ассемблера, которые приведут, (либо – нет) к записи результата выполнения инструкции NR, WR, WC, или WZ.

Описание

Блок данных представляет собой секцию исходного кода, которая может содержать предопределенные данные, зарезервированную для использования при выполнении память, а так же код языка Propeller ассемблер. Идентификатор DAT – это одно из шести специальных объявлений (CON, VAR, OBJ, PUB, PRI, и DAT), которые обеспечивают четкую структуру языка *Spin*.

Блоки данных – это многофункциональные секции кода, которые используются для размещения таблиц данных, рабочего места для выполнения, и ассемблерного кода. При необходимости, ассемблерный код и данные могут быть смешанными между собой, таким образом данные будут загружаться в *Cog* совместно с кодом ассемблера.

Объявление Данных (Синтаксис 1)

Блок данных объявляется с заданным выравниванием и размером данных (BYTE, WORD, или LONG) для указания того, как они должны располагаться в памяти. Где на самом деле расположены данные, зависит от структуры объекта и приложения, в которое он откомпилирован, поскольку данные включаются как часть откомпилированного кода.

Например:

```
DAT
  byte 64, "A", "String", 0
  word $FFC2, 75000
  long $44332211, 32
```

Первое объявление на второй строке этого примера, BYTE, указывает, что данные, следующие далее, должны быть байт-выровнены и байт-размерные. Во время компиляции, данные, следующие за BYTE, 64, "A", и т.д., сохраняются в памяти программ, байт за байтом, начиная со следующего свободного адреса. Третья строка задает данные размером в слово и выравниванием в слово. Ее данные, $FFC2 и 75000, начнутся с позиции следующей границы выравнивания по размеру слова, следом за предшествующими данными; все неиспользованные байты после предыдущих данных будут дополнены нолями до достижения следующей границы слова. Четвертая строка задает данные размером в двойное слово, и выровненные по границе двойного слова; ее данные будут сохранены, начиная со следующей границы выравнивания по двойному слову, следом за предыдущими данными, с добавлением нулей в незанятые ячейки до достижения этой границы. В Табл. 4-8 показано, как это выглядит в памяти (показано в шестнадцатеричной системе).

Табл. 4-8: Распределение данных в памяти (из примера)

L	0				1				2				3				4				5			
W	0		1		2		3		4		5		6		7		8		9		10		11	
B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
D	40	41	53	74	72	69	6E	67	00	00	C2	FF	F8	24	00	00	11	22	33	44	20	00	00	00

L = longs(двойные слова), W = words (слова), B = bytes (байты), D = data (данные)

Первые девять байтов (0 – 8) – это однобайтовые данные из первой строки; $40 = 64 (десятичное), $41 = "A", $53 = "S", и т.д. Байт 9 установлен в ноль для выравнивания первого слова из данных, выровненных по слову и размером в слово, $FFC2, по границе слова - байту 10. Байты 10 и 11 (слово 5) содержат первое значение размера слово, $FFC2, сохраненное в формате little-endian (меньший байт по меньшему адресу) как $C2 и $FF. Байты 12 и 13 (слово 6) – это меньшее слово от 75000, об этом позднее.

Байты 14 и 15 (слово 7) – это добавленные ноли для выравнивания первого значения с размером двойного слова, $44332211. Байты от 16 до 19 (двойное слово 5) содержат это значение в формате little-endian. В завершение, байты с 20 по 23 (двойное слово 6) содержат второе двойное слово данных, 32 формате little-endian.

Вы могли заметить, что значение 75000 было задано как значение размером в слово. Число 75000 в шестнадцатеричной системе равно $124F8, и поскольку оно больше слова, было сохранено лишь его младшее слово ($24F8). В результате слово 6 (байты 12 и 13) содержат $F8 и $24, а слово 7 (байты 14 и 15) содержат $00 и $00, добавленные для достижения границы двойного слова.

Такое же поведение, намеренно или нет, наблюдается также и с данными размером в байт и выровненными в байт, например:

```
DAT
  byte $FFAA, $BB995511
```

...дает в результате сохранение только младших байтов каждого значения, — $AA и $11 сохраняются в последовательных ячейках.

Иногда, все же, желательно сохранять всё большое значение как более маленькие элементарные части, которые совсем не обязательно должны быть выровнены по размеру самой величины. Чтобы реализовать подобное, задайте размер величины перед самим ее значением.

```
DAT
  byte word $FFAA, long $BB995511
```

В этом примере задано байтовое выравнивание, а величины для сохранения заданы размером в слово и двойное слово. В результате в памяти будут находиться все байты величин: сначала последовательно $AA и $FF, а затем $11, $55, $99 и $BB.

Если бы мы изменили третью строку первого примера, приведенного выше, таким образом:

```
  word $FFC2, long 75000
```

...то мы бы получили $F8, $24, $01, и $00 расположенные в байтах от 12 до 15. Байт 15 – это старший байт числа, и так просто совпало, что он расположен сразу слева от границы следующего двойного слова, поэтому дополнительные байты для выравнивания следующего числа с размером в двойное слово, не понадобились.

При желании, для задания данным символического имени, может быть использовано поле *Symbol* синтаксиса 1. Это облегчает обращение к данным из блоков **PUB** или **PRI**. Например:

```
DAT
  MyData byte $FF, 25, %1010

PUB GetData | Temp
  Temp := MyData[0]                'Get first byte of data table
```

В этом примере создается таблица данных с именем MyData, которая состоит из байтов $FF, 25 и %1010. Public-метод GetData читает первый байт MyData из основной памяти и сохраняет его в своей локальной переменной Temp.

Вы также можете использовать объявления **BYTE**, **WORD**, и **LONG** для чтения из ячеек основной памяти. Например:

```
DAT
  MyData byte $FF, 25, %1010

PUB GetData | Temp
  Temp := BYTE[@MyData][0]         'Get first byte of data table
```

Этот пример похож на предыдущий, за одним исключением – он использует объявление **BYTE** для чтения значения, расположенного по адресу MyData. Для более детальной информации о чтении и записи основной памяти см. **BYTE**, стр. 192; **WORD**, стр. 382; и **LONG**, стр. 274.

Объявление повторяющихся данных (Синтаксис 1)

Элементы данных могут повторяться при использовании опционального поля *Count*:

```
DAT
  MyData      byte  64, $AA[8], 55
```

В приведенном примере объявляется байт-выровненная таблица однобайтовых величин с именем MyData, состоящая из следующих десяти значений: 64, $AA, $AA, $AA, $AA, $AA, $AA, $AA, $AA, 55. Значение $AA повторялось 8 раз из-за наличия символов ‘[8]’, следующих в объявлении сразу за этим значением.

Написание кода Propeller ассемблер (Синтаксис 2)

Вдобавок к численным и строковым данным, блок DAT используется также и для написания в нем кода Propeller ассемблер. В следующем примере производится переключение линии 0 каждую ¼ часть секунды.

```
DAT
                org 0                           'Reset assembly pointer
Toggle          rdlong  Delay, #0               'Get clock frequency
                shr     Delay, #2               'Divide by 4
                mov     Time,  cnt              'Get current time
                add     Time,  Delay            'Adjust by 1/4 second
                mov     dira,  #1               'set pin 0 to output
Loop            waitcnt Time,  Delay            'Wait for 1/4 second
                xor     outa,  #1               'toggle pin
                jmp     #Loop                   'loop back

Delay   res     1
Time    res     1
```

При начальной загрузке приложения может выполняться только код *Spin*. Однако, в любой момент времени, *Spin*–программа может запустить в своем процессоре выполнение кода ассемблера. Для этого используются команды COGNEW (стр. 221) и COGINIT (стр. 218). Далее приведен пример, в котором *Spin*-код запускает ассемблерную подпрограмму Toggle, рассмотренную выше.

```
PUB Main
  cognew(@Toggle, 0)                            'Launch Toggle code
```

Здесь инструкция COGNEW указывает ИМС Propeller запустить ассемблерную подпрограмму Toggle на выполнение в новом процессоре. В результате ИМС Propeller находит следующий доступный *cog*, копирует код из блока DAT, начиная с позиции Toggle, в его ОЗУ, после чего запускает этот процессор, который начинает выполнение кода с позиции 0 своего ОЗУ.

Блок DAT может содержать несколько ассемблерных подпрограмм, так же несколько блоков DAT могут каждый содержать свою ассемблерную подпрограмму, но в обоих случаях, каждая ассемблерная подпрограмма должна начинаться с директивы ORG (стр. 488) для корректного сброса указателя на адрес выполняемого ассемблерного кода.

Двойные команды

Языки *Spin* и *Propeller*-ассемблер имеют набор созвучных команд, называемых двойными командами. Эти двойные команды выполняют одинаковые действия, однако имеют различную структуру синтаксиса, которая зависит от структуры языка, которому принадлежит данная команда — *Spin* либо ассемблеру. Любая двойная команда, используемая в блоке DAT, рассматривается как команда ассемблера. И наоборот, любая двойная команда, используемая в блоке PUB или PRI, рассматривается как команда языка *Spin*.

DIRA, DIRB

Регистр: Регистр направления для 32-битных портов Port A и Port B.

((PUB ¦ PRI))
DIRA 〈[*Pin(s)*]〉

((PUB ¦ PRI))
DIRB 〈[*Pin(s)*]〉 (Зарезервирован)

Возвращает: Текущее значение битов направления для линий B/B в портах Port A или Port B, если используется как переменная-источник.

- ***Pin(s)*** – это опциональный параметр либо параметр типа «диапазон», который задает линию(линии) B/B для доступа в порту Port A (0-31) или порту Port B (32-63). Если приведен в виде одиночного выражения, доступ производится только к указанной линии. Если приведен в виде диапазона (два параметра в формате диапазона: x..y) то доступ происходит к смежным линиям в диапазоне, начало и конец которого соответствуют указанным параметрам.

Описание

Регистры DIRA и DIRB входят в состав шести регистров (DIRA, DIRB, INA, INB, OUTA и OUTB), которые напрямую управляют линиями B/B. Регистр DIRA содержит направления для каждой из 32 линий B/B порта Port A; биты от 0 до 31 соответствуют линиям от P0 до P31. Регистр DIRB содержит направления для каждой из 32 линий B/B порта Port B; биты от 0 до 31 соответствуют линиям от P32 до P63.

ПРИМЕЧАНИЕ: DIRB зарезервирован для использования в будущем; ИМС Propeller P8X32A не содержит линий B/B порта Port B, поэтому ниже обсуждается только DIRA.

DIRA используется как для установки, так и для чтения текущего направления для одной или более линий B/B в порту Port A. Сброс бита в ноль (0) устанавливает направление соответствующей линии на ввод. Установка же бита в единицу (1) устанавливает направление соответствующей линии B/B на вывод. Исходное состояние регистра DIRA при запуске процессора – сброшенное, все биты равны нолю, все линии B/B при этом заданы как входы до тех пор, пока исполняемый процессором код не установит их иначе.

Каждый *Cog* имеет доступ ко всем линиям B/B в любой момент времени. Все линии непосредственно подключены к каждому процессору, поэтому *Hub* никакого влияния на доступ к линиям не оказывает. Каждый *Cog* содержит свой собственный регистр

DIRA, который предоставляет ему возможность установить направление любой из линий B/B. Регистр DIRA каждого процессора складывается по ИЛИ с таковыми у остальных процессоров, и результирующее 32-битное значение задает направление линий с P0 по P31 порта Port A. В результате направление каждой линии B/B – это "монтажное-ИЛИ" всей группы процессоров. Для более подробной информации см. «Линии B/B» на стр. 30.

Эта конфигурация может быль легко описана с помощью простых правил:

A. Линия является входом только если не один из *Cog* не установил ее на выход.
B. Линия является выходом если хотя бы один из *Cog* установил ее на выход.

Если *Cog* выключен, то его регистр направления сброшен в ноль, что исключает его влияние на направление и состояние линий B/B.

Заметьте, что из-за природы реализации линий B/B в виде "монтажного ИЛИ", электрическое соединение между процессорами отсутствует, хотя они все же могут осуществлять одновременный доступ к линиям B/B. На разработчика ложится ответственность убедиться в отсутствии коллизий на одной и той же линии B/B в процессе выполнения приложения.

Использование DIRA

Для изменения направления линий B/B необходимо установить либо сбросить соответствующие биты в регистре DIRA. Например:

```
DIRA := %00000000_00000000_10110000_11110011
```

Приведенный выше код устанавливает весь регистр DIRA (все 32 бита одновременно) в значение, которое устанавливает линии B/B 15, 13, 12, 7, 6, 5, 4, 1 и 0 как выходы, а остальные – как входы.

Используя унарные операторы *post*-очистки(~) и *post*-установки (~~), *Cog* может установить все линии B/B соответственно как входы или выходы; однако устанавливать все линии B/B как выходы обычно нежелательно. Например:

```
DIRA~                   'Clear DIRA register (all I/Os are inputs)
```

—и—

```
DIRA~~                  'Set DIRA register (all I/Os are outputs)
```

Первый приведенный пример очищает весь регистр DIRA (все 32бита сразу) в ноль, устанавливая все линии от P0 до P31 как входы. Второй же пример устанавливает все

биты регистра **DIRA** (все 32 бита сразу) в единицы; все линии В/В с P0 по P31 становятся выходами.

Для влияния только на одну линию В/В (один бит), добавьте опциональное поле *Pin(s)*. Здесь регистр **DIRA** рассматривается как массив из 32 бит.

```
DIRA[5]~~               'Set DIRA bit 5 (P5 to output)
```

Этот код устанавливает P5 на вывод. Все остальные биты регистра **DIRA** (следовательно и все соответствующие линии В/В) остаются в прежнем состоянии.

Регистр **DIRA** поддерживает специальный формат выражений, называемый ‘выражения-диапазон’, которые позволяют влиять одновременно на целую группу линий В/В, не влияя на остальные, не входящие в заданный диапазон. Для одновременного влияния на несколько смежных линий В/В, используйте выражение-диапазон в поле *Pin(s)*.

```
DIRA[5..3]~~            'Set DIRA bits 5 through 3 (P5-P3 to output)
```

Этот код устанавливает P5, P4 и P3 на вывод; все остальные биты регистра **DIRA** остаются в своем прежнем состоянии. Вот еще один пример:

```
DIRA[5..3] := %110      'Set P5 and P4 to output, P3 to input
```

Этот код устанавливает биты 5, 4 и 3 регистра **DIRA** равными соответственно 1, 1, и 0, не изменяя остальные. Следовательно, P5 и P4 становятся выходами, а P3 – входом.

ВАЖНО: Порядок величин в выражении-диапазоне влияет на результат. Например, далее в примере изменен порядок в выражении-диапазоне из предыдущего примера.

```
DIRA[3..5] := %110      'Set P3 and P4 to output, P5 to input
```

Биты 3, 4 и 5 **DIRA** установлены в 1, 1, и 0, переводя P3 и P4 на вывод, а P5 – на ввод.

Это очень мощное свойство диапазонных выражений, но если ему уделить недостаточно внимания, оно может привести к неожиданным результатам.

Обычно регистр **DIRA** только записывается, однако он также может быть прочитан для получения текущего значения направлений линий В/В. Далее в примере считается, что переменная `Temp` уже создана ранее в другом месте:

```
Temp := DIRA[7..4]      'Get direction of P7 through P4
```

Этот код устанавливает переменную `Temp` равной битам 7, 6, 5, и 4 регистра **DIRA;** т.е. младшие 4 бита `Temp` теперь равны `DIRA7:4`, а остальные биты `Temp` сброшены в ноль.

FILE

Директива: Импортирует внешний файл как данные.

```
DAT
  FILE "FileName"
```

- ***FileName*** – имя желаемого файла данных, без расширения. При компиляции производится поиск файла с этим именем в редактируемых вкладках, рабочей и библиотечной директории. *FileName* может содержать любые корректные символы имени файла, запрещенными являются символы \, /, :, *, ?, ", <, >, и |.

Описание

Директива FILE используется для импорта внешнего файла данных (обычно двоичного) в блок DAT объекта. Затем данные могут быть доступны объектом как любые обычные данные блока DAT.

Использование FILE

FILE используется в блоках DAT аналогично использованию BYTE, за исключением того, что его сопровождает имя файла в кавычках, а не значения данных. Например:

```
DAT
  Str    byte  "This is a data string.", 0
  Data   file "Datafile.dat"
```

В этом примере блок DAT состоит из строки байтов, сопровождаемой данными из файла с именем Datafile.dat. Во время компиляции программа *Propeller Tool* ищет среди редактируемых вкладок, а также в рабочей и библиотечной директориях, файл с именем Datafile.dat, и загружает его данные в память, начиная с первого байта, следующего за *zero-terminated* строкой Str. Методы могут получить доступ к импортированным данным, используя объявления BYTE, WORD или LONG, как для обычных данных. Например:

```
PUB GetData | Index, Temp
  Index := 0
  repeat
    Temp := byte[Data][Index++] 'Read data into Temp 1 byte at a time
    <do something with Temp>    'Perform task with value in Temp
  while Temp > 0                'Loop until end found
```

В этом примере импортированные данные читаются побайтно, пока не будет найден 0.

FLOAT

Директива: Преобразует целое выражение-константу в значение в формате с плавающей точкой, вычисляемое при компиляции.

((CON ¦ VAR ¦ OBJ ¦ PUB ¦ PRI ¦ DAT))
  FLOAT (*IntegerConstant*)

Возвращает: Результат вычисления выражения-константы как число в формате с плавающей точкой.

- ***IntegerConstant*** – желаемое целое выражение-константа, которое необходимо использовать как константу в формате с плавающей точкой.

Описание

FLOAT – одна из трех директив (FLOAT, ROUND и TRUNC), которые используются для выражений-констант в формате с плавающей точкой. Директива FLOAT преобразовывает константу целого типа в константу формата с плавающей точкой.

Использование FLOAT

Хотя большинство используемых констант представляют собой целые 32-битные значения, ИМС Propeller на уровне компилятора поддерживает также и 32-битные вещественные величины и константные выражения. Отметьте, что это справедливо только для констант, но не для переменных.

При объявлении вещественной константы, выражение должно быть введено как величина в формате с плавающей точкой, одним из трех путей: 1)Десятичные цифры, сопровождаемые десятичной точкой и как минимум еще одной десятичной цифрой, 2)десятичные цифры, сопровождаемые “e” (экспонента) и целая величина показателя экспоненты, 3)комбинация 1 и 2. Например:

```
CON
  OneHalf = 0.5
  Ratio = 2.0 / 5.0
  Miles = 10e5
```

Приведенный код создает три float-константы. Константа OneHalf равна 0.5, Ratio равна 0.4, а Miles равна 1000000.

Отметьте, что в приведенном примере каждый компонент каждого выражения задан как величина в формате float. Теперь посмотрите на следующий пример:

```
CON
  Two = 2
  Ratio = Two / 5.0
```

Здесь константа Two определена как целая, а Ratio, по-видимому, должна быть определена как float-константа. Однако такая запись приводит к ошибке на строке Ratio, поскольку при объявлении вещественных выражений-констант все величины в них должны быть тоже вещественными; нельзя смешивать целые и float- величины, как сделано в примере (Ratio = 2 / 5.0).

В подобной ситуации, для преобразования целого значения в float-величину, можно использовать директиву FLOAT, как показано ниже:

```
CON
  Two = 2
  Ratio = float(Two) / 5.0
```

Директива FLOAT в этом примере преобразует целую константу Two, в вещественную (float) величину, которая может использоваться в float-выражении.

О формате с плавающей точкой

В компиляторе Propeller константы с плавающей точкой представляются как вещественные числа одинарной точности, согласно стандарту IEEE-754. Вещественные числа одинарной точности хранятся в 32 битах, с 1 битом на знак, 8-битной экспонентой и 23-битной мантиссой (дробная часть). Это обеспечивает примерно 7.2 значащих десятичных разряда.

Для выполнения операций с float-числами, объекты FloatMath, FloatString, Float32 и Float32Full предоставляют математические функции, совместимые с числами одинарной точности. Для более подробной информации см. «Присваивание констант '='» в секции «Операторы *Spin*» на стр. 296, ROUND на стр. 351, и TRUNC на стр. 363, а также объекты FloatMath , FloatString, Float32, и Float32Full.

_FREE

Константа: Предопределенная, один раз устанавливаемая константа, используемая для задания размера свободного пространства памяти для приложения.

```
CON
  _FREE = Expression
```

- ***Expression*** – целое выражение, указывающее количество двойных слов для резервирования на свободное пространство.

Описание

_FREE – это предопределенная, один раз устанавливаемая опциональная константа, которая задает требуемый приложением объем свободной памяти. Эта величина добавляется к константе _STACK, если она используется, для определения общего объема свободной/стековой памяти на резервирование приложению Propeller. Используйте _FREE, если приложению для правильной работы требуется какой-то минимальный объем свободной памяти. Если откомпилированное приложение настолько большое, что не позволяет выделить заданный объем свободной памяти, будет выдано сообщение об ошибке. Например:

```
CON
  _FREE = 1000
```

Объявление _FREE в приведенном блоке CON указывает на то, что приложению необходимо, чтобы после компиляции осталось, как минимум, 1000 двойных слов свободной памяти. Если в результате откомпилированное приложение не резервирует такого количества свободной памяти, сообщение об ошибке укажет на то, на сколько превышен заданный объем. Это хороший путь для предотвращения неправильной работы успешно откомпилированных приложений, из-за нехватки памяти.

Помните, что только верхний объектный файл может устанавливать значение _FREE. Любое объявление _FREE в дочерних объектах будет игнорировано компилятором.

FRQA, FRQB

Регистр: Регистры частоты Счетчика A и Счетчика B .

((PUB ¦ PRI))
  FRQA

((PUB ¦ PRI))
  FRQB

Возвращает: Текущее значение частоты Счетчика A или Счетчика B, если используется как переменная-источник.

Описание

FRQA и FRQB - это два из шести регистров (CTRA, CTRB, FRQA, FRQB, PHSA, и PHSB), которые влияют на режим работы Модулей Счетчиков процессора. Каждый *Cog* имеет два идентичных модуля счетчиков (A и B), которые могут выполнять множество повторяющихся задач. Регистр FRQA содержит величину, которая аккумулируется в регистре PHSA. Регистр FRQB содержит величину, которая аккумулируется в регистре PHSB. Для более подробной информации см. CTRA, CTRB на стр. 239.

Использование FRQA и FRQB

FRQA и FRQB могут быть прочитаны/записаны так же, как любой другой регистр или предустановленная переменная. Например:

```
FRQA := $00001AFF
```

Приведенный код устанавливает FRQA в $00001AFF. В зависимости от поля CTRMODE регистра CTRA, это значение из FRQA может добавляться в регистр PHSA с частотой, определяемой частотой Системного Генератора и основной и/или вспомогательной линией B/B. Для более детальной информации см. CTRA, CTRB на стр. 239.

IF

Команда: Проверить условие(я) и выполнить блок кода, если верно (положительная логика).

((PUB ¦ PRI))
IF ***Condition(s)***
→ ***IfStatement(s)***
〈 ELSEIF ***Condition(s)***
→ ***ElseIfStatement(s)*** 〉...
〈 ELSEIFNOT ***Condition(s)***
→ ***ElseIfNotStatement(s)*** 〉...
〈 ELSE
→ ***ElseStatement(s)*** 〉

- ***Condition(s)*** – одно или более условий - логических выражений для проверки.
- ***IfStatement(s)*** – блок из одной или более строк кода для выполнения, если условие IF*Condition(s)* – истина.
- ***ElseIfStatement(s)*** – опциональный блок из одной или более строк кода для выполнения, если все предыдущие условия *Condition(s)* неверны, а условие ELSEIF *Condition(s)* – истина.
- ***ElseIfNotStatement(s)*** – опциональный блок из одной или более строк кода для выполнения, когда все предыдущие условия *Condition(s)* неверны, и условие ELSEIFNOT's *Condition(s)* – ложь.
- ***ElseStatement(s)*** – опциональный блок из одной или более строк кода для выполнения, если все предыдущие условия *Condition(s*) неверны.

Описание

IF - это одна из трех главных условных команд (IF, IFNOT, и CASE), которая производит условное выполнение блока кода. Команда IF может быть опционально скомбинирована с одной или более ELSEIF, одной или более ELSEIFNOT, и/или ELSE для формирования сложных условных структур.

Команда IF проверяет условия *Condition(s)* и, если результат – истина, выполняет блок *IfStatement(s)*. Если условия *Condition(s)* не верны, то проверяются по порядку условия следующего опционального ELSEIF *Condition(s)*, и/или ELSEIFNOT *Condition(s),* до тех пор, пока не будет найдено верное условие, после чего выполняется ассоциированный

с ним блок *ElseIfStatement(s)*, или *ElseIfNotStatement(s)*. Если ни одно из предыдущих условий не дало истину, выполняется опциональный блок *ElseStatement(s)*.

“Верное” условие – это то, которое дает при своем решении результат TRUE для позитивной логики (IF или ELSEIF), либо FALSE для отрицательной логики (ELSEIFNOT).

Отступы важны

ВНИМАНИЕ: Отступы важны! Язык *Spin* чувствителен к отступам (даже на один пробел) в строках, сопровождающих команды условного выполнения — для определения, принадлежат ли они блоку данной команды, или нет. Чтобы показать на экране программы *Propeller Tool* такие логически сгруппированные блоки кода, можно нажать *Ctrl+I* (включение индикаторов блок-групп). Повторное нажатие *Ctrl+I* отключит эту функцию. См. «Отступы и Выступы», стр. 84, и «Индикаторы Блок-Групп», стр. 89.

Простая форма IF

Наиболее общая форма условной команды IF выполняет действие, только лишь если условие верно. Это записывается как «IF условие», и далее одна или более форматированных строк кода. Например:

```
if X > 10               'If X is greater than 10
  !outa[0]              'Toggle P0
!outa[1]                'Toggle P1
```

В этом примере проверяется, что X больше, чем 10; если это истина, то переключается линия P0. Не зависимо от результата условия IF, затем переключается линия P1.

Поскольку строка !outa[0] введена с отступом от строки с IF, она принадлежит блоку *IfStatement(s)* и выполняется только если условие в IF верно. Следующая строка, !outa[1], введена без отступа от строки с IF, поэтому она выполняется независимо от результата вычисления условия в IF. Вот другой вариант этого же примера:

```
if X > 10               'If X is greater than 10
  !outa[0]              'Toggle P0
  !outa[1]              'Toggle P1
waitcnt(2_000 + cnt)    'Wait for 2,000 cycles
```

Этот вариант очень похож на предыдущий, за тем лишь исключением, что здесь две строки введены с отступом от строки с IF. В этом случае, если X больше, чем 10, переключится P0, затем переключится P1, и в конце выполнится строка waitcnt. Если, однако, X не был больше, чем 10, строки !outa[0] и !outa[1] пропускаются (поскольку

они являются частью блока *IfStatement(s)*) и выполняется строка waitcnt (она введена без отступа, поэтому не является частью блока *IfStatement(s)*).

Комбинирование условий

Поле *Condition(s)* вычисляется как одно простое логическое выражение, однако оно может быть составлено и из нескольких, связанных между собой операторами AND и OR; см. стр. 317-318. Например:

```
if X > 10 AND X < 100        'If X greater than 10 and less than 100
```

Здесь условие IF даст истину, если X больше, чем 10, и, в то же время, X меньше, чем 100. Другими словами, получим истину, если X находится в диапазоне от 11 до 99. Иногда подобные условия немного сложны для чтения. Для облегчения восприятия могут использоваться круглые скобки, используемые для группировки каждого из под-условий:

```
if (X > 10) AND (X < 100)    'If X greater than 10 and less than 100
```

Использование IF совместно с ELSE

Вторая наиболее общая форма использования IF выполняет одно действие, если условие верно, либо другое действие, если это условие неверно. Это записывается как «IF условие», далее его блок *IfStatement(s)*, затем ELSE с его блоком *ElseStatement(s)*:

```
if X > 100                   'If X is greater than 100
  !outa[0]                   'Toggle P0
else                         'Else, X <= 100
  !outa[1]                   'Toggle P1
```

Здесь если X больше, чем 100, переключается линия P0, иначе, когда X меньше или равен 100, переключается линия P1. Эта IF...ELSE конструкция, как видно, всегда выполняет переключение либо P0, либо P1, но никогда — обеих сразу и никогда — ни одной.

Помните, что код, который логически принадлежит блоку *IfStatement(s)* или *ElseStatement(s)*, должен быть введен с отступом как минимум на один пробел соответственно от IF или ELSE. Также отметьте, что ELSE должно находиться по вертикали прямо под IF; обе эти строки должны начинаться с одинаковой колонки, иначе компилятор не поймет, что данное ELSE относится к данному IF.

Для каждого условия IF может быть один либо ни одного компонента ELSE. Компонент ELSE должен быть последним в записи IF, он вводится после всех возможных условий ELSEIF.

Использование IF совместно с ELSEIF

Третья форма условной команды IF выполняет одно действие, если условие верно или другое действие, если первое условие не верно, но верно второе условие и т.д. Это записывается как условие IF со своим блоком *IfStatement(s)*, затем одно или более условий ELSEIF со своими соответствующими блоками *ElseIfStatement(s)*. Приведем пример:

```
if X > 100                      'If X is greater than 100
  !outa[0]                      'Toggle P0
elseif X == 90                  'Else If X = 90
  !outa[1]                      'Toggle P1
```

Здесь, если X больше, чем 100, переключается линия P0, иначе если X равен 90, переключается линия P1, и если ни одно из этих условий не верно, не переключается ни одна из линий. Это немного более короткий вариант записи такого кода:

```
if X > 100                      'If X is greater than 100
  !outa[0]                      'Toggle P0
else                            'Otherwise,
  if X == 90                    'If X = 90
    !outa[1]                    'Toggle P1
```

Оба этих примера выполняют одинаковые действия, но первый короче и обычно более легко читаемый. Отметьте, что ELSEIF, так же, как и ELSE, должно находиться под ассоциированным с ним IF (начинаться с той же колонки).

Каждая условная команда IF может иметь ноль и более условий ELSEIF, ассоциированных с ней. Посмотрите пример:

```
if X > 100                      'If X is greater than 100
  !outa[0]                      'Toggle P0
elseif X == 90                  'Else If X = 90
  !outa[1]                      'Toggle P1
elseif X > 50                   'Else If X > 50
  !outa[2]                      'Toggle P2
```

Здесь мы имеем три условия и три возможных действия. Так же, как и в предыдущем примере, если X больше, чем 100, переключается P0, иначе, если X равен 90, переключается P1, но если ни одно из условий не верно, а X больше, чем 50, переключается P2. Если ни одно из условий не верно, эти действия не произойдут.

В этом примере отражена важная концепция. Если X равен 101 или более, переключается P0, или если X равен 90, переключается P1, или если X от 51 до 89 или от 91 до 100, переключается P2. Так происходит из-за того, что условия **IF** и **ELSEIF** проверяются по одному по очереди, в которой они перечислены и будет выполнен только блок первого найденного условия, давшего истину; после этого никакие условия этой группы не проверяются. Это значит, что если бы мы перегруппировали два **ELSEIF** таким образом, что "X > 50" проверялось бы первым, мы бы получили ошибку в коде.

Вот как бы это выглядело:

```
if X > 100                  'If X is greater than 100
  !outa[0]                  'Toggle P0
elseif X > 50               'Else If X > 50
  !outa[2]                  'Toggle P2
elseif X == 90              'Else If X = 90 <-- ERROR, ABOVE COND.
  !outa[1]                  'Toggle P1      <-- SUPERSEDES THIS AND
                            '                   THIS CODE NEVER RUNS
```

Приведенный пример содержит ошибку, так как даже при X равном 90, условие `elseif X == 90` никогда не проверится, потому что предыдущее, `elseif X > 50`, проверилось ранее и, поскольку оно дало истину, выполнился его блок и больше никакие условия этого **IF** не проверяются. Если бы X был 50 или менее, последнее **ELSEIF** условие бы проверилось, но, конечно же, оно никогда бы не дало истину.

Использование IF совместно с ELSEIF и ELSE

Другая форма условной команды **IF** выполняет одно из многих действий если одно из многих условий верно, или альтернативное действие, если ни одно из предыдущих условий не дало истину. Записывается как **IF**, одно или более **ELSEIF**, и в конце — **ELSE:**

```
if X > 100                  'If X is greater than 100
  !outa[0]                  'Toggle P0
elseif X == 90              'Else If X = 90
  !outa[1]                  'Toggle P1
elseif X > 50               'Else If X > 50
  !outa[2]                  'Toggle P2
```

```
else                            'Otherwise,
  !outa[3]                      'Toggle P3
```

Этот пример похож на пример выше, за исключением того, что если ни одно из условий IF или ELSEIF не выполнится, переключится P3.

Условие ELSEIFNOT

Условие ELSEIFNOT ведет себя точно так же, как ELSEIF, за исключением того, что оно использует отрицательную логику; ELSEIFNOT выполняет свой блок *ElseIfNotStatement(s)* только если *Condition(s)* дает результат ЛОЖЬ (FALSE). Между единственным IF и опциональным ELSE может быть в любом порядке скомбинировано множество условий ELSEIFNOT.

IFNOT

Команда: Проверить условие(я) и выполнить блок кода, если условие верно (отрицательная логика).

((PUB ¦ PRI))
IFNOT ***Condition(s)***
→ ***IfNotStatement(s)***
〈 ELSEIF ***Condition(s)***
→ ***ElseIfStatement(s)*** 〉...
〈 ELSEIFNOT ***Condition(s)***
→ ***ElseIfNotStatement(s)*** 〉...
〈 ELSE
→ ***ElseStatement(s)*** 〉

- ***Condition(s)*** – одно или более условий – логических выражений для проверки.
- ***IfNotStatement(s)*** – блок из одной или более строк кода для выполнения, если условие IFNOT – ложь.
- ***ElseIfStatement(s)*** – опциональный блок из одной или более строк кода для выполнения, если все предыдущие условия *Condition(s)* неверны, а условие ELSEIF *Condition(s)* – истина.
- ***ElseIfNotStatement(s)*** – опциональный блок из одной или более строк кода для выполнения, когда все предыдущие условия *Condition(s)* неверны, и условие ELSEIFNOT's *Condition(s)* – ложь.
- ***ElseStatement(s)*** – опциональный блок из одной или более строк кода для выполнения, если все предыдущие условия *Condition(s*) неверны.

Описание

IFNOT – это одна из трех главных условных команд (IF, IFNOT, и CASE), которая производит условное выполнение блока кода. IFNOT – это обратная форма IF.

IFNOT проверяет условия и, если ложь, выполняет блок *IfNotStatement(s)*. Если *Condition(s)* – истина, по порядку проверяются условия дальнейших опциональных ELSEIF, и/или ELSEIFNOT, пока не найдется верное условие, после чего выполняется соответствующий блок *ElseIfStatement(s)* или *ElseIfNotStatement(s)*. Опциональный блок *ElseStatement(s)* выполняется, если не было найдено ни одного верного условия.

"Верное" условие – то, которое дает при своем решении результат TRUE для позитивной логики (IF или ELSEIF), либо FALSE — для отрицательной логики (ELSEIFNOT). Для информации об опциональных компонентах IFNOT, см. «IF» на стр. 257.

INA, INB

Регистр: Входные регистры для 32-битных портов Ports A и B.

((PUB ¦ PRI))
INA ⟨[*Pin(s)*]⟩

((PUB ¦ PRI))
INB ⟨[*Pin(s)*]⟩ (Reserved for future use)

Возвращает: Текущее состояние линий B/B для порта Port A или Port B.

- ***Pin(s)*** – опциональное выражение либо выражение-диапазон, которое задает линию(и) B/B для доступа к порту Port A (0-31) или Port B (32-63). Если дано как простое выражение, доступ осуществляется только к указанной линии B/B. Если введено как выражение-диапазон (два выражения в формате диапазона; x..y), доступ осуществляется к смежным линиям от начального до конечного номера.

Описание

INA и INB – это два из шести регистров (DIRA, DIRB, INA, INB, OUTA и OUTB), которые напрямую влияют на линии B/B. Регистр INA содержит текущие состояния для каждой из 32 линий B/B в порту Port A; биты от 0 до 31 соответствуют линиям от P0 до P31. Регистр INB содержит текущие состояния для каждой из 32 линий B/B в порту Port B; биты от 0 до 31 соответствуют линиям от P32 до P63.

ПРИМЕЧАНИЕ: INB зарезервирован для использования в будущем; ИМС Propeller P8X32A не содержит линий B/B порта Port B, поэтому ниже обсуждается только INA.

INA предназначен только для чтения и на самом деле реализован не как отдельный регистр, а как адрес, при обращении к которому как к источнику, инициируется прямое чтение линий порта Port A. В результате бит с низким уровнем (0) указывает, что соответствующая линия притянута к земле, а бит с высоким уровнем (1) указывает, что соответствующая линия притянута к VDD (3.3 Вольт). Поскольку ИМС Propeller произведена по технологии КМОП, линии B/B распознают любой уровень выше ½ VDD как высокий, поэтому «высокий уровень» означает, что на линии присутствует примерно 1.65 Вольт или больше.

Все линии подключены непосредственно к каждому процессору, поэтому *Hub* никакого влияния на доступ к линиям не оказывает. Каждый *Cog* содержит свой собственный регистр INA, который предоставляет ему возможность прочесть состояние

линий в любой момент времени. Читается реальное состояние линий, не зависимо от назначенного им направления – на ввод или на вывод.

Заметьте, что из-за природы реализации линий В/В в виде “монтажного ИЛИ”, электрическое соединение между процессорами отсутствует, хотя они все же могут осуществлять одновременный доступ к линиям В/В. На разработчика ложится задача убедиться, что ни одни два из процессоров не приводят к коллизиям на одной и той же линии В/В в процессе выполнения приложения. Поскольку все процессоры разделяют все линии В/В, *Cog* может использовать `INA` для чтения как линий используемых им самим, так и используемых одним или более другими процессорами.

Использование INA

При чтении, `INA` возвращает состояние всех линий В/В в данный момент времени. В следующем примере считается, что переменная `Temp` уже создана в другом месте.

```
Temp := INA             'Get state of P0 through P31
```

В этом примере читается состояние всех 32 линий В/В порта Port A в переменную `Temp`.

Используя опцию *Pin(s)*, *Cog* может читать по одной линии за один раз. Например:

```
Temp := INA[16]         'Get state of P16
```

Здесь состояние линии В/В 16 читается (0 или 1) и сохраняется в младшем бите переменной `Temp`; все остальные биты `Temp` очищаются в 0.

В языке *Spin* регистр `INA` поддерживает специальный формат выражений, называемый выражения-диапазон, которые позволяют читать одновременно одну группу линий В/В, не читая остальные, не входящие в заданный диапазон. Для одновременного чтения нескольких смежных линий, используйте выражение-диапазон в поле *Pin(s)*.

```
Temp := INA[18..15]     'Get states of P18:P15
```

Здесь младшие четыре бита `Temp` (3, 2, 1, и 0) установлены в состояния линий В/В соответственно 18, 17, 16, и 15, а все остальные биты `Temp` очищены в 0.

ВАЖНО: Порядок величин в выражении-диапазоне влияет на результат. Допустим, мы изменим порядок в выражении-диапазоне из предыдущего примера

```
Temp := INA[15..18]     'Get states of P15:P18
```

Биты `Temp` 3, 2, 1, и 0 установлены как линии В/В 15, 16, 17, и 18, соответственно.

Это очень мощное свойство выражений-диапазонов, но если ему уделить недостаточно внимания, оно может привести к неожиданным результатам.

LOCKCLR

Команда: Сбрасывает бит защиты в FALSE и возвращает его предыдущее значение.

((PUB ¦ PRI))
  LOCKCLR (*ID*)

Возвращает: Предыдущее значение бита защиты (TRUE или FALSE).

- ***ID*** – *ID*-номер (0 – 7) бита защиты, который нужно сбросить в FALSE.

Описание

LOCKCLR – это одна из четырех команд работы с битами защиты (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемых для управления ресурсами, определяемыми пользователем и считающимися взаимоисключающими. LOCKCLR очищает бит защиты с номером *ID* в значение FALSE и возвращает его предыдущее значение (TRUE или FALSE).

Для подробной информации об использовании битов защиты и команд LOCKxxx, см. «О битах защиты», стр. 268 и «Рекомендуемые правила для битов защиты», стр. 269.

В следующем примере будем предполагать, что *Cog* (либо текущий, либо другой) уже запросил бит защиты при помощи команды LOCKNEW и передал его *ID*-номер этому процессору, который сохранил его как SemID. Считаем также, что *Cog* имеет массив двойных слов с именем LocalData.

```
PUB ReadResource | Idx
  repeat until not lockset(SemID)    'wait until we lock the resource
  repeat Idx from 0 to 9             'read all 10 longs of resource
    LocalData[Idx] := long[Idx]
  lockclr(SemID)                     'unlock the resource

PUB WriteResource | Idx
  repeat until not lockset(SemID)    'wait until we lock the resource
  repeat Idx from 0 to 9             'write all 10 longs to resource
     long[Idx] := LocalData[Idx]
  lockclr(SemID)                     'unlock the resource
```

Оба из приведенных методов, ReadResource и WriteResource, придерживаются одних и тех же правил перед и после доступа к ресурсу. Сначала они ожидают неопределенное время в первом цикле repeat, пока он защитит ресурс, т.е. он успешно установит

соответствующий бит защиты. Если **LOCKSET** возвращает **TRUE**, условие "`until not lockset...`" даст **FALSE**, что означает, что какой-то другой *Cog* производит доступ к этому ресурсу, поэтому первый цикл `repeat` повторяется опять. Если же **LOCKSET** возвратит **FALSE**, условие "`until not lockset`" даст **TRUE**, что будет значить "ресурс защищен", и первый цикл `repeat` завершится. Второй цикл `repeat` в каждом из методов читает либо записывает ресурс с помощью `long[Idx]` и `LocalData[Idx]`. Последняя строка каждого из методов, `lockclr(SemID)`, очищает ассоциированный с ресурсом бит защиты в **FALSE**, логически открывая доступ или освобождая ресурс для использования другими.

Для дополнительной информации см. **LOCKNEW**, стр. 268; **LOCKRET**, стр. 271; и **LOCKSET**, стр. 272.

LOCKNEW

Команда: Запрашивает новый бит защиты и получает его *ID*-номер.

((PUB ¦ PRI))
  LOCKNEW

Возвращает: *ID*-номер (0-7) предоставленного бита защиты, либо -1, если ни один не был доступен.

Описание

LOCKNEW – это одна из четырех команд работы с битами защиты (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемая для управления ресурсами, определяемыми пользователем и считающимися взаимно-исключающими. LOCKNEW запрашивает уникальный бит защиты из *Hub*, и получает *ID*-номер этого бита защиты. Если на момент проверки не было доступных битов защиты, LOCKNEW возвращает -1.

О битах защиты

Биты защиты – это семафорный механизм, используемый для общения между двумя и более объектами. В ИМС Propeller, бит защиты – это просто один из восьми глобальных битов в защищенном регистре внутри Переключателя (*Hub*-а). *Hub* содержит реестр используемых битов защиты с их текущими состояниями. Процессоры могут запросить, установить, очистить и вернуть биты защиты по необходимости во время выполнения приложения для указания, доступен ли общий ресурс, такой как блок памяти, или – нет. Поскольку эти биты управляются только *Hub*-ом, только один *Cog* может повлиять на них в каждый момент времени, делая этот механизм контроля очень эффективным.

В приложениях, где два и более процессоров разделяют одну и ту же область памяти, инструмент, подобный битам защиты, может стать очень необходимым для исключения появления катастрофических ситуаций. В каждый момент времени, *Hub* предотвращает появление таких ситуаций на уровне элементарных данных (таких, как байт, слово или двойное слово), однако он не может исключить “логических” коллизий в блоках из многих элементов (таких, как блок из байтов, слов, двойных слов либо любой их комбинации). Например, если два или более процессоров разделяют один и тот же байт основной памяти, каждый из них имеет гарантированно монопольный доступ к этому байту по природе построения Переключателя. Однако если эти два процессора разделяют блок из множества байт в основной памяти, *Hub* не сможет предохранить один процессор от перезаписи некоторых из тех байт в то время, как

другой процессор читает их; все взаимодействия процессоров с этими байтами могут чередоваться во времени. В этом случае разработчик должен организовать каждый процесс (в каждом процессоре, использующем эту память) так, чтобы они использовали этот блок памяти совместно, но неразрушающим путем. Биты защиты служат флагами, которые информируют каждый *Cog*, безопасна ли в данный момент работа с ресурсом, или нет.

Использование LOCKNEW

Определяемые пользователем взаимоисключающие ресурсы должны быть изначально проинициализированы процессором, после этого этот же процессор должен при помощи **LOCKNEW** проверить наличие уникального бита защиты, с помощью которого он будет управлять этим ресурсом и затем передавать *ID*-номер этого бита любому другому процессору, который его затребует. Например:

```
VAR
  byte SemID

PUB SetupSharedResource
  <code to set up user-defined, shared resource here>
  if (SemID := locknew) == -1
    <error, no locks available>
 else
   <share SemID's value with other Cogs>
```

В этом примере вызывается **LOCKNEW** и результат сохраняется в переменной SemID. Если этот результат равен -1, возникает ошибка. Если же SemID не равен -1, то найден доступный бит защиты и этот SemID должен быть передан другим процессорам вместе с адресом ресурса, для которого используется SemID. Метод, используемый для связи SemID и адреса ресурса зависит от приложения, но обычно они оба передаются как параметры методу *Spin*, который запущен в процессоре, или как параметр **PAR**, если в процессоре выполняется ассемблерная подпрограмма. См. **COGNEW**, стр. 221.

Рекомендуемые правила для битов защиты

Далее приведены правила, принятые при использовании битов защиты.

- Объекты, требующие защиту для работы с определяемым пользователем взаимоисключающим ресурсом должны запросить наличие бита защиты при помощи **LOCKNEW**, затем сохранить возвращенный *ID*-номер (будем здесь называть его SemID). Только один *Cog* может получить этот бит. Cog,

получивший этот бит, должен передать SemID всем *Cog*-ам, которые будут использовать этот ресурс.

- Любой *Cog*, которому необходимо получить доступ к ресурсу, должен вначале успешно "установить" бит защиты SemID. Бит успешно "установлен", когда команда LOCKSET(SemID) возвращает **FALSE**, это значит, что этот бит защиты до этого не был установлен. Если **LOCKSET** вернула **TRUE**, то должно быть, другой *Cog* в это время имеет доступ к этому ресурсу; Вы должны подождать и попытаться еще раз выполнить успешную "установку" позднее.
- Процессор, достигнувший успешной "установки" может управлять ресурсом по своему усмотрению. По окончанию, он должен очистить бит защиты командой LOCKCLR(SemID), после чего другой *Cog* может получить доступ к ресурсу. В отлаженной системе результат **LOCKCLR** может игнорироваться, поскольку этот *Cog* – единственный, имеющий право очищать этот бит защиты.
- Если ресурс более не нужен, либо становится не взаимоисключающим, ассоциированный с ним бит защиты должен быть возвращен в очередь свободных битов командой LOCKRET(SemID). Обычно это делается тем же процессором, что и первоначально запрашивал наличие бита.

Приложения должны писаться таким образом, чтобы команды **LOCKSET** или **LOCKCLR** не выполнялись, пока биты защиты не будут выделены.

Отметьте, что определяемые пользователем ресурсы на самом деле не защищаются ни *Hub*-ом, ни выделением битов защиты. Свойство битов защиты – это только предоставить объектам средства для совместной защиты этих ресурсов. Сами объекты должны решать, как использовать правила для битов защиты, и какие ресурсы будут ими управляться. Вдобавок, *Hub* напрямую не сопоставляет конкретный бит защиты процессору, который вызвал **LOCKNEW**, скорее, он просто отмечает бит как "занятый" процессором; любой другой *Cog* может "вернуть" этот бит в очередь свободных битов защиты. Любой процессор также может получить доступ к любому биту защиты посредством команд **LOCKCLR** и **LOCKSET**, даже если эти биты никогда не были выделены. Подобные действия выполнять не рекомендуется из-за беспорядка, который возникнет в приложении, даже при работе с другими, хорошо отлаженными объектами.

Для дополнительной информации см. **LOCKRET**, стр. 271; **LOCKCLR**, стр. 266; и **LOCKSET**, стр. 272.

LOCKRET

Команда: Освобождает бит защиты и отправляет его назад, в очередь свободных битов, делая его доступным для будущих запросов LOCKNEW.

((PUB ¦ PRI))
  LOCKRET (*ID*)

- ***ID*** – *ID*-номер (0 – 7) бита защиты, возвращаемого в очередь свободных.

Описание

LOCKRET – это одна из четырех команд работы с битами защиты (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемая для управления ресурсами, определяемыми пользователем как взаимно-исключающие. LOCKRET возвращает бит защиты по *ID*, назад в набор свободных битов защиты в *Hub*-е, так что он может быть использован вновь другими процессорами. Например:

```
LOCKRET (2)
```

В этом примере бит защиты 2 возвращается назад в *Hub*. Это не предотвращает процессоры от дальнейшего доступа к биту 2, это просто позволит *Hub*-у отдать его новому *Cog*-у, который вызовет LOCKNEW. Приложения должны писаться таким образом, чтобы команды LOCKSET или LOCKCLR не выполнялись, пока биты защиты не будут выделены.

Для информации о типичном использовании битов защиты и команд LOCKxxx, см. «О битах защиты», стр. 268, и «Рекомендуемые правила для битов защиты», стр. 269.

Отметьте, что определяемые пользователем ресурсы на самом деле не защищаются ни *Hub*-ом, ни выделением битов защиты. Свойство битов защиты – только предоставить объектам средства для совместной защиты этих ресурсов. Сами объекты должны решать, как использовать правила для битов защиты, и какие ресурсы будут ими управляться. В добавок, *Hub* напрямую не сопоставляет конкретный бит защиты процессору, который вызвал LOCKNEW, скорее, он просто отмечает бит как “занятый” процессором; любой другой *Cog* может “вернуть” этот бит в очередь свободных битов защиты. Любой процессор также может получить доступ к любому биту защиты посредством команд LOCKCLR и LOCKSET, даже если эти биты никогда не были выделены. Такие действия обычно не рекомендуется выполнять из-за беспорядка, который может это внести в приложение даже при работе с другими, хорошо отлаженными объектами.

Для детальной информации см. LOCKNEW, стр. 268; LOCKCLR, стр. 266; и LOCKSET, стр. 272.

LOCKSET

Команда: Установить бит защиты в TRUE и возвратить его предыдущее состояние.

((PUB ¦ PRI))
  LOCKSET (*ID*)

Возвращает: Предыдущее состояние бита защиты (TRUE или FALSE).

- ***ID*** – *ID*-номер (0 – 7) бита защиты, устанавливаемого в TRUE.

Описание

LOCKSET – это одна из четырех команд работы с битами защиты (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемая для управления ресурсами, определяемыми пользователем и считающимися взаимно-исключающими. LOCKSET устанавливает бит с номером *ID* в TRUE и возвращает его предыдущее состояние (TRUE или FALSE).

Для информации о типичном использовании битов защиты и команд LOCKxxx см. «О битах защиты», стр. 268, и «Рекомендуемые правила для битов защиты», стр. 269.

В следующем примере будем предполагать, что *Cog* (либо текущий, либо другой) уже запросил бит защиты при помощи команды LOCKNEW и передал его *ID*-номер этому процессору, который сохранил его как SemID. Считаем также, что *Cog* имеет массив двойных слов с именем LocalData.

```
PUB ReadResource | Idx
  repeat until not lockset(SemID) 'wait until we lock the resource
  repeat Idx from 0 to 9          'read all 10 longs of resource
    LocalData[Idx] := long[Idx]
  lockclr(SemID)                  'unlock the resource

PUB WriteResource | Idx
  repeat until not lockset(SemID) 'wait until we lock the resource
  repeat Idx from 0 to 9          'write all 10 longs to resource
    long[Idx] := LocalData[Idx]
  lockclr(SemID)                  'unlock the resource
```

Оба из приведенных методов, ReadResource и WriteResource, придерживаются одних и тех же правил перед и после доступа к ресурсу. Сначала они ожидают неопределенное время в первом цикле repeat, пока он защитит ресурс, т.е. он успешно установит

соответствующий бит защиты. Если **LOCKSET** возвращает **TRUE**, условие "`until not lockset...`" даст **FALSE**, что означает, что какой-то другой *Cog* производит доступ к этому ресурсу, поэтому первый цикл `repeat` повторяется опять. Если же **LOCKSET** возвратит **FALSE**, условие "`until not lockset`" даст **TRUE**, что будет значить "ресурс защищен", и первый цикл `repeat` завершится. Второй цикл `repeat` в каждом из методов читает либо записывает ресурс с помощью `long[Idx]` и `LocalData[Idx]`. Последняя строка каждого из методов, `lockclr(SemID)`, очищает ассоциированный с ресурсом бит защиты в **FALSE**, логически открывая доступ или освобождая ресурс для использования другими.

Для дополнительной информации См. **LOCKNEW**, стр. 268; **LOCKRET**, стр. 271; и **LOCKCLR**, стр. 266.

LONG

Объявление: Объявляет переменную размером *long* (двойное слово), либо данные размером *long* и выровненные по границе *long*, либо читает/записывает *long* основной памяти.

```
VAR
  LONG Symbol {[Count]}
DAT
  {Symbol} LONG Data {[Count]}
((PUB | PRI))
  LONG [BaseAddress] {[Offset]}
```

- ***Symbol*** –имя переменной (синтаксис 1) или блока данных (синтаксис 2).
- ***Count*** – опциональное выражение, указывающее количество элементов размером *long* для идентификатора *Symbol* (синтаксис 1), либо количество *long*-элементов данных ***Data*** (синтаксис 2) для сохранения в виде таблицы.
- ***Data*** – константное выражение либо список константных выражений, разделенных запятыми.
- ***BaseAddress*** – выражение, задающее адрес в основной памяти для чтения или записи. Если *Offset* опущен, *BaseAddress* – это реальный адрес данных. Если *Offset* задан, реальный адрес данных равен *BaseAddress* + *Offset*.
- ***Offset*** – опциональное выражение, указывающее смещение до данных относительно адреса *BaseAddress*. Измеряется ***Offset*** в *long*-ах.

Описание

LONG - это одно из трех объявлений (BYTE, WORD, и LONG), которые объявляют либо оперируют с памятью. Объявление LONG может использоваться для:

1) объявления идентификатора переменной размером *long* (двойное слово, 32-бита) либо массива из таких идентификаторов в блоке VAR, или
2) объявления *long*-выровненных и/или *long*-размерных данных в блоке DAT, или
3) чтения или записи *long* в основной памяти по базовому адресу со смещением.

Диапазон значений long

Ячейка памяти размером в двойное слово (32 бита) может содержать значение, равное одной из 2^{32} возможных комбинаций битов (то есть одной из 4294967296 комбинаций). Поскольку язык *Spin* выполняет все математические операции, используя 32-битную

арифметику со знаком, то значения *long* имеют диапазон от -2147483648 до +2147483647. Однако число, которое содержится в *long*, в известной степени зависит от представления компьютера и пользователя о нем. В *Propeller*-ассемблере величина *long* может рассматриваться как знаковое, так и беззнаковое значение.

Объявление переменной типа Long (Синтаксис 1)

В блоках **VAR**, для объявления глобальных переменных, которые либо имеют размер *long*, либо являются массивом из *long*-ов, используется синтаксис 1 объявления **LONG**. Например:

```
VAR
  long Temp                   'Temp is a long (2 words, 4 bytes)
  long List[25]               'List is a long array
```

В приведенном примере объявляются две переменные, с идентификаторами `Temp` и `List`. Переменная `Temp` – это просто одиночная переменная размером *long*. Строка под объявлением переменной `Temp` использует опциональное поле *Count* для создания массива из 25 *long*-переменных с названием `List`. Как `Temp` , так и `List` доступны из любого **PUB** или **PRI** метода в рамках того же объекта, в котором расположен блок **VAR** с их объявлением; они являются глобальными по отношению к объекту. Например.

```
PUB SomeMethod
  Temp := 25_000_000          'Set Temp to 25,000,000
  List[0] := 500_000          'Set first element of List to 500,000
  List[1] := 9_000            'Set second element of List to 9,000
  List[24] := 60              'Set last element of List to 60
```

Для более детальной информации об использовании **LONG** в этом синтаксисе, обратитесь к **VAR**-секции «Объявление переменных (Синтаксис 1)» на стр. 364, и помните, что в поле *Size* в этом описании нужно использовать **LONG**.

Объявление данных Long (Синтаксис 2)

В блоках **DAT**, для объявления данных, выровненных по размеру *long* и/или размером *long*, которые компилируются в главной памяти как константы, используется синтаксис 2 объявления **LONG**. В этих блоках данным может быть сопоставлен опциональный идентификатор, который затем используется для ссылок на эти данные (см. **DAT**, стр. 243). Например:

```
DAT
  MyData long 640_000, $BB50              'Long-aligned/sized data
  MyList byte long $FF995544, long 1_000  'Byte-aligned/long-sized
```

В этом примере объявлено два идентификатора – MyData и MyList. Идентификатор MyData указывает на начало *long*-выровненных и *long*-размерных данных в основной памяти. Значения MyData в основной памяти – это соответственно 640000 и $0000BB50. Идентификатор MyList использует особый синтаксис объявления LONG в блоке DAT и создает набор *byte*-выровненных, но *long*-размерных данных в основной памяти. Значения MyList в основной памяти – это соответственно $FF995544 и 1000. При побайтном доступе, MyList содержит $44, $55, $99, $FF и 232, 3, 0, 0 поскольку данные хранятся в формате little-endian.

Отметьте: MyList можно объявить как *word*-выровненные, *long*-размерные данные, если заменить "byte" на "word".

Эти данные компилируются в объект и в конечное приложение как часть выполнимого кода и доступны для чтения/записи при использовании синтаксиса 3 объявления LONG (см.ниже). Для более детальной информации об использовании LONG в этом синтаксисе, обратитесь к DAT-секции «Объявление Данных (Синтаксис 1)», на стр. 244, и помните, что в поле *Size* в том описании нужно использовать LONG.

Элементы данных могут повторяться, если использовать опциональное поле *Count*. Например:

```
DAT
  MyData long 640_000, $BB50[3]
```

В приведенном выше примере объявляется *long*-выровненная таблица *long*-размерных данных с именем MyData, состоящая из следующих четырех значений: 640000, $BB50, $BB50, $BB50. Значение $BB50 встречается три раза, что обусловлено наличием в объявлении сразу после него поля '[3]'.

Чтение/Запись Long-величин основной памяти (Синтаксис 3)

Для чтения или записи *long*-величин основной памяти, в блоках PUB и PRI используется синтаксис 3 объявления LONG. При этом запись выражений, обращающихся к памяти, выполняется в следующем формате: long[*BaseAddress*][*Offset*]. Например:

```
PUB MemTest | Temp
  Temp := LONG[@MyData][1]                'Read long value
  LONG[@MyList][0] := Temp + $01234567    'Write long value
```

```
DAT
  MyData  long 640_000, $BB50              'Long-sized/aligned data
  MyList  byte long $FF995544, long 1_000  'Byte-sized/aligned long
data
```

Данные, объявляемые в нижней части этого примера в блоке **DAT**, располагаются в памяти согласно Рис. 4-2. Первый элемент `MyData` располагается по адресу $18. Последний – по адресу $1C, сопровождаемый первым элементом `MyList` по адресу $20. Отметьте, что начальный адрес ($18) – произволен и может изменяться компилятором при изменении кода или при включении объекта в другое приложение.

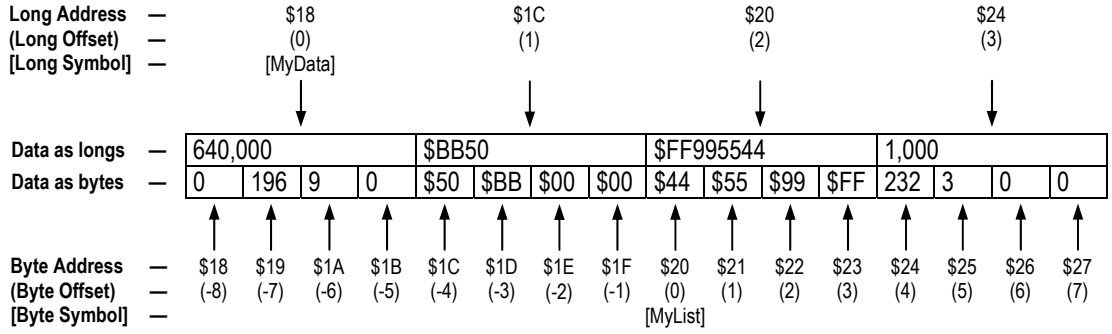


Рис. 4-2: Структура и адрессация данных размером Long в основной памяти

В верхней части кода, в первой исполнимой строке метода `MemTest`, `Temp := LONG[@MyData][1]`, производится чтение *long*-величины из основной памяти. В результате локальная переменная `Temp` устанавливается в $BB50, то есть в значение, прочитанное по адресу $1C. Адрес $1C был определен по адресу `MyData` ($18) плюс 1 двойное слово смещения (4 байта). Этот процесс показан на упрощенной диаграмме.

LONG[@MyData][1] → LONG[$18][1] → LONG[$18 + (1*4)] → LONG[$1C]

В следующей строке, `LONG[@MyList][0] := Temp + $01234567`, производится запись значения *long* в основную память. В результате, *long*-величина в памяти по адресу $20 устанавливается в $012400BC. Адрес $20 был вычислен по адресу `MyList` ($20) плюс смещение 0 двойных слов (0 байтов).

LONG[@MyList][0] → LONG[$20][0] → LONG[$20 + (0*4)] → LONG[$20]

Величина $012400BC была получена из текущего значения Temp плюс $012400BC.

Адресация основной памяти

Как отображено на Рис.4-2, основная память представляет собой набор последовательно расположенных байтов (см. строку “data as bytes”), которые при правильном подходе могут быть прочитаны как *long*-значения (4-байтные величины). На самом деле, в приведенном выше примере показано, что адреса также вычисляются в позициях байтов. Такой подход действителен для всех команд, использующих адресацию.

Адресация основной памяти всегда байтовая, независимо от размера переменной, к которой осуществляется доступ, — будь она однобайтовая, одинарное либо двойное слово. Это упрощает понимание взаимного расположения байтов, слов и двойных слов, но в то же время вносит некоторые сложности при рассмотрении нескольких элементов одинакового размера, таких как двойные слова.

С этой точки зрения, директива LONG имеет очень удобное свойство обеспечивать адрессацию в масштабе *long*. При использовании поля *BaseAddress* совместно с опциональным полем *Offset*, действия производятся в базисе, соответствующем размеру данных.

Представьте себе доступ к двойным словам памяти с известной начальной точки (*BaseAddress*). Естественно думать, что следующий *long* или *long*-и будут находиться от этой точки на определенном смещении (*Offset*). Хотя эти *long*-и на самом деле находятся за этой точкой по смещению на определенное количество байт, все же будет проще понять, если говорить о смещении на некоторое количество *long*-ов (то есть: 4^{й} *long*, вместо: *long*, который начинается с 12го байта). Идентификатор LONG манипулирует данными именно таким образом, умножая величину *Offset* (предоставляемую в *long*-ах), на 4 (количество байт в одном *long*), и прибавляя результат к *BaseAddress* для определения адреса необходимой области памяти для чтения. Он также очищает два младших бита *BaseAddress* для обеспечения выравнивания адреса по размеру двойного слова.

Таким образом, при чтении значений из массива MyData, команда long[@MyData][0] читает первую *long*-величину, а long[@MyData][1] – вторую.

Если бы поле *Offset* не использовалось, приведенные выше операторы выглядели бы соответственно как long[@MyData] и long[@MyData+4]. Результат бы оказался таким же, но запись бы не воспринималась так легко, как ранее.

Для дополнительной информации о размещении данных в памяти, см. «Объявление данных (Синтаксис 1)» блока DAT на стр. 243.

Альтернативный доступ к памяти

Существует еще один метод доступа к данным, альтернативный использованному в предыдущем примере; в нем можно обращаться к данным напрямую по имени. К примеру, в следующем выражении производится чтение первых двух *long*-ов массива MyData:

```
Temp := MyData[0]
Temp := MyData[1]
```

Так почему бы постоянно не использовать прямое обращение по имени? Рассмотрим следующий случай:

```
Temp := MyList[0]
Temp := MyList[1]
```

Обращаясь к рассмотренному ранее Рис. 4-2, можно ожидать, что эти два оператора прочтут первый и второй *long*-и массива MyList: соответственно $FF995544 и 1000. Однако, напротив, они прочтут лишь первый и второй байты MyList, то есть соответственно $44 и $55.

Что произошло? В отличие от MyData, идентификатор MyList был задан как байт-размерные и байт-выровненные данные. Хотя данные и состоят из величин *long*, так как каждый элемент предварен директивой **LONG,** но поскольку размерность была задана байтовая, все прямые обращения будут возвращать отдельные байты.

Однако в этом случае, для получения значений *long* все же можно использовать указатель **LONG**, поскольку данные, следуя за MyData, получились выровненными по границе двойных слов.

```
Temp := long[@MyList][0]
Temp := long[@MyList][1]
```

Здесь из MyList прочтется первый *long*, $FF995544, а затем – второй, 1000. Это свойство очень удобно в случае, когда к одним и тем же данным в одно время необходимо обращаться как к байтам, а в другое – как к *long*-ам.

Другие особенности доступа

Оба приведенных метода доступа к памяти, — как директива **LONG,** так и прямое обращение по имени, — могут использоваться для доступа к любой области основной памяти, независимо от того, как она относится к конкретным структурам данных. Несколько примеров:

```
Temp := long[@MyList][-1]  'Read last long of MyData (before MyList)
Temp := long[@MyData][2]   'Read first long of MyList (after MyData)
```

```
Temp := MyList[-8]          'Read first byte of MyData
Temp := MyData[-2]          'Read long that is two longs before MyData
```

В этих примерах производится чтение данных за рамками логических границ (начальной и конечной точек) структуры объявленных данных. Это может оказаться и полезным трюком, однако чаще такое происходит по ошибке; будьте внимательны при адресации памяти, особенно когда вы выполняете операции записи.

LONGFILL

Команда: Заполняет двойные слова в основной памяти заданной величиной .

((PUB ¦ PRI))
LONGFILL(*StartAddress*, *Value*, *Count*)

- ***StartAddress*** – выражение, указывающее на адрес первого двойного слова (*long*) памяти для заполнения значением *Value*.
- ***Value*** – выражение, указывающее значение, которым необходимо заполнять двойные слова памяти.
- ***Count*** – выражение, указывающее количество двойных слов для заполнения, начиная с адреса *StartAddress*.

Описание

LONGFILL – это одна из трех команд (BYTEFILL, WORDFILL, и LONGFILL), используемых для заполнения блоков основной памяти заданным значением. LONGFILL заполняет *Count* *long*-ов основной памяти значением *Value*, начиная с адреса *StartAddress*.

Использование LONGFILL

LONGFILL – это мощное средство для очистки больших блоков *long*-размерной памяти. Например:

```
VAR
  long Buff[100]

PUB Main
  longfill(@Buff, 0, 100)           'Clear Buff to 0
```

Первая строка метода Main очищает весь массив Buff из 100 двойных слов (400 байт) в ноли. Для таких задач LONGFILL работает быстрее соответствующих циклов REPEAT.

LONGMOVE

Команда: Копирует двойные слова (*long*-и) из одной области памяти в другую.

((PUB ¦ PRI))
LONGMOVE (*DestAddress*, *SrcAddress*, *Count*)

- ***DestAddress*** – выражение, задающее адрес области в основной памяти, куда будет скопирован первый *long* из источника.
- ***SrcAddress*** – выражение, задающее адрес области в основной памяти, где находится первый копируемый *long*.
- ***Count*** – выражение, отображающее количество *long*-ов в области источника для копирования в область приемника

Описание

LONGMOVE - это одна из трех команд (BYTEMOVE, WORDMOVE, и LONGMOVE), используемых для копирования блоков данных основной памяти из одной области в другую. LONGMOVE копирует *Count* двойных слов из области основной памяти, начинающейся с адреса *SrcAddress*, в область основной памяти, начинающуюся с *DestAddress*.

Использование LONGMOVE

LONGMOVE – это мощный способ, используемый для копирования больших блоков *long*-размерной памяти. Например:

```
VAR
  long Buff1[100]
  long Buff2[100]

PUB Main
  longmove(@Buff2, @Buff1, 100)        'Copy Buff1 to Buff2
```

Первая строка метода Main копирует весь массив Buff1 из 100 *long*-ов (400 байт) в массив Buff2. Для таких задач LONGMOVE работает быстрее соответствующих циклов REPEAT.

LOOKDOWN, LOOKDOWNZ

Команда: Получить индекс значения в списке.

((PUB ¦ PRI))
LOOKDOWN (*Value* : *ExpressionList*)

((PUB ¦ PRI))
LOOKDOWNZ (*Value* : *ExpressionList*)

Возвращает: Индекс с базой 1 (LOOKDOWN) либо индекс с базой 0 (LOOKDOWNZ) значения *Value* в списке *ExpressionList*, либо 0, если значение *Value* не найдено.

- ***Value*** – выражение, указывающее значение, которое требуется найти в списке *ExpressionList*.
- ***ExpressionList*** – разделенный запятыми список выражений. Допускается вводить строки символов, заключенные в кавычки; они рассматриваются как разделенный запятыми список символов.

Описание

LOOKDOWN и LOOKDOWNZ - это команды, находящие индексы величин в списке величин. LOOKDOWN возвращает значение индекса с базой 1 (1..N) величины *Value* в списке *ExpressionList*. LOOKDOWNZ работает так же, как LOOKDOWN, за исключением того, что возвращает индекс с базой 0 (0..N−1). Для обеих команд возвращается 0, если значение *Value* не найдено в списке *ExpressionLis,*.

Использование LOOKDOWN или LOOKDOWNZ

LOOKDOWN и LOOKDOWNZ полезны для отображения набора неупорядоченных чисел (25, -103, 18, и т.д.) на упорядоченный набор чисел (1, 2, 3, и т.д. –или– 0, 1, 2, и т.д.), где невозможно определить алгебраическую зависимость для краткого описания набора. В следующем примере считается, что метод Print создан где-то в другом месте объекта.

```
PUB ShowList | Index
  Print(GetIndex(25))
  Print(GetIndex(300))
  Print(GetIndex(2510))
  Print(GetIndex(163))
  Print(GetIndex(17))
  Print(GetIndex(8000))
  Print(GetIndex(3))
```

```
PUB GetIndex(Value): Index
  Index := lookdown(Value: 25, 300, 2_510, 163, 17, 8_000, 3)
```

Метод GetIndex в этом примере использует **LOOKDOWN** , чтобы найти значение Value, и возвращает его индекс в списке *ExpressionList*, либо 0, если значение не найдено. Метод ShowList периодически вызывает GetIndex с различными величинами и печатает найденные индексы на дисплее. Предполагая, что Print – это метод, который выводит на экран заданную ему величину, в этом примере на дисплее будет напечатано 1, 2, 3, 4, 5, 6 и 7.

Если бы вместо **LOOKDOWN** использовалась команда **LOOKDOWNZ**, в этом примере на дисплее было бы напечатано соответственно 0, 1, 2, 3, 4, 5, и 6.

Если величина *Value* не найдена, **LOOKDOWN**, или **LOOKDOWNZ**, возвращают 0. Поэтому, если бы одна из строк метода ShowList была Print(GetIndex(50)), на дисплее в момент ее выполнения отобразился бы 0.

При использовании **LOOKDOWNZ**, помните, что она может вернуть 0 либо если величина *Value* не найдена, либо если она находится по индексу 0. Убедитесь, что эта особенность не вызовет ошибок в Вашем коде, или используйте вместо нее **LOOKDOWN**.

LOOKUP, LOOKUPZ

Команда: Получить значение из списка по заданному индексу.

((PUB ¦ PRI))
  LOOKUP (***Index*** : ***ExpressionList***)

((PUB ¦ PRI))
  LOOKUPZ (***Index*** : ***ExpressionList***)

Возвращает: Значение по индексу *Index* из списка *ExpressionList* с базой 1 (LOOKUP) либо с базой 0 (LOOKUPZ), либо 0 – если вне диапазона.

- ***Index*** – выражение, указывающее положение желаемого значения в списке *ExpressionList*. Для LOOKUP, база у *Index* равна 1 (1..N). Для LOOKUPZ, база у *Index* равна 0 (0..N-1).
- ***ExpressionList*** – разделенный запятыми список выражений. Допускается вводить строки символов, заключенные в кавычки; они рассматриваются как разделенный запятыми список символов.

Описание

LOOKUP и LOOKUPZ - это команды, которые позволяют найти нужную запись в списке значений. LOOKUP возвращает из списка *ExpressionList* величину, которая находится в позиции, представленной в индексе *Index* с базой 1 (1..N). LOOKUPZ такая же, как и LOOKUP, за исключением того, что она использует базу, равную 0 (0..N-1). Для обеих команд возвращается 0, если индекс *Index* находится вне диапазона.

Использование LOOKUP или LOOKUPZ

LOOKUP и LOOKUPZ полезны для отображения набора упорядоченных чисел (1, 2, 3, и т.д. –или– 0, 1, 2, и т.д.), на набор неупорядоченных чисел (25, -103, 18, и т.д.), где невозможно определить алгебраическую зависимость для краткого описания набора. В следующем примере считается, что метод Print создан где-то в другом месте объекта.

```
PUB ShowList | Index, Temp
  repeat Index from 1 to 7
    Temp := lookup(Index: 25, 300, 2_510, 163, 17, 8_000, 3)
    Print(Temp)
```

Здесь просматриваются и отображаются все значения из списка *ExpressionList*. Цикл REPEAT организован по Index от 1 до 7. Каждый проход цикла LOOKUP использует Index

для получения значения из своего списка. При Index равном 1, возвращается значение 25. При Index равном 2, возвращается 300. Если полагать, что Print – это метод, отображающий на дисплее значение переменной Temp, то в этом примере на дисплее будет отображено 25, 300, 2510, 163, 17, 8000 и 3.

При использовании **LOOKUPZ**, список будет с нулевой базой (0..N-1), а не с единичной; при этом Index равный 0 даст значение 25, а Index равный 1 даст 300, и т.д.

Если индекс *Index* находится вне диапазона индексов списка, возвращается 0. Так, для команды **LOOKUP**, если бы **REPEAT** был организован от 0 до 8, а не от 1 до 7, в этом примере на дисплее отобразилось бы 0, 25, 300, 2510, 163, 17, 8000, 3 и 0.

NEXT

Команда: Пропустить оставшиеся выражения цикла REPEAT и начать следующий проход цикла.

((PUB ¦ PRI))
  NEXT

Описание

NEXT – это одна из двух команд (NEXT и QUIT), которые влияют на циклы REPEAT. NEXT приводит к пропусканию всех оставшихся выражений в цикле REPEAT и к дальнейшему старту нового прохода этого цикла.

Использование NEXT

NEXT обычно используется как исключительная ситуация, в условном выражении в циклах REPEAT для мгновенного перехода к началу следующего прохода цикла. Например, допустим, что X – это переменная, созданная ранее, а Print() – это метод, созданный в другом месте, который печатает значение на дисплее:

```
repeat X from 0 to 9 'Repeat 10 times
  if X == 4
    next 'Skip if X = 4
  byte[$7000][X] := 0 'Clear RAM locations
  Print(X) 'Print X on screen
```

Приведенный пример циклично очищает ячейки ОЗУ и печатает значение X на дисплее, но с одним исключением. Если X равен 4, тело IF выполнит команду NEXT, которая приведет к пропусканию оставшихся команд в теле цикла и переходу на начало следующего прохода цикла. Это приведет к тому, что очистятся ячейки ОЗУ с $7000 по $7003 и ячейки с $7005 по $7009, на дисплей будет выведено 0, 1, 2, 3, 5, 6, 7, 8, 9.

Команда NEXT может быть использована только внутри цикла REPEAT, иначе возникнет ошибка.

OBJ

Объявление: Объявляет объектный блок.

```
OBJ
  Symbol {[Count]}: "ObjectName" { ↳Symbol {[Count]}: "ObjectName"}...
```

- ***Symbol*** – желаемое имя идентификатора объекта.
- ***Count*** – опциональное выражение, заключенное в квадратные скобки, что указывает, что это – массив объектов, с количеством элементов *Count*. В дальнейшем при ссылке на эти элементы, первый имеет индекс 0, а последний - *Count*-1.
- ***ObjectName*** – имя файла желаемого объекта, без расширения. Во время компиляции объект с этим именем ищется среди редактируемых вкладок, в рабочей и библиотечной директории. Имя объекта может содержать любые разрешенные символы; запрещенными являются \, /, :, *, ?, ", <, >, и |.

Описание

Объектный блок – это секция исходного кода, которая объявляет, какие объекты будут использоваться и какие идентификаторы будут их представлять. Это одно из шести объявлений (**CON**, **VAR**, **OBJ**, **PUB**, **PRI**, **DAT**), обеспечивающих четкую структуру языка *Spin*.

Объявление объектного блока начинается с **OBJ,** сопровождаемой одним или более объявлением. **OBJ** должна начинаться с самой левой колонки, и мы рекомендуем, чтобы остальные строки этого блока были введены с отступом. Например:

```
OBJ
  Num  : "Numbers"
  Term : "TV_Terminal"
```

В этом примере определяется идентификатор Num как объект типа "Numbers", и идентификатор Term как объект типа "TV_Terminal". Методы **PUB** и **PRI** могут обращаться к этим объектам используя объявленные идентификаторы, как показано в следующем примере.

```
PUB Print | S
  S := Num.ToStr(LongVal, Num#DEC)
  Term.Str(@S)
```

Здесь *Public*-метод Print, вызывает метод ToStr объекта Numbers, а так же метод Str объекта TV_Terminal. Делает это он посредством объектных идентификаторов Num и Term, сопровождаемых символом ссылки Объект-Метод (точка '.'), и в конце — именем вызываемого метода. Например, Num.ToStr вызывает *Public*-метод ToStr объекта Numbers, а Term.Str вызывает *Public*-метод Str объекта TV_Terminal. В этом случае Num.ToStr имеет два параметра, в скобках, а Term.Str – один параметр.

Также отметьте, что второй параметр вызова Num.ToStr – это Num#DEC. Символ '#' — это идентификатор ссылки Объект-Константа; он дает доступ к константам объекта. В этом случае, Num#DEC ссылается на **DEC** (десятичную) константу объекта Numbers.

Для более детальной информации см. «Ссылка Объект-Метод '.'» и «Ссылка Объект-Константа '#'» в Табл. 4-16, на стр. 361.

Используя синтаксис **OBJ** с массивом, можно объявить несколько экземпляров объекта с одинаковыми именами; доступ к ним производится как к массивам. Например:

```
OBJ
  PWM[2] : ''PWM''
PUB GenPWM
  PWM[0].Start
  PWM[1].Start
```

Здесь объявляется PWM как массив из двух объектов (двух экземпляров одного объекта). Так совпало, что сам объект тоже называется "PWM". Public-метод GenPWM вызывает метод Start каждого экземпляра, используя индексы 0 и 1 с идентификатором массива объектов PWM.

Оба экземпляра объекта PWM компилируются в приложении таким образом, что существует лишь одна копия их программного кода (**PUB**, **PRI**, и **DAT**), но две копии их переменных (**VAR**). Так происходит потому, что код для каждого экземпляра одинаков, но, чтобы выполняться независимо друг от друга, каждый требует своей области переменных.

Важно отметить, что для нескольких экземпляров объекта существует лишь один блок **DAT**, поскольку он может содержать код ассемблера. Блоки **DAT** также могут содержать инициализированные данные и дополнительные области для работы, имеющие свои имена. Поскольку существует лишь одна их копия, эта область является общей для всех экземпляров объекта. Таким образом существует удобный механизм для создания общей памяти между несколькими экземплярами одного объекта.

Область видимости объектных идентификаторов

Идентификаторы, созданные в Объектных Блоках – глобальные для объекта, в котором они объявлены, но не доступны вне этого объекта. Это значит, что эти идентификаторы могут быть доступны напрямую из любого места внутри объекта, и их имена не будут конфликтовать с такими же у их родительских или дочерних объектов.

Операторы Spin

ИМС Propeller имеет мощный набор математических и логических операторов. Подмножество этих операторов поддерживается также ассемблером Propeller; однако, поскольку все формы поддерживаемых в ИМС Propeller операторов используются в языке *Spin*, каждый оператор будет детально описан в этой секции. Для перечня операторов, доступных в языке ассемблер см. секцию «Операторы Spin» на стр. 484.

Разрядная сетка

ИМС Propeller – это 32-разрядный контроллер, и, если это не оговорено отдельно, выражения всегда вычисляются с использованием математики для 32-разрядных целых чисел со знаком. Это также относится и к промежуточным результатам. Если какой-либо промежуточный результат переполняет 32-разрядное целое со знаком (больше 2147483647 или меньше -2147483648), окончательный результат также будет некорректным. Разрядная сетка в 32 бита обеспечивает большой диапазон для промежуточных результатов, однако все же будет разумным учитывать потенциальную возможность ее переполнения.

Если диапазона целых чисел не достаточно, либо вместо целых чисел в приложении планируется применять вещественные, на помощь приходит поддержка чисел в формате с плавающей точкой. Компилятор поддерживает 32-битные float-величины и константные выражения со многими математическими операторами, такими же, как и для целых выражений. Отметьте, что это свойство работает только для констант, а не для переменных. Работу с переменными выражениями в float-формате ИМС Propeller поддерживает через объекты FloatMath, устанавливаемые при инсталляции. Для более детальной информации См. «Присваивание констант '='», стр. 296; FLOAT, стр. 253; ROUND, стр. 351; и TRUNC, стр. 363, а также объекты FloatMath и FloatString

Атрибуты операторов

Операторы имеют следующие важные атрибуты, каждый из которых показан в следующих двух таблицах и подробно описан далее:

- Унарный/ Бинарный
- Обычный / Присваивания
- Постоянное и/или Переменное выражение
- Уровень Приоритета

Табл. 4-9: Математические и логические операторы					
Оператор	Использование Присваивания	Константа выражения[1]		Унарное ?	Описание, номер страницы
		Целые	Float		
=	всегда	н/д[1]	н/д[1]		Присваивание константам (только блоки CON), 296
:=	всегда	н/д[1]	н/д[1]		Присваивание переменным (только блоки PUB/PRI), 297
+	+=	✓	✓		Сложение, 298
+	никогда	✓	✓	✓	Положительное (+X); унарная форма Сложения, 298
-	-=	✓	✓		Вычитание, 299
-	если единств.	✓	✓	✓	Отрицание (-X), унарная форма Вычитания, 299
--	всегда			✓	Пре-декремент(--X) или пост-декремент(X--), 299
++	всегда			✓	Пре-инкремент(++X) или пост-инкремент (X++), 300
*	*=	✓	✓		Умножить и вернуть младшие 32 бита (знаковое), 301
**	**=	✓			Умножить и вернуть старшие 32 бита (знаковое),, 302
/	/=	✓	✓		Деление (знаковое), 302
//	//=	✓			Остаток о деления (знаковое), 303
#>	#>=	✓	✓		Ограничение по минимуму (знаковое), 303
<#	<#=	✓	✓		Ограничение по максимуму (знаковое), 304
^^	если единств.	✓	✓	✓	Квадратный корень, 304
||	если единств.	✓	✓	✓	Абсолютное значение, 305
~	всегда			✓	Распростр.знак бита 7 (~X) или пост-очистить все биты 0 (X~),305
~~	всегда			✓	Распростр.знак бита 15 (~~X) или пост-устан. все биты 1(X~~),306
~>	~>=	✓			Арифметический сдвиг вправо, 307
?	всегда			✓	Случайное число вперед (?X) или назад (X?), 308
|<	если единств.	✓		✓	Побитовое: Дешифр. значение (0 - 31) в простое-high-bit long, 309
>|	если единств.	✓		✓	Побитовое: Шифров.long в значение (0 - 32) как high-bit приор., 310
<<	<<=	✓			Побитовое: Сдвиг влево, 310
>>	>>=	✓			Побитовое: Сдвиг вправо, 311
<-	<-=	✓			Побитовое: Циклический сдвиг влево, 312
->	->=	✓			Побитовое: Циклический сдвиг вправо, 312
><	><=	✓			Побитовое: Реверс, 313
&	&=	✓			Побитовое: AND, 314
|	|=	✓			Побитовое: OR, 315
^	^=	✓			Побитовое: XOR, 316
!	если единств.	✓		✓	Побитовое: NOT, 317
AND	AND=	✓	✓		Логичекое: AND (переводит не-0 в -1), 317
OR	OR=	✓	✓		Логичекое: OR (переводит не-0 в -1), 318
NOT	если единств.	✓	✓	✓	Логичекое: NOT (переводит не-0 в -1), 319
==	===	✓	✓		Логичекое: Равно, 320
<>	<>=	✓	✓		Логичекое: Не равно, 321
<	<=	✓	✓		Логичекое: Меньше чем (знаковое), 322
>	>=	✓	✓		Логичекое: Больше чем (знаковое), 322
=<	=<=	✓	✓		Логичекое: Меньше либо равно (знаковое), 322
=>	=>=	✓	✓		ЛогичекоеБольше либо равно (знаковое), 323
@	никогда	✓		✓	Адрес идентификатора, 324
@@	никогда			✓	Адрес объекта плюс идентификатора, 324

[1] **Формы операторов с присваиванием недопустимы в константных выражениях.**

Табл. 4-10: Уровни приоритетов операторов			
Уровень	**Примеч.**	**Операторы**	**Имена операторов**
Высший (0)	Унарный	`--, ++, ~, ~~, ?, @, @@`	Инк/Декрем.,Очистить, Установить, Случайн., Адрес объекта/идентиф.
1	Унарный	`+, -, ^^, ||, |<, >|, !`	Положит, Отрицат., Квадр.корень,Абсол., Кодир,Раскодир, Побитн.NOT
2		`->, <-, >>, <<, ~>, ><`	Сдвиг/Цикл.сдвиг Вправо /Влево, Арифм.сдвиг Вправо, Реверс
3		`&`	Побитовое AND
4		`|, ^`	Побитовое OR, побитовое XOR
5		`*, **, /, //`	Умножить-младш, Умнож-старш, Разделить, Модуль
6		`+, -`	Сложить, Вычесть
7		`#>, <#`	Ограничение по Минимуму/Максимуму
8		`<, >, <>, ==, =<, =>`	Логичекое: Больше/Меньше, Не-/Равно, Больше/Меньше либо равно
9	Унарный	`NOT`	Логичекое NOT
10		`AND`	Логичекое AND
11		`OR`	Логичекое OR
Низший(12)		`=, :=,` все другие присваивания	Присваивание Констант/Переменных, присв. формы Бинарных Операт.

Унарный / Бинарный

Каждый оператор по своей природе является либо унарным, либо бинарным. Унарными операторами являются те, которые действуют лишь на один операнд. Например:

```
!Flag     ' bitwise NOT of Flag
^^Total   ' square root of Total
```

Бинарные операторы – это те, которые проводят действия с двумя операндами. Например:

```
X + Y     ' add X and Y
Num << 4  ' shift Num left 4 bits
```

Имейте в виду, что термин “бинарный оператор” означает “два операнда,” и не имеет ничего общего с операциями с двоичными числами. Чтобы отличать операторы, работающие с двоичными числами, для них мы будем использовать термин “Побитовое”.

Обычные / Присваивания

Обычные операторы, такие как Сложить ‘**+**’ и Сдвиг Влево ‘**<<**’, выполняют операцию над своими операндами и предоставляют результат для использования остальной части выражения без влияния на сами операнды. Операторы присваивания, однако, кроме

предоставления результата для использования остальной части выражения, записывают результат либо в переменную, над которой они выполняли операцию (унарные), либо в переменную, расположенную сразу слева (бинарные).

Вот примеры операторов присваивания:

```
Count++      ' (Unary) evaluate Count + 1
             ' and write result to Count
Data >>= 3   ' (Binary) shift Data right 3 bits
             ' and write result to Data
```

Для указания того, что бинарный оператор является оператором присваивания, существуют специальные формы записи, которые заканчиваются на знак равенства '**=**'. Унарные операторы не могут быть также и операторами присваивания; некоторые из них всегда присваивают значения, в то время как другие присваивают в особых ситуациях. Для более детальной информации см. приведенную выше Табл. 4-9 и описание операторов.

Большинство операторов присваивания могут использоваться лишь в рамках методов (блоков **PUB** и **PRI**). Единственным исключением является оператор присваивания константы '=', используемый лишь в рамках блоков **CON**.

Постоянное и/или Переменное выражение

Операторы, которые имеют атрибут в виде целого константного выражения, могут использоваться как в выражениях с переменными во время выполнения, так и в выражениях-константах на этапе компиляции. Операторы, имеющие в атрибуте выражение-константу формата float, могут использоваться только как константы, на этапе компиляции. Операторы без наличия операндов в виде констант могут быть использованы только во время выполнения в выражениях-переменных. Большинство операндов имеют обычную форму, без присваивания, которая позволяет использовать их как в выражениях-константах, так и в выражениях с переменными.

Уровень приоритета

Каждый оператор имеет присвоенный ему уровень приоритета, который определяет, когда он будет выполнен по отношению к остальным операторам в этом выражении. Например, общеизвестно, что алгебраические правила гласят, что операции умножения и деления выполняются раньше операций сложения и вычитания. Говорят, что операции умножения и деления имеют "более высокий приоритет", чем сложение и вычитание. Вдобавок, умножение и деление обладают свойством коммутативности;

они обладают одинаковым уровнем приоритета, и результат вычисления выражения не зависит от того, какая из операций была выполнена в первую очередь (сначала умножение, затем – деление, либо наоборот). Коммутативные операторы всегда вычисляются слева направо, за исключением случая, когда круглые скобки принудительно меняют этот порядок.

В ИМС Propeller используются такие же правила порядка выполнения операций, как и в алгебре: выражения вычисляются слева направо, за исключением случаев со скобками и различными уровнями приоритета операций.

Следуя этим правилам, в ИМС Propeller результат вычисления такого примера будет:

X = 20 + 8 * 4 – 6 / 2

...будет равным 49; то есть, 8 * 4 = 32, 6 / 2 = 3, и 20 + 32 – 3 = 49. Если Вы хотите изменить порядок вычисления выражения, используйте скобки, заключив в них необходимые части выражения.

Например:

X = (20 + 8) * 4 – 6 / 2

В этом примере сначала выполняются операции в скобках, 20 + 8, что дает результат решения выражения 109, в отличие от предыдущего 49.

В Табл. 4-10 указаны уровни приоритетов для каждого из операторов, начиная от самого высокого (уровень 0), заканчивая самым низким (уровень 12). Операторы с более высоким приоритетом выполняются перед таковыми с низким: умножение – перед сложением, абсолютность – перед умножением, и т.д. Единственное исключение может быть только в случае наличия круглых скобок, – они перекрывают приоритет любого уровня.

Промежуточные присваивания

Вычислительные алгоритмы в ИМС Propeller допускают и применяют операторы присваивания на промежуточных стадиях. Это называется “промежуточными присваиваниями” и может использоваться для выполнения сложных вычислений в более компактном коде. Например, следующее выражение зависит и от X, и от X + 1.

```
X := X - 3 * (X + 1) / ||(X + 1)
```

Это выражение может быть переписано с учетом преимущества промежуточного присваивания оператора инкремента:

```
X := X++ - 3 * X / ||X
```

Положив X равным -5, оба из этих выражений дадут решение -2, и оба в конце сохранят результат в X. Второе выражение, однако, выполняет вычисления, основываясь на промежуточных присваиваниях (часть X++) с тем, чтобы упростить остальную часть выражения. Оператор Инкремента '++' выполняется в первую очередь (наивысший приоритет) и инкрементирует X с -5 до -4. Поскольку это – оператор "пост-инкремента" (см. «Инкремент, пре- или пост- '++'», стр. 300), он сначала возвращает в выражение оригинальное значение X, то есть -5, а затем записывает в X новое значение, -4. Таким образом, часть выражения "X++ - 3…" становится "-5 – 3…" Затем выполняются операторы абсолютного умножения и деления, но значение X уже изменено, поэтому в этих операциях используется его новое значение, -4:

-5 – 3 * -4 / ||-4 → -5 – 3 * -4 / 4 → -5 – 3 * -1 → -5 – -3 = -2

Использование промежуточных присваиваний иногда может сжать сложное выражение из нескольких строк в одну, обеспечивая меньший размер кода и несколько более быстрое выполнение.

Далее в этой секции описывается каждый математический и логический оператор, показанный в Табл. 4-9, в приведенном там порядке.

Присваивание констант '='

Оператор Присваивания Констант используется только в блоках **CON**, для объявления констант на уровне компилятора. Например,

```
CON
   _xinfreq = 4096000
   WakeUp = %00110000
```

Этот код устанавливает идентификатор _xinfreq в 4096000, а идентификатор WakeUp – в %00110000. Во всей остальной части программы компилятор будет использовать эти числа в местах соответствующих им идентификаторов. См. **CON**, стр. 228.

Эти объявления представляют собой выражения-константы, поэтому для подсчета результирующего значения константы во время компиляции может использоваться множество обычных операторов. К примеру, возможно, более понятным было бы переписать предыдущий пример таким образом:

```
CON
 _xinfreq = 4096000
 Reset = %00100000
 Initialize = %00010000
 WakeUp = Reset & Initialize
```

Здесь WakeUp также устанавливается в %00110000 во время компиляции, но теперь для потенциальных читателей более понятно, что идентификатор WakeUp включает в себя двоичные коды для последовательностей Reset и Initialize этого приложения.

Приведенные примеры создают 32-х битные целые константы со знаком; однако, также возможно создать и 32-х битные константы в формате с плавающей точкой (float). Float-константа вводится одним из трех путей: 1)Десятичные цифры, сопровождаемые десятичной точкой и как минимум еще одной десятичной цифрой, 2)десятичные цифры, сопровождаемые “e” (экспонента) и целая величина показателя экспоненты, 3)комбинация 1 и 2 Например:

```
CON
  OneHalf = 0.5
  Ratio = 2.0 / 5.0
  Miles = 10e5
```

Приведенный код создает три float-константы. OneHalf равна 0.5, Ratio равна 0.4 и Miles равна 1000000. Отметьте, что если бы Ratio была определена как 2 / 5, а не 2.0 / 5.0, выражение бы рассматривалось как целая константа, равная 0. В объявлении float-выражений-констант каждое значение в выражении должно быть типа float; не допускается смешивать целые- и float-величины, как к примеру Ratio = 2 / 5.0. Однако, Вы можете использовать декларацию FLOAT для преобразования целой величины в формат float, например Ratio = FLOAT(2) / 5.0.

Компилятор Propeller рассматривает константы с плавающей точкой как вещественные числа одинарной точности, как описано в стандарте IEEE-754. Вещественные числа одинарной точности хранятся в 32 битах, с 1 битом на знак, 8-битной экспонентой и 23-битной мантиссой (дробная часть). Это обеспечивает примерно 7.2 значащих десятичных разряда.

Для выполнения операций с float-числами, объекты FloatMath, FloatString, Float32 и Float32Full предоставляют математические функции, совместимые с числами одинарной точности.

См. FLOAT, стр. 253; ROUND, стр. 351; TRUNC, стр. 363, а также объекты FloatMath, FloatString, Float32 и Float32Full для более детальной информации.

Присваивание переменных ‘:=’

Оператор Присваивания Переменных используется только в методах (блоках PUB и PRI), для присваивания значения переменной. Например:

```
Temp := 21
Triple := Temp * 3
```

Во время выполнения этого примера, переменной Temp присваивается значение 21, а переменной Triple – значение 21 * 3, то есть 63.

Как и для других операторов присваивания, оператор присваивания переменных может использоваться внутри выражений для присваивания промежуточных результатов, например:

```
Triple := 1 + (Temp := 21) * 3
```

В этом примере сначала устанавливается Temp в 21, затем Temp умножается на 3 и складывается с 1, в конце результат вычислений, 64, присваивается переменной Triple.

Сложение '+', '+='

Оператор Сложения складывает значения двух величин. Сложение может использоваться как для констант, так и для переменных. Пример:

```
X := Y + 5
```

Сложение имеет форму с присваиванием, **+=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата.

Например,

```
X += 10  'Short form of X := X + 10
```

Здесь значение X складывается с 10, а результат сохраняется назад в X. Присваиваемая форма оператора Сложения может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Положительное '+' (унарная форма Сложения)

Оператор Положительное – это унарная форма оператора Сложения, и используется аналогично оператору Отрицание, за тем исключением, что у оператора Положительное нет формы с присваиванием., На самом деле оператор Положительное компилятором игнорируется, но он удобен, когда при написании выражения важно подчеркнуть знак операнда. Например:

```
Val := +2 - A
```

Вычитание '-', '-='

Оператор Вычитания вычитает одну величину из другой. Вычитание может использоваться как для констант, так и для переменных. Пример:

```
X := Y - 5
```

Вычитание имеет форму с присваиванием, **-=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X -= 10  'Short form of X := X - 10
```

Здесь 10 вычитается из значения переменной X, а результат сохраняется назад в X. Присваиваемая форма оператора Вычитания может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Отрицание '-' (унарная форма Вычитания)

Оператор Отрицания – это унарная форма оператора Вычитания. Отрицание изменяет знак операнда, стоящего справа от него, на противоположный; положительная величина становится отрицательной, а отрицательная – положительной. Например:

```
Val := -2 + A
```

Отрицание становится оператором присваивания, когда он является единственным оператором слева от переменной на той же строке. Например:

```
-A
```

Этот код меняет знак величины A и сохраняет результат назад в A.

Декремент, пре- или пост- '--'

Оператор Декремента – это особый оператор мгновенного действия, который уменьшает значение переменной на единицу и присваивает результат этой же переменной. Он может использоваться лишь в выражениях-переменных, вычисляемых во время выполнения. Оператор Декремента имеет две формы: пре- декремент и пост-декремент, в зависимости от того, с какой стороны от переменной он введен. Если оператор введен слева от переменной, то это – пре-декремент, если же справа – пост-декремент. Эти операторы чрезвычайно полезны при программировании, поскольку часто возникают ситуации, когда требуется декремент значения переменной прямо перед или сразу после ее использования. Например:

```
Y := --X + 2
```

Выше приведена форма оператора пре-декремента; это значит “декрементировать перед предоставлением значения для следующей операции”. Оператор декрементирует значение переменной X на единицу, записывает этот результат назад в X, и передает его остальной части выражения. Если в этом примере X в начале был равен 5, оператор --X преобразует его значение X в 4, затем вычисляется выражение 4 + 2 и, в конце, результат, 6, записывается в переменную Y. После вычисления выражения переменная X равна 4, а Y равен 6.

```
Y := X-- + 2
```

Эта форма оператора – пост-декремент, что значит “декрементировать после предоставления значения следующей операции”. Оператор предоставляет текущее значение X следующей операции выражения, затем декрементирует значение X на единицу и записывает результат в X. Если в этом примере X в начале был равен 5, оператор X-- сначала передаст текущее значение в выражение для следующей операции (5 + 2), а затем сохранит 4 в X. Затем вычисляется выражение 5 + 2 и результат, 7, сохраняется в Y. После вычислений X равен 4, а Y равен 7.

Поскольку Декремент – это всегда оператор с присваиванием, к нему применимы правила из секции «Промежуточные присваивания» (см стр. 295). Допустим, для дальнейших примеров, в начале X равен 5.

```
Y := --X + X
```

Здесь X сначала устанавливается в 4, затем вычисляется 4 + 4 и Y устанавливается в 8.

```
Y := X-- + X
```

Здесь текущее значение X, 5, сохраняется для следующей операции (Сложения), а сама переменная X декрементируется до 4, затем вычисляется 5 + 4, и Y устанавливается в 9.

Инкремент, пре- или пост- ‘++’

Оператор Инкремента – это особый оператор мгновенного действия, который увеличивает значение переменной на единицу и присваивает результат этой же переменной. Он может использоваться лишь в выражениях-переменных, вычисляемых во время выполнения. Оператор Инкремента имеет две формы: пре-инкремент и пост-инкремент, в зависимости от того, с какой стороны от переменной он введен. Если оператор введен слева от переменной, то это – пре-инкремент, если же справа – пост-инкремент. Эти операторы чрезвычайно полезны при программировании, поскольку

часто возникают ситуации, когда требуется инкремент значения переменной прямо перед или сразу после ее использования. Например:

```
Y := ++X - 4
```

Выше приведена форма оператора пре-инкремента; это значит “инкрементировать перед предоставлением значения для следующей операции”. Оператор инкрементирует значение переменной X на единицу, записывает этот результат назад в X, и передает его остальной части выражения. Если в этом примере X в начале был равен 5, оператор ++X преобразует его значение в 6, затем вычисляется выражение 6 - 2 и, в конце, результат, 2, записывается в переменную Y. После вычисления выражения переменная X равна 6, а Y равен 2.

```
Y := X++ - 4
```

Эта форма оператора – пост-инкремент, что значит “инкрементировать после предоставления значения следующей операции”. Оператор предоставляет текущее значение X следующей операции выражения, затем инкрементирует значение X на единицу и записывает результат в X. Если в этом примере X в начале был равен 5, оператор X++ сначала передаст текущее значение в выражение для следующей операции (5 - 4), а затем сохранит 6 в X. Затем вычисляется выражение 5 - 4 и результат, 1, сохраняется в Y. После вычислений X равен 6, а Y равен 1.

Поскольку Инкремент – это всегда оператор с присваиванием, к нему применимы правила из секции «Промежуточные присваивания» (см стр. 295). Допустим, для дальнейших примеров, в начале X равен 5.

```
Y := ++X + X
```

Здесь X сначала устанавливается в 6, затем вычисляется 6 + 6, и Y устанавливается в 12.

```
Y := X++ + X
```

Здесь текущее значение X, 5, сохраняется для следующей операции (Сложения), а сама переменная X инкрементируется до 6, затем вычисляется 5 + 6 и Y устанавливается в 11.

Умножение, Вернуть младшее ‘*’, ‘*=’

Этот оператор также называется Умножить Младшее или просто Умножение. Умножение может использоваться как для констант, так и для переменных. Когда оператор используется с выражениями-переменными либо целыми выражениями-константами, Умножить Младшее умножает два значения одно на другое и возвращает младшие 32 бита 64-битного результата. Когда оператор используется с float-

выражениями-константами, Умножить Младшее умножает два значения одно на другое и возвращает 32-битный результат в виде числа в формате с плавающей точкой одинарной точности. Пример:

```
X := Y * 8
```

Умножить Младшее имеет форму с присваиванием, ***=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X *= 20  'Short form of X := X * 20
```

Здесь значение переменной X умножается на 20, и младшие 32 бита результата сохраняются назад в X. Присваиваемая форма оператора Умножить Младшее может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295

Умножение, Вернуть старшее '**', '**='

Этот оператор также называется Умножить Старшее. Может использоваться для целых констант и переменных, но не для float-выражений-констант. Умножить Старшее умножает два значения одно на другое и возвращает старшие 32 бита 64-битного результата. Пример:

```
X := Y ** 8
```

Если переменная Y в начале была равна 536,870,912 (2^{29}) то Y ** 8 даст 1 – значение старших 32 бит результата.

Умножить Старшее имеет форму с присваиванием, ****=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X **= 20 'Short form of X := X ** 20
```

Здесь значение X умножается на 20 и старшие 32 бита результата сохраняются в X. Присваиваемая форма оператора Умножить Старшее может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Деление '/', '/='

Оператор Деления может использоваться как для констант, так и для переменных. При использовании с целыми переменными или константами, он делит одно значение на другое и возвращает 32-битный результат в виде целого. При использовании с float-

константами, он делит одно значение на другое и возвращает 32-битный результат в виде числа в формате с плавающей точкой одинарной точности. Пример:

```
X := Y / 4
```

Деление имеет форму с присваиванием, **/=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X /= 20  'Short form of X := X / 20
```

Здесь величина X делится на 20 и целый результат сохраняется назад в X. Присваиваемая форма оператора Деления может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Остаток от деления Mod '//', '//='

Оператор Mod может использоваться для целых констант и переменных, но не для float-выражений-констант. Mod делит одно значение на другое и возвращает 32-битное целое значение остатка. Пример:

```
X := Y // 4
```

Если Y равен 5, то Y // 4 даст результат 1, что означает, что при делении 5 на 4 получаем в результате вещественное число с остатком ¼, или .25.

Mod имеет форму с присваиванием, //=, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X //= 20 'Short form of X := X // 20
```

Здесь значение X делится на 20 и 32-битный целый остаток сохраняется назад в X. Присваиваемая форма оператора Mod может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Ограничение по минимуму '#>', '#>='

Оператор Ограничения по минимуму сравнивает два значения и возвращает большее из них. Может использоваться как для констант, так и для переменных. Пример:

```
X := Y - 5 #> 100
```

В этом примере из значения Y вычитается величина 5 и производится ограничение результата по минимальному значению величиной 100. Если Y равен 120, то 120 – 5 =

115; это больше, чем 100, поэтому X устанавливается в 115. Если же Y равен 102, то 102 – 5 = 97; это меньше, чем 100, поэтому в этом случае X устанавливается в 100.

Ограничение по минимуму имеет форму с присваиванием, **#>=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X #>= 50 'Short form of X := X #> 50
```

Здесь значение X ограничивается по минимуму значением 50 и результат сохраняется назад в X. Присваиваемая форма оператора Ограничения по минимуму может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Ограничение по максимуму '<#', '<#='

Оператор Ограничения по максимуму сравнивает два значения и возвращает меньшее из них. Может использоваться как для констант, так и для переменных. Пример:

```
X := Y + 21 <# 250
```

В этом примере к значению Y прибавляется величина 21 и производится ограничение по максимальному значению 250. Если Y равен 200, то 200 + 21 = 221; это меньше, чем 250, поэтому X устанавливается в 221. Если же Y равен 240, то240 + 21 = 261; это больше 250, поэтому X в этом случае устанавливается в 250.

Ограничение по максимуму имеет форму с присваиванием, **<#=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X <#= 50 'Short form of X := X <# 50
```

Здесь значение X ограничивается максимальным значением 50 и результат сохраняется обратно в X. Присваиваемая форма оператора Ограничения по максимуму может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Квадратный Корень '^^'

Оператор Квадратного Корня возвращает значение квадратного корня от величины. Может использоваться как для констант, так и для переменных. При использовании этого оператора с переменными или целыми константами, он возвращает результат в виде 32-битного отсеченного целого. При использовании с float-константами, он

возвращает результат в виде 32-битного значения в формате с плавающей точкой одинарной точности. Пример:

```
X := ^^Y
```

Квадратный Корень становится оператором с присваиванием, когда он – единственный оператор слева от переменной в этой строке. Например:

```
^^Y
```

Этот код сохранит значение квадратного корня переменной Y назад в Y.

Абсолютное значение '||'

Оператор Абсолютного Значения, также называемый *Absolute*, возвращает абсолютное значение числа (в положительной форме). Оператор Абсолютное Значение может использоваться как в константах, так и в выражениях с переменными. При использовании в выражениях с целыми переменными или константами, *Absolute* возвращает 32-битный целый результат. При использовании с float-выражениями-константами, оператор абсолютного значения возвращает результат в виде 32-битного числа в формате с плавающей точкой одинарной точности. Пример:

```
X := ||Y
```

Если Y равен -15, то абсолютное значение, 15, будет сохранено в X.

Оператор Абсолютного Значения становится оператором с присваиванием, когда он – единственный оператор слева от переменной в этой строке. Например:

```
||Y
```

Этот код сохранит абсолютное значение Y назад в переменную Y.

Распространение Знака 7 или Пост-Очистка '~'

Этот оператор – специальный оператор мгновенного действия, который выполняет два различных действия, в зависимости от того, с какой стороны от переменной он введен. Он может использоваться лишь в выражениях-переменных, вычисляемых во время выполнения. Если оператор введен слева от переменной, то это – Распространение знака 7, если же справа – Пост-Очистка.

Вот пример его использования в виде оператора Распространения Знака 7:

```
Y := ~X + 25
```

Оператор Распространения Знака 7 в этом примере распространяет знак величины, X в этом случае, от бита 7 до бита 31. Целое 32-битное число со знаком хранится в памяти в дополнительном коде (*twos-complement*), и самый старший бит (31) указывает знак величины (положительная либо отрицательная). Встречаются задачи, в которых вычисления с простыми данными дают результат в виде однобайтовой целой величины со знаком, с диапазоном от -128 до +127. Когда Вам необходимо произвести дальнейшие вычисления с такими байтовыми данными, используйте оператор Распространения Знака 7 для преобразования этих чисел в надлежащий формат 32-битного целого со знаком. Допустим, в предыдущем примере, X представляет величину -20, которая в 8-битном доп-коде дает число 236 (%11101100). Часть выражения ~X распространяет знак из бита 7 во все биты до бита 31, преобразуя это число в надлежащий формат 32-битного целого со знаком в доп. коде: -20 (%11111111 11111111 11111111 11101100). Складывая это преобразованное значение с величиной 25, мы получим ожидаемый результат, 5, в то время как без надлежащего преобразования знака мы бы получили результат 261.

Далее приведем пример формы Пост-Очистка этого оператора.

```
Y := X~ + 2
```

Оператор Пост-Очистка в этом примере очищает переменную в 0 (все биты в 0) после того, как передаст ее текущее значение следующей операции. В этом случае если X в начале равна 5, то X~ сначала предоставит текущее значение выражению (5 + 2) для дальнейших вычислений, а затем сохраняет 0 в X. Далее вычисляется выражение 5 + 2 и результат, 7, сохраняется в Y. После вычислений X равен 0, а Y равен 7.

Поскольку операторы Распространения Знака 7 и Пост- Очистки всегда обладают свойством присваивания, они подчиняются правилам, приведенным в секции «Промежуточные присваивания», стр. 295.

Распространение Знака 15 или Пост- Установка '~~'

Этот оператор – это специальный оператор мгновенного действия, который выполняет два различных действия, в зависимости от того, с какой стороны от переменной он введен. Он может использоваться лишь в выражениях-переменных, вычисляемых во время выполнения. Если оператор введен слева от переменной, то это – Распространение знака 15, если же справа – Пост- Установка.

Вот пример его использования в виде оператора Распространения Знака 15:

```
Y := ~~X + 50
```

Оператор Распространения Знака 15 в этом примере распространяет знак величины, X в этом случае, от бита 15 до бита 31. Целое 32-битное число со знаком хранится в памяти в дополнительном коде (*twos-complement*), и самый старший бит (31) указывает знак величины (положительная либо отрицательная). Встречаются задачи, в которых вычисления с простыми данными дают результат в виде целой величины со знаком размером в слово, с диапазоном от -32768 to +32767. Когда Вам необходимо произвести дальнейшие вычисления с такими данными, используйте оператор Распространения Знака 15 для преобразования этих чисел в надлежащий формат 32-битного целого со знаком. В приведенном примере, допустим, X представляет величину -300, которая в 16-битном доп-коде даст значение 65236 (%11111110 11010100). Часть выражения ~~X распространяет знак из бита 15 во все биты до бита 31, преобразуя число в надлежащий формат 32-битного целого со знаком в доп-коде: -300 (%11111111 11111111 11111110 11010100). Складывая это преобразованное значение с величиной 50, мы получим ожидаемый результат, -250, в то время как без надлежащего преобразования знака мы бы получили результат 65286.

Далее приведем пример формы Пост-Установка этого оператора.

```
Y := X~~ + 2
```

Оператор Пост- Установки в этом примере устанавливает значение переменной в -1 (все биты в 1) после того, как предоставит ее текущее значение следующей операции. В этом случае если X в начале была равна 6, X~~ сначала предоставит это текущее значение выражению (6 + 2) для дальнейших вычислений, а затем сохранит -1 в X. Затем вычисляется выражение 6 + 2 и результат, 8, сохраняется в Y. В результате вычислений X равен -1, а Y равен 8.

Поскольку операторы Распространения Знака 15 и Пост- Установки всегда обладают свойством присваивания, они подчиняются правилам , приведенным в секции «Промежуточные присваивания», стр. 295.

Арифметический Сдвиг Вправо '~>', '~>='

Оператор Арифметического Сдвига Вправо – такой же, как и оператор Сдвиг Вправо, за исключением того, что он сохраняет знак, как при делении знаковой величины на 2, 4, 8, и т.д. Арифметический Сдвиг Вправо может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
X := Y ~> 4
```

В этом примере производится сдвиг числа вправо на 4 бита, сохраняя при этом знак. Если Y равен -3200 (%11111111 11111111 11110011 10000000), то -3200 ~> 4 = -200

(%11111111 11111111 11111111 00111000). Если же эта операция проводилась бы с оператором Сдвига Вправо, результат был бы 268435256 (%00001111 11111111 11111111 00111000).

Арифметический Сдвиг Вправо имеет форму с присваиванием, **~>=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X ~>= 2  'Short form of X := X ~> 2
```

Здесь значение X сдвигается вправо на 2 бита, сохраняя знак, а результат сохраняется назад в X. Присваиваемая форма оператора Арифметического Сдвига Вправо может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Случайное '?'

Оператор Случайное – это специальный оператор мгновенного действия, который использует значение переменной как базу для генерации псевдо-случайного числа и присваивает это значение этой же переменной. Он может использоваться лишь в выражениях-переменных, вычисляемых во время выполнения. Оператор Случайное имеет две формы, прямую и обратную, в зависимости от того, с какой стороны от переменной он введен. Если он введен слева от переменной – это прямая форма, иначе, если он введен справа – это обратная форма.

Оператор Случайное генерирует псевдо-случайные числа в диапазоне от -2147483648 до +2147483647. Числа называются "псевдо-случайными", поскольку хотя они и кажутся случайными, но на самом деле генерируются логическими операциями, которые используют "базовое" значение как отвод от последовательности из более 4 миллионов действительно случайных чисел. Если будет использоваться то же самое значение в качестве базы, будет генерироваться такая же последовательность чисел. Выход генератора случайных чисел у ИМС Propeller является реверсивным. На самом деле, технически, это 32-битный (максимум) сдвиговый регистр с линейной обратной связью с четырьмя отводами, отводы присутствуют как на бите LSB (*Least Significant Bit*, самый правый бит), так и на бите MSB (*Most Significant Bit*, самый левый бит), что позволяет осуществлять реверсивное функционирование.

Рассматривайте генерируемую псевдо-случайную последовательность как простой статический список из более чем 4 миллионов чисел. Начиная с определенного базового значения и продвигаясь вперед мы получим список оригинальных наборов чисел. Если же Вы возьмете последнее сгенерированное значение и используете его в качестве первого базового значения при движении назад, Вы в конце получите

перечень тех же значений, таких же, как и ранее, но в обратном порядке. Это удобно для многих приложений.

Вот пример:

```
?X
```

Здесь оператор Случайное использован в прямой форме, он использует текущее значение переменной X для получения следующего псевдо-случайного числа в прямом направлении и сохраняет его назад в переменной X. Выполнив ?X вновь, мы получим следующее псевдо-случайное число, отличное от предыдущего, опять сохраненное в переменной X.

```
X?
```

В этом примере оператор Случайное использован в его обратной форме; он использует текущее значение переменной Xдля получения следующего псевдо-случайного числа в обратном направлении и сохраняет его обратно в переменной X. Повторное выполнение X? приведет к генерации нового псевдо-случайного числа, отличного от предыдущего, и сохранению его в переменной X.

Поскольку оператор Случайное всегда обладают свойством присваивания, он подчиняется правилам, приведенным в секции «Промежуточные присваивания» (см. стр. 295).

Побитовое Дешифровать '|<'

Оператор Побитовое Дешифровать преобразует величину (0 – 31) в 32-битную величину с одним установленным битом в позиции, соответствующей оригинальному значению. Может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
Pin := |<PinNum
```

В этом примере Pin устанавливается равным 32-битному значению с единственным взведенный битом, соответствующим позиции, указанной в переменной PinNum.

Если PinNum равен 3, Pin устанавливается в %00000000 00000000 00000000 00001000.

Если PinNum равен 31, Pin устанавливается в %10000000 00000000 00000000 00000000.

Оператор Побитного Дешифрования имеет множество применений, но одно из наиболее полезных – это преобразование из номера пина в 32-битное значение,

описывающее этот пин по отношению к регистрам B/B. Например, побитовое дешифрование очень удобно для получения параметра маски команд **WAITPEQ** и **WAITPNE**.

Побитовое Дешифрование становится оператором с присваиванием, когда он – единственный оператор слева от переменной в этой строке. Например:

```
|<PinNum
```

Этот код сохраняет дешифрованное значение PinNum назад в PinNum.

Побитовое Шифровать '>|'

Оператор Двоичного Шифрования кодирует 32-битную *long* величину в величину (0 – 32), которая представляет номер наивысшего установленного бита плюс 1. Побитовое Шифрование может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
PinNum := >|Pin
```

В этом примере PinNum устанавливается равным номеру наивысшего установленного бита в Pin, плюс 1.

Если Pin равен %00000000 00000000 00000000 00000000, то PinNum равен 0.

Если Pin равен %00000000 00000000 00000000 10000000, то PinNum равен 8.

Если Pin равен %10000000 00000000 00000000 00000000, то PinNum равен 32.

Если Pin равен %00000000 00010011 00010010 00100000, то PinNum равен 21.

Побитовое Сдвиг Влево '<<', '<<='

Оператор Побитного Сдвига Влево сдвигает биты первого операнда влево на количество битов, указанное во втором операнде. Биты MSB (самые левые) исходного операнда портятся, а биты LSB (самые правые) становятся нолями. Побитовый сдвиг влево может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
X := Y << 2
```

Если Y в начале был:

%10000000 01110000 11111111 00110101

...после побитного сдвига влево на два бита и записи в переменную X, получим:

%00000001 11000011 11111100 11010100

Поскольку природа двоичного исчисления базируется на основании 2, сдвиг величины влево аналогичен умножению этой величины на 2^b, где b – количество сдвигаемых бит.

Оператор Побитного Сдвига Влево имеет форму с присваиванием, **<<=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X <<= 4   'Short form of X := X << 4
```

Здесь значение X сдвигается влево на 4 бита и сохраняется назад в переменную X. Присваиваемая форма оператора Побитного Сдвига Влево может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Побитовый Сдвиг Вправо '>>', '>>='

Оператор Побитового Сдвига Вправо сдвигает биты первого операнда вправо на количество битов, указанное во втором операнде. Биты LSB (самые правые) исходного операнда портятся, а биты MSB (самые левые) становятся нолями. Побитовый сдвиг вправо может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
X := Y >> 3
```

Если Y в начале был:

%10000000 01110000 11111111 00110101

...после побитного сдвига вправо на три бита и записи в переменную X, получим:

%00010000 00001110 00011111 11100110

Поскольку природа двоичного исчисления базируется на основании 2, сдвиг величины вправо аналогичен целочисленному делению этой величины на 2^b, где b – количество сдвигаемых бит.

Оператор Побитового Сдвига Вправо имеет форму с присваиванием, **>>=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X >>= 2   'Short form of X := X >> 2
```

Здесь значение X сдвигается вправо на 2 бита и сохраняется назад в переменную X. Присваиваемая форма оператора Побитового Сдвига Вправо может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Побитовый Циклический Сдвиг Влево '<-', '<-='

Оператор Побитового Циклического Сдвига Влево идентичен оператору Побитового Сдвига Влево, за исключением того, что биты MSB (самые левые) при сдвиге переносятся назад по кругу в биты LSB (самые правые). Побитовый Циклический Сдвиг Влево может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
X := Y <- 4
```

Если Y в начале был:

%10000000 01110000 11111111 00110101

оператор побитового циклического сдвига влево циклически сдвинет эту величину влево на 4 бита, перемещая 4 MSB бита исходной величины в четыре новых бита LSB, устанавливая X в:

%00000111 00001111 11110011 01011000

Оператор Побитового Циклического Сдвига Влево имеет форму с присваиванием, **<-=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X <-= 1  'Short form of X := X <- 1
```

Здесь значение X циклически сдвигается на один бит влево и сохраняется назад в X. Присваиваемая форма оператора Побитового Циклического Сдвига Влево может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Побитовый Циклический Сдвиг Вправо'->', '->='

Оператор Побитового Циклического Сдвига Вправо идентичен оператору Побитового Сдвига Вправо, за исключением того, что биты LSB (самые правые) при сдвиге переносятся назад по кругу в биты MSB (самые левые). Побитовый Циклический Сдвиг Вправо может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
X := Y -> 5
```

Если Y в начале был:

%10000000 01110000 11111111 00110101

... оператор побитового циклического сдвига вправо циклически сдвинет эту величину вправо на 5 бит, перемещая 5 LSB-битов исходной величины в пять новых битов MSB, устанавливая X в:

%10101100 00000011 10000111 11111001

Оператор Побитового Циклического Сдвига Вправо имеет форму с присваиванием, **->=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X ->= 3   'Short form of X := X -> 3
```

Здесь значение X циклически сдвигается вправо на три бита и сохраняется назад в X. Присваиваемая форма оператора Побитового Циклического Сдвига Вправо может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Побитовое Обратное значение '><', '><='

Оператор Побитового Обратного значения возвращает младшие биты первого операнда, общее количество которых указано во втором операнде, в обратном порядке. Все остальные биты слева от измененных обращаются в ноли. Побитовое Обратное значение может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
X := Y >< 6
```

Если Y в начале был:

%10000000 01110000 11111111 00110101

...оператор Побитового Обратного значения вернет шесть битов LSB в обратном порядке следования со всеми остальными битами сброшенными в ноль, устанавливая X в:

%00000000 00000000 00000000 00101011

Оператор Побитового Обратного значения имеет форму с присваиванием, **><=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X ><= 8   'Short form of X := X >< 8
```

Здесь обращены восемь битов LSB значения X, все остальные биты сброшены в ноль, а результат записан назад в X. Присваиваемая форма оператора Побитового Обратного

значения может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Побитовое И (AND) '&', '&='

Оператор Побитового И выполняет побитовое И битов первого операнда с битами второго операнда. Побитовое И может использоваться для целых констант и переменных, но не для float-выражений-констант.

Каждый бит обоих операндов подчиняется следующей логике:

Табл. 4-11: Таблица истинности Побитовое И

Состояние бита		Результат
0	0	0
0	1	0
1	0	0
1	1	1

Пример:

```
X := %00101100 & %00001111
```

В этом примере выполняется операция побитового И величин %00101100 и %00001111, и запись результата, %00001100, в X.

Оператор Побитового И имеет форму с присваиванием, **&=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X &= $F   'Short form of X := X & $F
```

Здесь значение X умножается побитно по И с $F и результат сохраняется назад в X. Присваиваемая форма оператора Побитового И может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Будьте внимательны, чтобы не путать оператор побитового И с оператором логического И ('AND'). Побитовое И предназначено для операций с битами, в то время как логическое И используется для операций сравнения (см. стр. 317).

Побитовое ИЛИ (OR) '|', '|='

Оператор Побитового ИЛИ, **|,** выполняет побитовое ИЛИ битов первого операнда с битами второго операнда. Побитовое ИЛИ может использоваться для целых констант и переменных, но не для float-выражений-констант.

Каждый бит обоих операндов подчиняется следующей логике:

Табл. 4-12: Таблица истинности Побитовое ИЛИ

Состояние бита		Результат
0	0	0
0	1	1
1	0	1
1	1	1

Пример:

```
X := %00101100 | %00001111
```

В этом примере выполняется операция побитового ИЛИ величин %00101100 и %00001111, , и запись результата, %00101111, в X.

Оператор Побитового ИЛИ имеет форму с присваиванием, **|=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X |= $F   'Short form of X := X | $F
```

Здесь значение X складывается побитно с $F, а результат сохраняется назад в X. Присваиваемая форма оператора Побитового ИЛИ может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Будьте внимательны, чтобы не путать оператор побитового ИЛИ, '|', с оператором логического ИЛИ ('OR'). Побитовое ИЛИ предназначено для операций с битами, в то время как логическое ИЛИ используется для операций сравнения (см. стр. 317).

Побитовое ИСКЛЮЧАЮЩЕЕ–ИЛИ (XOR) '^', '^='

Оператор Побитового ИСКЛЮЧАЮЩЕЕ-ИЛИ, **^,** выполняет побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ битов первого операнда с битами второго операнда. Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ может использоваться для целых констант и переменных, но не для float-выражений-констант.

Каждый бит обоих операндов подчиняется следующей логике:

Табл. 4-13: Таблица истинности Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ

Состояние бита		Результат
0	0	0
0	1	1
1	0	1
1	1	0

Пример:

```
X := %00101100 ^ %00001111
```

В этом примере выполняется операция побитового XOR величины %00101100 с %00001111, и запись результата , %00100011, назад в X.

Оператор Побитового ИЛИ имеет форму с присваиванием, **^=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X ^= $F   'Short form of X := X ^ $F
```

Здесь значение X складывается по XOR с $F, а результат сохраняется назад в X. Присваиваемая форма оператора Побитового ИСКЛЮЧАЮЩЕЕ-ИЛИ может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Побитовое НЕ (NOT) '!'

Оператор Побитового НЕ, **!,** выполняет побитовое НЕ (инверсию, дополнение до 1), следующего за ним операнда. Побитовое НЕ может использоваться для целых констант и переменных, но не для float-выражений-констант.

Каждый бит операнда подчиняется следующей логике:

Табл. 4-14: Таблица истинности Побитовое НЕ

Состояние бита	Результат
0	1
1	0

Пример:

```
X := !%00101100
```

Здесь величина %00101100 преобразуется по НЕ (инвертируется), а результат, %11010011, сохраняется в X.

Побитовое НЕ становится оператором с присваиванием, когда он – единственный оператор слева от переменной в этой строке. Например:

```
!Flag
```

Этот код сохранит инвертированное значение `Flag` назад в переменную `Flag`.

Будьте внимательны, чтобы не путать оператор побитового НЕ, '**!**', с оператором логического НЕ ('NOT'). Побитовое НЕ предназначено для операций с битами, в то время как логическое НЕ используется для операций сравнения (см. стр. 317).

Логическое И (AND) 'AND', 'AND='

Логический оператор И ('AND') сравнивает два операнда и возвращает TRUE (-1), если оба значения TRUE (не ноль), либо возвращает FALSE (0), если один или оба операнда FALSE (0). Логическое И может использоваться как для констант, так и для переменных. Пример:

```
X := Y AND Z
```

В этом примере сравнивается значение Y со значением Z, и значение X устанавливается: TRUE (-1), если и Y, и Z не равны нолю, либо FALSE (0), если либо Y, либо Z равно нолю. При сравнении этот оператор представляет каждое из двух значений как -1, если они не нулевые, делая таким образом любое ненулевое значение равным -1, при этом условие сравнения звучит как: "Если Y – истина, и Z – истина…"

Часто этот оператор используется в комбинации с другими операторами сравнения, такими, как в следующем примере.

```
IF (Y == 20) AND (Z == 100)
```

В этом примере вычисляется результат Y == 20 по сравнению с Z == 100, и если оба имеют значение истина (TRUE), логический оператор И возвратит TRUE (-1).

Оператор Логического И имеет форму с присваиванием, AND=, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X AND= True     'Short form of X := X AND True
```

Здесь значение X рассматривается как TRUE, если не равно нолю, затем оно сравнивается с TRUE, и логический результат (TRUE / FALSE, -1 / 0) сохраняется назад в X. Присваиваемая форма оператора Логического И может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Будьте внимательны, чтобы не путать оператор логического И ('AND') с оператором побитового И. Логическое И используется для операций сравнения, в то время как побитовое И предназначено для операций с битами (см. стр. 317).

Логическое ИЛИ (OR) 'OR', 'OR='

Логический оператор ИЛИ ('OR') сравнивает два операнда и возвращает TRUE (-1), если какой либо из операндов TRUE (не ноль), либо возвращает FALSE (0), если оба операнда FALSE (0). Логическое ИЛИ может использоваться как для констант, так и для переменных. Пример:

```
X := Y OR Z
```

В этом примере производится сравнение значения Y со значением Z и устанавливается значение X, равное: либо TRUE (-1), если Y или Z не равны нолю, или FALSE (0), если и Y, и Z равны нолю. При сравнении каждое из двух значений представляется как -1, если они не равны нолю, делая таким образом любое значение, не равное нолю, равным -1, при этом условие сравнения звучит как: "Если Y – истина или Z – истина …"

Часто этот оператор используется в комбинации с другими операторами сравнения, такими, как в следующем примере.

```
IF (Y == 1) OR (Z > 50)
```

В этом примере вычисляется результат `Y == 1` по сравнению с `Z > 50`, и если один из них имеет значение истина (**TRUE**), логический оператор ИЛИ возвратит **TRUE** (-1).

Оператор Логического И имеет форму с присваиванием, **OR=**, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X OR= Y      'Short form of X := X OR Y
```

Здесь значение `X` представляется как **TRUE,** если не равно нолю, далее оно сравнивается с `Y` (которое также **TRUE,** если не ноль), и затем логический результат (**TRUE** / **FALSE**, -1 / 0) сохраняется назад в `X`. Присваиваемая форма оператора Логического ИЛИ может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Будьте внимательны, чтобы не путать оператор логического ИЛИ (‘**OR**’) с оператором побитового ИЛИ, ‘**|**’. Логическое ИЛИ используется для операций сравнения, в то время как побитовое ИЛИ предназначено для операций с битами. (см. стр. 317).

Логическое НЕ (NOT) ‘NOT’

Логический оператор НЕ (‘**NOT**’) возвращает значение **TRUE** (-1), если значение операнда – **FALSE** (0), либо возвращает **FALSE** (0), если операнд – **TRUE** (не ноль). Логическое НЕ может использоваться как для констант, так и для переменных. Пример:

```
X := NOT Y
```

В этом примере возвращается величина, противоположная величине `Y`; **TRUE** (-1) если `Y` равен 0, либо **FALSE** (0) если `Y` – не ноль. При сравнении значение `Y` представляется как -1, если оно не ноль, делая любую величину, отличную от 0, равным -1, при этом условие сравнения звучит как: “Если НЕ истина ” или “Если НЕ ложь”

Часто этот оператор используется в комбинации с другими операторами сравнения, такими, как в следующем примере.

```
IF NOT ( (Y > 9) AND (Y < 21) )
```

В этом примере вычисляется результат (Y > 9 AND Y < 21), и возвращается логически противоположное результату значение; в этом случае – истина (TRUE, -1), если Y находится в диапазоне от 10 до 20.

Логическое НЕ становится оператором с присваиванием, когда он – единственный оператор слева от переменной в этой строке. Например:

```
NOT Flag
```

Этот код сохранит логически противоположное значение Flag назад в Flag.

Будьте внимательны, чтобы не путать оператор логического НЕ ('NOT'), с оператором побитового НЕ, '!'. Логическое НЕ используется для операций сравнения, в то время как побитовое НЕ предназначено для операций с битами (см. стр. 317).

Логическое Равенство (Is Equal) '==', '==='

Логический оператор Равенство сравнивает два операнда и возвращает TRUE (-1), если оба значения одинаковы, иначе возвращает FALSE (0). Оператор Равенства может использоваться как для констант, так и для переменных. Пример:

```
X := Y == Z
```

В этом примере сравнивается значение Y со значением Z, при этом X устанавливается: в TRUE (-1), если в Y такое же значение, как и в Z, либо FALSE (0), если значения различны.

Этот оператор часто используется в условных выражениях, как в следующем примере.

```
IF (Y == 1)
```

Здесь логический оператор Равенства возвратит **TRUE, если** Y равен 1.

Логический оператор Равенства имеет форму с присваиванием, ===, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X === Y      'Short form of X := X == Y
```

Здесь X сравнивается с Y, и если их значения равны, X устанавливается в TRUE (-1), иначе X устанавливается в FALSE (0). Присваиваемая форма оператора Равенство может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Логическое Не Равно '<>', '<>='

Логический оператор Не Равно сравнивает два операнда и возвращает TRUE (-1), если эти значения не равны, иначе возвращает FALSE (0). Оператор Не Равно может использоваться как для констант, так и для переменных. Пример:

```
X := Y <> Z
```

В этом примере сравнивается значение Y со значением Z, при этом X устанавливается: в FALSE (0), если в Y такое же значение, как и в Z, либо TRUE (-1) если значения различны.

Этот оператор часто используется в условных выражениях, как в следующем примере.

```
IF (Y <> 25)
```

Здесь логический оператор Не Равно возвратит TRUE, если Y не равен 25.

Логический оператор Не Равно имеет форму с присваиванием, <>=, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X <>= Y      'Short form of X := X <> Y
```

Здесь X сравнивается с Y, и если их значения не равны, X устанавливается в TRUE (-1), иначе X устанавливается в FALSE (0). Присваиваемая форма оператора Не Равно может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Логическое Меньше '<', '<='

Логический оператор Меньше сравнивает два операнда и возвращает значение TRUE (-1), если первая величина меньше второй, иначе возвращает FALSE (0). Оператор Меньше может использоваться как для констант, так и для переменных. Пример:

```
X := Y < Z
```

В этом примере сравнивается значение Y со значением Z, при этом X устанавливается: в TRUE (-1), если Y меньше Z, либо иначе FALSE (0).

Этот оператор часто используется в условных выражениях, как в следующем примере.

```
IF (Y < 32)
```

Здесь оператор Меньше возвращает TRUE, если Y меньше 32.

Логический оператор Меньше имеет форму с присваиванием, <=, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X <= Y        'Short form of X := X < Y
```

Здесь X сравнивается с Y, и если X меньше Y, X устанавливается в TRUE (-1), иначе X устанавливается в FALSE (0). Присваиваемая форма оператора Меньше может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Логическое Больше '>', '>='

Логический оператор Больше сравнивает два операнда и возвращает значение TRUE (-1), если первая величина больше второй, иначе возвращает FALSE (0). Оператор Больше может использоваться как для констант, так и для переменных. Пример:

```
X := Y > Z
```

В этом примере сравнивается значение Y со значением Z, при этом X устанавливается: в TRUE (-1), если Y больше Z, иначе – в FALSE (0).

Этот оператор часто используется в условных выражениях, как в следующем примере.

```
IF (Y > 50)
```

Здесь оператор Меньше возвращает TRUE, если Y больше 50.

Логический оператор Больше имеет форму с присваиванием, >=, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X >= Y        'Short form of X := X > Y
```

Здесь X сравнивается с Y, и если X больше Y, X устанавливается в TRUE (-1), иначе X устанавливается в FALSE (0). Присваиваемая форма оператора Больше может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Логическое Меньше или Равно '=<', '=<='

Логический оператор Меньше или Равно сравнивает два операнда и возвращает значение TRUE (-1), если первая величина меньше или равна второй, иначе возвращает

FALSE (0). Оператор Меньше или Равно может использоваться как для констант, так и для переменных. Пример:

```
X := Y =< Z
```

В этом примере сравнивается значение Y со значением Z, при этом X устанавливается: в TRUE (-1), если Y меньше или равно Z, иначе в FALSE (0).

Этот оператор часто используется в условных выражениях, как в следующем примере.

```
IF (Y =< 75)
```

Здесь оператор Меньше или Равно возвращает TRUE, если Y меньше либо равно 75.

Логический оператор Меньше или Равно имеет форму с присваиванием, =<=, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X =<= Y      'Short form of X := X =< Y
```

Здесь X сравнивается с Y, и если X меньше или равно Y, X устанавливается в TRUE (-1), иначе X устанавливается в FALSE (0). Присваиваемая форма оператора Меньше или равно может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Логическое Больше или Равно '=>', '=>='

Логический оператор Больше или Равно сравнивает два операнда и возвращает значение TRUE (-1), если первая величина больше или равна второй, иначе возвращает FALSE (0). Оператор Больше или Равно может использоваться как для констант, так и для переменных. Пример:

```
X := Y => Z
```

В этом примере сравнивается значение Y со значением Z, при этом X устанавливается: в TRUE (-1), если Y больше или равно Z, иначе – в FALSE (0).

Этот оператор часто используется в условных выражениях, как в следующем примере.

```
IF (Y => 100)
```

Здесь оператор Больше или Равно возвращает TRUE, если Y больше либо равно 100.

Логический оператор Больше или Равно имеет форму с присваиванием, =>=, которая использует переменную слева от себя и как первый операнд, и как приемник результата. Например,

```
X =>= Y      'Short form of X := X => Y
```

Здесь X сравнивается с Y, и если X больше или равно Y, то X устанавливается в TRUE (-1), иначе X устанавливается в FALSE (0). Присваиваемая форма оператора Больше или Равно может также использоваться и для промежуточных результатов; см. «Промежуточные присваивания», стр. 295.

Адрес идентификатора '@'

Оператор взятия адреса идентификатора возвращает адрес идентификатора, сопровождающего его. Этот оператор может использоваться для целых констант и переменных, но не для float-выражений-констант. Пример:

```
BYTE[@Str] := ''A''
```

В приведенном коде оператор взятия адреса идентификатора возвращает адрес идентификатора Str, который затем используется как индекс массива байтов для сохранения символа ''A'' по этому адресу.

Адрес идентификатора обычно используется для передачи адреса строк и структур данных, определенных в блоке DAT, методам, которые с ними оперируют.

Важно отметить, что это особый оператор, который действует по-разному в выражениях-переменных и выражениях-константах. Во время выполнения, как показано в приведенном выше примере, он возвращает абсолютный адрес сопровождающего его идентификатора. Этот полученный во время выполнения адрес состоит из базового адреса программы объекта плюс смещение идентификатора.

В выражениях-константах он возвращает лишь смещение идентификатора по отношению к объекту. Он не может возвратить абсолютный адрес, имеющий место при выполнении программы, потому как этот адрес изменяется в зависимости от реального адреса объекта во время выполнения. Для правильного использования Адреса идентификатора в константах, таких как таблицы данных, см. оператор «Адрес Объекта Плюс Идентификатора '@@'», описанный далее.

Адрес Объекта Плюс Идентификатор '@@'

Оператор Адреса Объекта Плюс Идентификатор возвращает значение идентификатора, следующего за ним, плюс базовый адрес программы текущего объекта. Адрес Объекта Плюс Идентификатор может использоваться только с выражениями-переменными.

Этот оператор полезен, когда создается таблица из адресов смещений, используемая затем при выполнении программы для определения абсолютных адресов величин, которые они представляют. Например, блок DAT может включать несколько строк, к которым Вам необходимо осуществлять прямой и косвенный доступ:

```
DAT
  Str1 byte "Hello.", 0
  Str2 byte "This is an example", 0
  Str3 byte "of strings in a DAT block.",0
```

Используя @Str1, @Str2, и @Str3, мы можем получить прямой доступ к этим строкам во время выполнения, однако косвенный доступ к ним проблематичен, поскольку каждая строка имеет различную длину, делая сложным использование любой из них как базы для косвенных вычислений адреса.

Проблему можно решить созданием еще одной таблицы самих этих адресов смещений:

```
DAT
  StrAddr word @Str1, @Str2, @Str3
```

Этот код создает таблицу слов, начиная с StrAddr, где каждое слово содержит адрес определенной строки. К сожалению, для констант компилирования (таких, как таблица StrAddr), адрес, возвращенный оператором «@» – это только смещение во время компиляции, а не абсолютный адрес идентификатора во время выполнения. Чтобы получить настоящий адрес идентификатора при выполнении, нам необходим, кроме адреса смещения, еще и базовый адрес программы этого объекта. Это дает оператор Адрес Объекта Плюс Идентификатор. Пример:

```
REPEAT Idx FROM 0 TO 2
  PrintStr(@@StrAddr[Idx])
```

В этом примере Idx инкрементируется от 0 до 2. Запись StrAddr[Idx] получает смещение строки, сохраненной в элементе Idx таблицы StrAddr при компиляции. Оператор «@@», перед записью StrAddr[Idx], добавляет базовый адрес программы объекта к полученному при компиляции смещению, вычисляя корректный адрес строки при выполнении. Метод PrintStr, код которого не показан в примере, может использовать этот адрес для доступа к каждому символу строки.

OUTA, OUTB

Регистр: Выходные регистры 32-битных портов Port A и B.

((PUB ¦ PRI))
OUTA ⟨[*Pin(s)*]⟩

((PUB ¦ PRI))
OUTB ⟨[*Pin(s)*]⟩ (Reserved for future use)

Возвращает: Текущее состояние выводов *Pin(s)* для портов Port A или B, если используется как переменная-источник.

- ***Pin(s)*** – опциональное выражение, либо выражение-диапазон, которое задает линию(линии) B/B, к которой будет производиться доступ в порту Port A (0-31) или Port B (32-63). Если задано как простое выражение, доступ производится лишь к одной указанной линии. Если же задается как выражение-диапазон (два выражения в формате диапазона; x..y), доступ производится к смежным линиям в диапазоне от первого до второго значения.

Описание

OUTA и OUTB - это два из шести регистров (DIRA, DIRB, INA, INB, OUTA и OUTB), которые напрямую влияют на линии B/B. Регистр OUTA содержит состояния каждой из 32 линий B/B порта Port A; биты с 0 по 31 соответствуют пинам от P0 до P31. Регистр OUTB содержит состояния каждой из 32 линий B/B порта Port B; биты с 0 по 31 соответствуют пинам от P32 до P63.

ПРИМЕЧАНИЕ: OUTB зарезервирован для будущего применения; ИМС Propeller P8X32A не содержит линий порта Port B, поэтому далее рассматривается только OUTA.

OUTA используется как для установки, так и для получения текущего состояния одной или более линий B/B порта Port A. Бит с нулевым значением (0) устанавливает на соответствующей линии B/B уровень земли GND. Бит, установленный в единицу (1) устанавливает на соответствующей линии B/B уровень VDD (3.3 Вольт). При запуске процессора, все биты регистра OUTA по умолчанию устанавливаются в 0.

Все линии непосредственно подключены к каждому процессору, поэтому *Hub* никакого влияния на доступ к линиям не оказывает. Каждый *Cog* содержит свой собственный регистр OUTA, который предоставляет ему возможность установить состояние линий (в высокий либо низкий уровень) в любой момент времени. Состояния выходных регистров всех процессоров складываются по ИЛИ, и этот 32-

битный результат используется для установки состояний линий B/B порта Port A с P0 по P31. В результате получается, что состояние выхода каждой из линий B/B порта представляет собой “монтажное ИЛИ” всего состава процессоров. См. «Линии B/B» на стр. 30 для более детальной информации.

Отметьте, что состояния выходов каждого процессора в свою очередь получены путем сложения по ИЛИ состояний линий внутренних аппаратных блоков (Выходного Регистра, Видео-Генератора и т.д.), после чего они умножены по И на значение их Регистра Направления.

На каждой линии B/B устанавливается заданный соответствующими выходами процессора уровень только тогда, когда соответствующий ей бит Регистра Направления (DIRA) этого же процессора равен единице (1). Иначе этот *Cog* задает данную линию как вход и установленное ей выходное состояние игнорируется.

Эта конфигурация может быть легко описана такими простыми правилами:

A. На линии выводится низкий уровень только тогда, когда все активные процессоры, установившие линию как выход, установили ее состояние в ноль.
B. На линии выводится высокий уровень, если любой из активных процессоров, установивших ее выходом, установит ее состояние в единицу (1).

Если *Cog* отключен, состояние его Регистра Направления рассматривается как установленного в ноль, т.е он никак не влияет на состояние и направление линий B/B.

Заметьте, что из-за природы реализации линий B/B в виде “монтажного ИЛИ”, электрическое соединение между процессорами отсутствует, хотя они все же могут осуществлять одновременный доступ к линиям B/B. На разработчика ложится задача убедиться, что ни одни два из процессоров не приводят к коллизиям на одной и той же линии B/B в процессе выполнения приложения.

Использование OUTA

Для задания необходимого состояния выходов соответствующих линий B/B необходимо установите либо сбросить биты регистра OUTA. Кроме того, необходимо убедитесь в правильной установке битов регистра DIRA для задания направления этих линий на вывод. Например:

```
DIRA := %00000100_00110000_00000001_11110000
OUTA := %01000100_00110000_00000001_10010000
```

Здесь строка с DIRA устанавливает линии B/B 26, 21, 20, 8, 7, 6, 5 и 4 как выходы, а остальные – как входы. Строка с OUTA устанавливает на линиях B/B 30, 26, 21, 20, 8, 7, и

4 высокий, а на остальных – низкий уровень. В результате пины 26, 21, 20, 8, 7, и 4 выводят высокий уровень, а пины 6 и 5 – низкий. Направление линии B/B 30 установлено на ввод (согласно DIRA), поэтому установленный в единицу бит 30 регистра OUTA игнорируется и этот пин остается входом для данного процессора.

Для управления отдельными пинами в каждый момент времени можно использовать опциональное поле *Pin(s)* и унарные операторы пост-очистки (**~**) и пост-установки (**~~**). При этом в поле *Pin(s)* регистр линий B/B рассматривается как массив из 32 бит. Например:

```
DIRA[10]~~ 'Set P10 to output
OUTA[10]~ 'Make P10 low
OUTA[10]~~ 'Make P10 high
```

Первая строка приведенного кода устанавливает линию B/B на вывод. Вторая строка сбрасывает бит защелки выхода линии P10, устанавливая выход P10 в ноль (GND). Третья строка устанавливает бит защелки выхода P10, устанавливая выход P10 в единицу (VDD).

В языке *Spin* регистр OUTA поддерживает специальную форму выражения, называемую выражение-диапазон, которое позволяет получить доступ к нескольким линиям за один раз, не влияя на другие линии вне указанного диапазона. Для одновременного влияния сразу на несколько смежных линий B/B, используйте выражение-диапазон (x..y) в поле *Pin(s)*.

```
DIRA[12..8]~~ 'Set DIRA12:8 (P12-P8 to output)
OUTA[12..8] := %11001 'Set P12:8 to 1, 1, 0, 0, and 1
```

В первой строке, "DIRA…," линии P12, P11, P10, P9 и P8 устанавливаются на вывод; все остальные пины остаются в прежнем состоянии. Во второй строке, "OUTA…," состояние линий P12, P11, и P8 устанавливается в высокий уровень, а P10 и P9 – в низкий.

ВАЖНО: Порядок указания величин в выражении-диапазоне соответствующим образом оказывает влияние и на линии. Например, в следующем коде изменен порядок следования величин в выражении-диапазоне из предыдущего примера.

```
DIRA[8..12]~~ 'Set DIRA8:12 (P8-P12 to output)
OUTA[8..12] := %11001 'Set OUTA8:12 to 1, 1, 0, 0, and 1
```

Здесь биты с 8 по 12 регистра DIRA установлены в 1 (как и ранее), однако биты 8, 9, 10, 11 и 12 регистра OUTA установлены равными соответственно 1, 1, 0, 0, и 1, устанавливая P8, P9 и P12 в высокий, а P10 и P11 – в низкий уровень.

Это мощное свойство выражений-диапазонов, однако если ему не уделить должного внимания, оно может привести к непредвиденным результатам.

Обычно регистр OUTA используется для записи, однако он также может быть прочитан для получения текущего состояния битов защелок линий В/В. Это будут лишь состояния битов защелок Выходного Регистра данного процессора, и не обязательно реальные состояния выходов на пинах ИМС Propeller, поскольку они могут быть далее изменены другими процессорами или даже другими аппаратными блоками В/В этого же процессора (Видео-Генератором, Счетчиком и т.д.). В следующем примере считается, что переменная Temp уже создана ранее, в другом месте программы:

```
Temp := OUTA[15..13] 'Get output latch state of P15 to P13
```

Здесь значение переменной Temp устанавливается равным битам 15, 14, и 13 регистра OUTA; т.е. младшие три бита переменной Temp сейчас равны OUTA15:13, а остальные биты Temp сброшены в ноль.

PAR

Регистр: Регистр Параметра Загрузки процессора .

((PUB ¦ PRI))
 PAR

Возвращает: Значение адреса, переданное во время запуска ассемблерной программы при помощи COGINIT или COGNEW.

Описание

Регистр PAR содержит значение адреса, переданное в поле *Parameter* команды COGINIT или COGNEW; см. COGINIT, стр. 218 и COGNEW, стр. 221. Содержимое регистра PAR используется программой на языке Propeller-ассемблер для нахождения и работы с памятью, общей для программ на языках *Spin* и ассемблер.

Поскольку регистр PAR предназначен для хранения адреса при запуске процессора, значение, сохраняемое в нем командами COGINIT и COGNEW ограничивается до 14 бит, т.е. это – 16-битное слово, с младшими двумя битами, сброшенными в ноль.

Использование регистра PAR

Значение регистра PAR задается в коде *Spin* и используется в коде ассемблера как механизм организации указателя на общую между ними область основной памяти. При запуске кода ассемблера на выполнение в процессоре, регистр PAR задается либо командой COGINIT, либо COGNEW. Например:

```
VAR
  long Shared                           'Shared variable (Spin & Assy)
PUB Main | Temp
  Cognew(@Process, @Shared)             'Launch assy, pass Shared addr
  repeat
    <do something with Shared vars>
DAT
                org 0
  Process       mov Mem, PAR            'Retrieve shared memory addr
  :loop         <do something>
                wrlong ValReg, Mem      'Move ValReg value to Shared
                jmp #:loop
Mem res 1
ValReg res 1
```

В этом примере метод Main запускает, при помощи **COGNEW,** в новом процессоре ассемблерную подпрограмму Process. Второй параметр **COGNEW** используется методом Main для передачи адреса переменной Shared. Ассемблерная подпрограмма Process получает значение этого адреса из его регистра **PAR** и сохраняет в локальной переменной Mem. Затем она выполняет какую-либо задачу, обновляя свой локальный регистр ValReg (созданный в конце блока **DAT**), и в конце обновляет переменную Shared при помощи `wrlong ValReg, Mem`.

В Propeller-ассемблере регистр PAR является регистром только для чтения, поэтому он может быть использован только в качестве переменной-источника значения (s-поля) (например: **mov** *dest*, *source*).

PHSA, PHSB

Регистр: Регистры ФАПЧ (PLL) счетчиков Counter A и Counter B.

((PUB ¦ PRI))
PHSA

((PUB ¦ PRI))
PHSB

Возвращает: Текущее значение регистров ФАПЧ счетчика Counter A или Counter B, если используется как переменная-источник.

Описание

Регистры PHSA и PHSB – это два из шести регистров (CTRA, CTRB, FRQA, FRQB, PHSA, и PHSB), которые влияют на режим работы Модулей Счетчиков процессора. Каждый *Cog* имеет два идентичных модуля счетчика (A и B), которые могут выполнять множество повторяющихся задач. Регистры PHSA и PHSB содержат значения, которые могут быть напрямую прочитаны либо записаны процессором, а также могут накапливаться на каждом цикле Системной Частоты, инкрементируясь значениями из соответственно FRQA и FRQB. Для более детальной информации см. CTRA на стр. 239.

Использование PHSA и PHSB

Регистры PHSA и PHSB могут быть прочитаны/записаны, как и другие регистры или предопределенные переменные. Например:

```
PHSA := $1FFFFFFF
```

Приведенный код устанавливает PHSA в $1FFFFFFF. В зависимости от поля CTRMODE регистра CTRA, это значение может оставаться неизменным, либо может автоматически инкрементироваться значением из FRQA на частоте, определяемой Системной Частотой, а также основной и/или дополнительной линией B/B. Для более детальной информации см. CTRA, CTRB на стр. 239

Имейте в виду, что в Propeller-ассемблере регистры PHSA и PHSB не могут использоваться в операциях чтение-модификация-запись в поле «приемник» инструкции. Должны выполняться отдельные операции чтения, модификации и записи.

Помните, что прямая запись в регистры PHSA или PHSB переписывает как текущее накопленное значение, так и любое аккумулирование, которое могло быть запланировано на этот момент.

PRI

Объявление: Объявляет блок метода *Private*.

((PUB ¦ PRI))

PRI ***Name*** ⟨ (***Param*** ⟨ , ***Param***⟩...) ⟩ ⟨ : ***RValue***⟩ ⟨ | ***LocalVar*** ⟨ [***Count***] ⟩⟩ ⟨ , ***LocalVar*** ⟨ [***Count***] ⟩⟩...
SourceCodeStatements

- ***Name*** – желаемое имя метода *Private*.
- ***Param*** – имя параметра (опционально). Методы могут содержать ноль и более разделенных запятыми параметров, заключенных в скобки. *Param* должно быть глобально уникальным, однако другие методы могут использовать такое же имя для идентификаторов. Каждый параметр является переменной *long*.
- ***RValue*** – имя возвращаемого методом значения (опционально). Становится копией встроенной переменной RESULT. *RValue* – глобально уникальное, однако другие методы могут использовать такое же имя для идентификаторов. *RValue* (и/или RESULT) при вызове метода инициализируются в 0.
- ***LocalVar*** – имя локальной переменной (опционально). *LocalVar* должно быть глобально уникальным, однако другие методы могут использовать такое же имя для идентификаторов. Все локальные переменные имеют размер *long* и остаются неинициализированными при вызове метода. Методы могут содержать ноль и более разделенных запятыми локальных переменных.
- ***Count*** – опциональное выражение, в квадратных скобках, которое указывает на то, что это – локальная переменная-массив с количеством *long*-элементов *Count*. При обращении к ним, они начинаются с элемента 0 по элемент *Count*-1.
- ***SourceCodeStatements*** – одна или более строк исполнимого исходного кода, осуществляющего функцию метода, с отступом как минимум в один пробел.

Описание

Идентификатор PRI служит для объявления блока метода *Private*. Метод *private* – это секция кода, выполняющая определенную функцию и возвращающая значение результата. Это одно из шести специальных объявлений (CON, VAR, OBJ, PUB, PRI, и DAT), обеспечивающих четкую структуру языка *Spin*.

Каждый объект может включать несколько *Private*- (PRI) и *Public*- (PUB) методов. *Private* методы могут быть доступны лишь внутри объекта и служат для выполнения важных, закрытых функций объекта. *Private*-методы во всем похожи на *Public*-методы за исключением того, что они объявлены как PRI, и не доступны извне объекта. Для дополнительной информации см. PUB, стр. 334.

PUB

Объявление: Объявляет блок метода *Public*.

((PUB ¦ PRI))

PUB ***Name*** ⟨ (***Param*** ⟨ , ***Param***⟩...) ⟩ ⟨ : ***RValue***⟩ ⟨ | ***LocalVar*** ⟨ [***Count***] ⟩⟩ ⟨ , ***LocalVar*** ⟨ [***Count***] ⟩⟩...
SourceCodeStatements

- ***Name*** желаемое имя метода *Public*.
- ***Param*** – имя параметра (опционально). Методы могут содержать ноль и более разделенных запятыми параметров, заключенных в скобки. *Param* должно быть глобально уникальным, однако другие методы могут использовать такое же имя для идентификаторов. Каждый параметр является переменной *long*.
- ***RValue*** – имя возвращаемого методом значения (опционально). Становится копией встроенной переменной RESULT. *RValue* – глобально уникальное, однако другие методы могут использовать такое же имя для идентификаторов. *RValue* (и/или RESULT) при вызове метода инициализируется в 0.
- ***LocalVar*** – имя локальной переменной (опционально). *LocalVar* должно быть глобально уникальным, однако другие методы могут использовать такое же имя для идентификаторов. Все локальные переменные имеют размер *long* (четыре байта) и остаются неинициализированными при вызове метода. Методы могут содержать ноль и более разделенных запятыми локальных переменных.
- ***Count*** – опциональное выражение, в квадратных скобках, которое указывает на то, что это – локальная переменная-массив с количеством *long*-элементов *Count*. При обращении к ним, они начинаются с элемента 0 по элемент *Count*-1.
- ***SourceCodeStatements*** – одна или более строк исполнимого исходного кода, осуществляющего функцию метода, с отступом как минимум в один пробел.

Описание

Идентификатор PUB служит для объявления блока метода *Public*. Метод public – это секция кода, выполняющая определенную функцию и возвращающая значение результата. Это одно из шести специальных объявлений (CON, VAR, OBJ, PUB, PRI, и DAT), обеспечивающих четкую структуру языка *Spin*.

Каждый объект может включать несколько *Public*- (PUB) и *Private*- (PRI)методов. *Public* методы могут быть доступны извне объекта и обеспечивают интерфейс с объектом.

Сами объявления PUB и PRI не возвращают значений, но методы *Public* и *Private*, которые они представляют, всегда возвращают, будучи вызваны где-либо в программе.

Объявление метода *Public*

Объявление метода *Public* начинается с **PUB**, в первой колонке строки, и сопровождается уникальным именем и опциональным набором параметров, переменной результата и локальных переменных.

Пример:

```
PUB Init
  <initialization code>

PUB MotorPos : Position
  Position := <code to retrieve motor position>

PUB MoveMotor(Position, Speed) : Success | PosIndex
  <code that moves motor to Position at Speed and returns True/False>
```

В этом примере содержится три метода *Public*: `Init`, `MotorPos` и `MoveMotor`. Метод `Init` не имеет параметров и не объявляет ни возвращаемого значения, ни локальных переменных. Метод `MotorPos` не имеет параметров, однако объявляет значение возврата с именем `Position`. Метод `MoveMotor` имеет два параметра, `Position` и `Speed`, значение возврата, `Success`, и локальную переменную `PosIndex`.

Все исполнимые выражения, принадлежащие методу **PUB и** вводимые после его объявления, должны иметь отступ как минимум в один пробел.

Величина возврата

Независимо от того, задано ли в объявлении **PUB** значение возврата *RValue*, всегда существует встроенная величина возврата, которая по умолчанию равна нолю (0). В каждом методе **PUB** существует предопределенное имя этой переменной – **RESULT**. В любой момент времени в рамках метода переменная **RESULT** может быть обновлена как и любая другая переменная, и, при выходе из метода, текущее значение **RESULT** передается вызвавшему методу. Вдобавок, если для метода объявлена своя переменная **RESULT**, это имя может быть использовано попеременно со встроенной переменной **RESULT**. В частности, метод `MotorPos`, приведенный выше, устанавливает "`Position :=` …", а мог с тем же успехом использовать "`Result :=` …". Несмотря на это, считается хорошей практикой давать значению возврата описательное имя (в объявлении `PUB`) для каждого метода, значение возврата которого существенно. Аналогично, считается правильной практикой оставлять значение возврата не объявленным (в объявлении **PUB**) для любого метода, значение возврата которого не существенно и не используется.

Параметры и локальные переменные

Параметры и локальные переменные имеют размер *long* (четыре байта). На самом деле параметры – это просто переменные, инициализированные в определенные значения, заданные вызывающим методом. Локальные переменные, однако, не инициализируются, и когда бы не был вызван метод, они содержат случайные данные.

Все параметры передаются в метод явно, значениями, а не ссылкой, поэтому любые изменения в самих параметрах никак не отражаются вне метода. Например, если мы вызвали `MoveMotor` с использованием переменной `Pos` как первого параметра, это будет выглядеть так:

```
Pos := 250
MoveMotor(Pos, 100)
```

Когда выполняется метод `MoveMotor`, он получает значение `Pos` в параметре `Position`, и значение 100 в параметре `Speed`. Внутри метода `MoveMotor`, значения `Position` и `Speed` могут меняться в любой момент, однако значение `Pos` (переменная вызывающего метода), остается равным 250.

Если же переменная должна быть изменена в подпрограмме, вызывающий метод должен передать ее косвенно, по ссылке; это значит, он должен передать адрес переменной вместо самого ее значения, и подпрограмма должна рассматривать этот параметр как адрес ячейки памяти, с которым нужно работать. Адрес переменной, или любого другого регистрового идентификатора, может быть получен с использованием оператора «Адрес идентификатора», '`@`'. Например,

```
Pos := 250
MoveMotor(@Pos, 100)
```

Вызывающий метод передает адрес `Pos` как первый параметр в метод `MoveMotor`. Что получает `MoveMotor` в своем параметре `Position`, так это адрес переменной `Pos` вызывающего метода. Адрес – это просто число, как и любое другое, поэтому метод `MoveMotor` должен быть написан таким образом, чтобы рассматривать это число как адрес, а не как величину. Метод `MoveMotor` теперь должен использовать конструкцию:

```
PosIndex := LONG[Position]
```
...для получения величины переменной `Pos` вызывающего метода, и конструкцию:

```
LONG[Position] := <some expression>
```
...для изменения этой переменной, при необходимости.

Передача величины по ссылке с использованием оператора Адреса Идентификатора обычно используется, когда необходимо передать в метод строковую переменную. Поскольку строковые переменные – это на самом деле просто массивы байтов, не существует способа передачи их в метод прямо, по значению; это приводит к тому, что метод получает только первый символ строки. Даже если в методе не нужно изменять строку либо другой логически организованный массив, этот массив все равно должен быть передан ссылкой, потому как в нем много элементов, к которым необходимо производить доступ.

Оптимизированная адресация

В откомпилированом Propeller-приложении первые восемь (8) *long*-ов, которые образуют параметры, переменную RESULT и локальные переменные, адресуются по оптимизированной схеме. Это значит, что доступ к этим первым восьми двойным словам (параметрам, RESULT, и локальным переменным) требует немного меньше времени, чем доступ к девятому и далее, *long*-ам. Для оптимизации скорости выполнения кода необходимо убедиться в том, что все локальные переменные, используемые в наиболее часто повторяющихся местах метода, находятся в числе этих восьми. Подобный механизм применим также и для глобальных переменных; см. секцию VAR, стр. 364, для подробной информации.

Выход из метода

Выход из метода производится при достижении его последнего выражения либо при выполнении одной из команд: RETURN или ABORT. В методе может быть одна точка выхода (последняя выполненная запись), либо их может быть множество (любое количество команд RETURN или ABORT вдобавок к последней выполнимой записи). При выходе по командам RETURN и ABORT есть возможность задания значения переменной RESULT; см. RETURN, стр. 349, и ABORT, стр. 187.

QUIT

Команда: Выход из цикла REPEAT.

((PUB □ PRI))
 QUIT

Описание

QUIT - это одна из двух команд (NEXT и QUIT), которые влияют на выполнение циклов REPEAT. Команда QUIT приводит к немедленному выходу из цикла REPEAT.

Использование QUIT

QUIT обычно используется для преждевременного выхода из цикла REPEAT при организации обслуживания исключительной ситуации в условных выражениях. Например, допустим, что DoMore и SystemOkay – это методы, созданные ранее и возвращающие логические значения:

```
repeat while DoMore          'Repeat while more to do
  !outa[0]                   'Toggle status light
  <do something>             'Perform some task
  if !SystemOkay
    quit                     'If system failure, exit
  <more code here>           'Perform other tasks
```

В приведенном выше коде производится переключение индикатора статуса на линии P0 и выполнение других задач, до тех пор, пока методом DoMore возвращается значение TRUE. Однако в случае, если метод SystemOkay в середине цикла возвратит значение FALSE, то условие IF выполнит команду QUIT, которая приведет к немедленному выходу из цикла.

Команда QUIT может использоваться только внутри цикла REPEAT; в другом случае возникнет ошибка.

REBOOT

Команда: Сброс ИМС Propeller.

((PUB ¦ PRI))
  REBOOT

Описание

Эта команда выполняет программный сброс, но эффект от её выполнения такой же, как и от аппаратного сброса на линии RES.

Команда REBOOT используется при необходимости инициализации ИМС Propeller в ее начальное состояние, аналогичное таковому по включению питания. При этом имеют место все те же аппаратные задержки и процесс начальной загрузки, что и при выполнении аппаратного сброса на линии RESn либо выключения/включения питания.

REPEAT

Команда: Циклически выполнить блок кода.

((PUB ¦ PRI))
  REPEAT ⟨*Count*⟩
  →| *Statement(s)*

((PUB ¦ PRI))
  REPEAT *Variable* FROM *Start* TO *Finish* ⟨STEP *Delta*⟩
  →| *Statement(s)*

((PUB ¦ PRI))
  REPEAT ((UNTIL ¦ WHILE)) *Condition(s)*
  →| *Statement(s)*

((PUB ¦ PRI))
  REPEAT
  →| *Statement(s)*
  ((UNTIL ¦ WHILE)) *Condition(s)*

- ***Count*** – опциональное выражение, указывающее определенное количество повторений выполнения блока *Statement(s)*. Если *Count* не указывается, синтаксис 1 организует бесконечный цикл выполнения кода *Statement(s)*.
- ***Statement(s)*** – опциональный блок из одного или более строк кода для циклического выполнения. Пропуск *Statement(s)* используется редко, но может быть полезен в синтаксисах 3 и 4 если *Condition(s)* достигает нужных значений.
- ***Variable*** – переменная, обычно определяемая пользователем, которая будет изменяться от значения *Start* до *Finish*, опционально с величиной *Delta* на каждое повторение. *Variable* может использоваться в блоке *Statement(s)* для использования текущего значения счетчика циклов.
- ***Start*** – выражение, определяющее начальное значение переменной *Variable* в синтаксисе 2. Если *Start* меньше, чем *Finish*, то переменная *Variable* будет инкрементироваться с каждым циклом; иначе она будет декрементироваться.
- ***Finish*** – выражение, определяющее конечное значение переменной *Variable* в синтаксисе 2. Если *Finish* больше, чем *Start,* то переменная *Variable* будет инкрементироваться с каждым циклом; иначе она будет декрементироваться.
- ***Delta*** – опциональное выражение, определяющее количество единиц, на которое будет инкрементироваться/декрементироваться переменная *Variable* при каждом повторении (синтаксис 2). Если оно не используется, переменная *Variable* инкрементируется/декрементируется при каждом повторении на 1.

- ***Condition(s)*** – одно или более логических выражений, используемых в синтаксисе 3 и 4 в качестве условий продолжения либо прекращения выполнения цикла. Когда они предваряются UNTIL, цикл завершается при логическом значении *Condition(s)* TRUE. Когда имеем цикл WHILE, он завершается при *Conditions(s)*, дающих в результате FALSE.

Описание

REPEAT – это очень гибкая структура для организации циклов в языке *Spin*. Она может использоваться для организации циклов любого типа, включая: бесконечный, конечный, с/без счетчиком циклов, а также условный циклы ноль-множество/один-множество.

Отступы важны!

ВАЖНО: Отступы важны! Язык *Spin* чувствителен к отступам (на один либо более пробел) в строках, сопровождающих команды условного выполнения для определения, принадлежат ли они блоку данной команды, или нет. Чтобы указать программе *Propeller Tool* индицировать такие логически сгруппированные блоки кода на экране, Вы можете нажать *Ctrl+I* для включения индикаторов блок-групп. Повторное нажатие *Ctrl+I* отключит эту функцию. См. «Отступы и Выступы», стр. 84, и «Индикаторы Блок-Групп», стр. 89.

Бесконечные циклы (Синтаксис 1)

На самом деле любая форма REPEAT может быть приведена к бесконечному циклу, однако формой, наиболее часто используемой в этих целях, является синтаксис 1 без поля *Count*. Например:

```
repeat                      'Repeat endlessly
  !outa[25]                 'Toggle P25
  waitcnt(2_000 + cnt)      'Pause for 2,000 cycles
```

В этом коде выполняется бесконечное повторение строк `!outa[25]` и `waitcnt(2_000 + cnt)`. Обе строки введены с отступом от **REPEAT,** поэтому они обе принадлежат циклу.

Поскольку *Statement(s)* – это опциональная часть REPEAT, сама команда REPEAT может использоваться в качестве бесконечного цикла, который ничего не выполняет, но сохраняет *Cog* активным. Такой цикл можно организовать преднамеренно, но иногда такое может случиться непреднамеренно, из-за неправильного выполнения отступов. Например:

```
repeat                      'Repeat endlessly
!outa[25]                   'Toggle P25 <-- This is never run
```

В этот пример закралась ошибка: последняя строка никогда не выполнится, поскольку перед ней находится бесконечный цикл **REPEAT**, который не имеет операторов *Statement(s)*; после него нет операторов с отступом, поэтому *Cog* просто находится в бесконечном цикле на строке **REPEAT**, которая ничего не выполняет, однако держит *Cog* активным и потребляющим энергию.

Простые Конечные Циклы (Синтаксис 1)

Большинство циклов по своей природе конечны; они выполняют только ограниченное количество повторений. Самая простая форма – это синтаксис 1 с полем *Count*.

Например:

```
repeat 10                   'Repeat 10 times
  !outa[25]                 'Toggle P25
byte[$7000]++               'Increment RAM location $7000
```

Приведенный код переключает линию P25 десять раз, затем инкрементирует значение в ОЗУ по адресу $7000.

Учтите, что в поле *Count* может находиться любое числовое выражение, но оно вычисляется только один раз, при первом входе в цикл. Это значит, что любые изменения в переменных, входящих в это выражение, ни как не повлияют на количество проходов цикла. В следующем примере предполагается, что переменная `Index` была определена ранее.

```
Index := 10                 'Set loop to repeat 10 times
repeat Index                'Repeat Index times
  !outa[25]                 'Toggle P25
  Index := 20               'Change Index to 20
```

В приведенном примере, переменная `Index` при первом входе в цикл **REPEAT** равна 10. Каждый раз при проходе цикла `Index` устанавливается в 20, однако цикл все равно повторяется лишь 10 раз.

Конечные циклы с подсчетом (Синтаксис 2)

Довольно часто необходимо подсчитывать количество повторений цикла таким образом, чтобы код выполнялся по разному, в зависимости от этого количества. Команда **REPEAT** позволяет с легкостью организовать такой цикл при использовании синтаксиса 2. В следующем примере считается, что `Index` была создана ранее.

```
repeat Index from 0 to 9     'Repeat 10 times
  byte[$7000][Index]++       'Increment RAM locations $7000 to $7009
```

Как и в предыдущем примере, приведенный код повторяется в цикле 10 раз, но каждый раз он изменяет переменную Index. В первый проход цикла, значение Index будет равно 0 (как указано в "from 0"), а каждый следующий проход Index будет на 1 больше, чем в предыдущий раз (как указано в "to 9"): ..1, 2, 3…9. После десятого повтора, Index инкрементируется в 10 и цикл будет завершен, приводя к выполнению следующих команд, следующих за структурой REPEAT, если таковые присутствуют. Код в цикле использует Index как смещение для изменения содержимого памяти, byte[$7000][Index]++; в этом случае это поочередное увеличение каждого байтового значения на 1, в ячейках ОЗУ с адресами от $7000 до $7009.

Команда REPEAT автоматически определяет, является ли направление диапазона, указанного в *Start* и *Finish,* на увеличение, либо на уменьшение. Поскольку в предыдущем примере использовался диапазон от 0 до 9, направление было на увеличение; Index изменялся каждый раз на +1. Чтобы заставить счетчик уменьшаться, нужно просто поменять величины *Start* и *Finish* местами, как в следующем примере:

```
repeat Index from 9 to 0     'Repeat 10 times
  byte[$7000][Index]++       'Increment RAM $7009 down through $7000
```

В этом примере также происходит повторение 10 раз, но со счетчиком Index от 9 до 0; Index изменяется каждый раз на -1. Тело цикла все также инкрементирует значения в ОЗУ, но теперь по адресам от $7009 до $7000. После десятого повтора Index равен -1.

Sпоскольку в полях *Start* и *Finish* могут стоять выражения, то они могут содержать переменные. В следующем примере полагается, что переменные S и F созданы ранее.

```
S := 0
F := 9
repeat 2                     'Repeat twice
  repeat Index from S to F   'Repeat 10 times
    byte[$7000][Index]++     'Increment RAM locations 7000..$7009
  S := 9
  F := 0
```

В этом примере используется вложенный цикл. Внешний цикл (первый) повторяется два раза. Внутренний цикл повторяется со счетчиком Index от S до F, которые были ранее установлены в соответственно 0 и 9. Внутренний цикл инкрементирует значения в ОЗУ по адресам в порядке с $7000 по $7009, поскольку счетчик в этом цикле

увеличивается с 0 до 9. Затем внутренний цикл завершается (с Index, равным 10), и последние две строки устанавливают S в 9, а F – в 0, получая эффект перемены значений *Start* и *Finish* местами. Поскольку все это находится внутри внешнего цикла, этот цикл затем выполняет свое тело снова (второй раз), заставляя внутренний цикл выполняться со счетчиком Index от 9 до 0. Внутренний цикл инкрементирует значения в ОЗУ по адресам в порядке с $7009 по $7000, (обратный порядок по сравнению с предыдущим разом) и завершается при значении Index, равном -1. Последние две строки устанавливают S и F снова, но внешний цикл в третий раз не повторяется.

Циклы REPEAT не ограничиваются величиной приращения или уменьшения только в 1. Если в команде REPEAT используется опциональный синтаксис STEP *Delta*, она будет инкрементировать либо декрементировать переменную *Variable* на величину *Delta*. В синтаксисе 2, REPEAT на самом деле всегда использует величину *Delta*, но когда компонент "STEP *Delta*" опускается, она использует по умолчанию либо +1, либо -1, в зависимости от значений *Start* и *Finish*. В следующем примере используется *Delta* = 2.

```
repeat Index from 0 to 8 step 2  'Repeat 5 times
  byte[$7000][Index]++           'Increment even RAM $7000 to $7008
```

Здесь цикл REPEAT повторяется пять раз, со значением Index соответственно 0, 2, 4, 6, 8. Этот код инкрементирует каждую следующую ячейку ОЗУ (по четному адресу) с $7000 по $7008 и завершается, когда Index равен 10.

Величина *Delta* может быть положительной либо отрицательной, в зависимости от направления в диапазоне, заданном значениями *Start* и *Finish*, и даже может быть изменена внутри цикла для получения интересных эффектов. Например, полагая, что переменные Index и D определены ранее, следующий код устанавливает Index в следующей последовательности: 5, 6, 6, 5, 3.

```
D := 2
repeat Index from 5 to 10 step D
  --D
```

Этот цикл начал выполняться со значения Index, равного 5 и *Delta* (D) равного +2. Но при каждом повторе цикла D декрементируется на единицу, поэтому в конце первого прохода Index = 5 и D = +1. Проход 2 даст Index = 6 и D = 0. Проход 3 даст Index = 6 и D = -1. Проход 4 даст Index = 5 и D = -2. Проход 5 даст Index = 3 и D = -3. Затем цикл завершается, т.к. Index плюс *Delta* (3 + -3) находится вне *Start* … *Finish* (от 5 до 10).

Условные циклы (Синтаксисы 3 и 4)

Последние две формы REPEAT, синтаксис 3 и 4, - это конечные циклы с условными выходами, которые имеют гибкие опции, позволяя использовать положительную либо

отрицательную логику и создание циклов типа ноль-множество или один-множество. Эти две формы **REPEAT** обычно называются циклами "*repeat while*" или "*repeat until*".

Давайте рассмотрим формат цикла **REPEAT**, описанный в синтаксисе 3. Он состоит из команды **REPEAT,** сопровождаемой оператором **WHILE** или **UNTIL,** затем *Condition(s)* и в конце, в расположенных ниже строках, опциональные операторы *Statement(s)*. Поскольку этот формат проверяет *Condition(s)* в начале каждого прохода, он создает цикл типа ноль-множество; блок из операторов *Statement(s)* будет выполняться ноль или более раз, в зависимости от *Condition(s)*. Например, пусть, X создана ранее:

```
X := 0
repeat while X < 10           'Repeat while X is less than 10
  byte[$7000][X] := 0         'Increment RAM value
  X++ 'Increment X
```

В этом примере сначала X устанавливается в 0, затем цикл повторяется до тех пор, пока X меньше, чем 10. Код внутри цикла очищает ячейки ОЗУ по смещению X (начиная с адреса $7000) и инкрементирует X. После 10-го прохода цикла, X становится равен 10, делая результат условия `while X < 10` равным **FALSE**, и цикл прекращается.

Говорят, что в этом цикле использована "положительная" логика, т.к.он продолжается, **WHILE** («пока») условие дает истину (**TRUE**). Он также может быть написан с использованием "отрицательной" логики, используя слово **UNTIL**, («пока не»). Пример:

```
X := 0
  repeat until X > 9          'Repeat until X is greater than 9
    byte[$7000][X] := 0       'Increment RAM value
    X++                       'Increment X
```

Этот пример работает аналогично предыдущему, но в нем цикл **REPEAT** использует отрицательную логику, потому как сопровождается ключевым словом "**UNTIL**". Этот цикл выполняется до тех пор, пока условие дает результат **FALSE**.

В любом из приведенных примеров, в случае, если перед первым проходом цикла **REPEAT,** X был больше или равен 10, условие вообще не допустит ни единого его прохода; поэтому такой цикл и называется циклом «ноль-множество».

Формат **REPEAT,** синтаксис 4, очень похож на синтаксис 3, но условие проверяется в конце каждого прохода, создавая структуру цикла типа «один-множество». Например:

```
X := 0
repeat
  byte[$7000][X] := 0         'Increment RAM value
```

```
    X++ 'Increment X
  while X < 10                'Repeat while X is less than 10
```

Этот пример, делая 10 проходов, работает так же, как и два предыдущие, за исключением того, что условие не проверяется до конца каждого из проходов. Однако, в отличие от предыдущих примеров, даже если X был больше или равен 10 перед самым первым проходом, цикл все равно выполнится один раз перед своим завершением; поэтому такой цикл и называется циклом типа «один-множество».

Другие опции REPEAT

Существуют еще две команды, которые влияют на работу циклов REPEAT: NEXT и QUIT. См. команды NEXT (стр. 287) и QUIT (стр. 338) для более детальной информации.

RESULT

Переменная: Переменная возвращаемого значения для методов.

((PUB ¦ PRI))
  RESULT

Описание

Переменная RESULT - это предопределенная локальная переменная, присутствующая в каждом PUB- и PRI-методе. RESULT содержит возвращаемое методом значение – значение, возвращаемое вызывающему по завершению метода.

При вызове *Public*- или *Private*- метода, его встроенная переменная RESULT обнуляется. Если этот метод не изменяет RESULT, или не вызывает RETURN либо ABORT с заданным значением, то по завершению метода будет возвращено нулевое значение.

Использование RESULT

В конце приведенного ниже примера метод DoSomething устанавливает RESULT в 100. Метод Main вызывает DoSomething и устанавливает свою локальную переменную Temp равной результату; так что по завершению DoSomething, Temp будет равна 100

```
PUB Main | Temp
  Temp := DoSomething 'Call DoSomething, set Temp to return value

PUB DoSomething
  <do something here>
  result := 100 'Set result to 100
```

Чтобы сделать более понятным, что именно возвращает метод, переменной RESULT можно задать второе имя. Это очень полезно, поскольку делает назначение метода более прозрачным. Например:

```
PUB GetChar : Char
  <do something>
  Char := <retrieved character> 'Set Char (result) to the character
```

Приведенный метод GetChar объявляет Char как второе имя своей встроенной переменной RESULT; см. PUB, стр. 334 или PRI, стр. 333, для более детальной информации. Затем метод GetChar выполняет какие-то действия для получения символа и присваивает его переменной Char. Можно было также использовать “result

:= …” для задания возвращаемого значения, поскольку оба эти варианта устанавливают значение переменной, возвращаемой методом.

И переменная RESULT, и ее двойник, могут быть изменены внутри, до выхода из метода, сколько угодно раз, поскольку они обе воздействуют на RESULT; но при выходе из метода, будет использовано только самое последнее значение RESULT.

RETURN

Команда: Выход из метода PUB/PRI с опциональным возвратом значения *Value*.

((PUB ¦ PRI))
  RETURN ⟨*Value*⟩

Возвращает: Либо текущее значение RESULT, либо заданное значение *Value*.

- ***Value*** – это опциональное выражение, значение которого должно быть возвращено из PUB или PRI метода.

Описание

RETURN - это одна из двух команд (ABORT и RETURN), которые завершают выполнение метода PUB или PRI. Команда RETURN приводит к выходу из PUB или PRI метода с обычным статусом; т.е. она выталкивает из стека вызовов одно значение и возвращается по этому адресу к вызывавшему этот метод, доставляя ему заданное значение.

В конце каждого PUB- или PRI-метода находится встроенная команда RETURN, но для создания нескольких точек выхода, RETURN может быть также введена вручную, в одном или более местах внутри метода.

Когда RETURN введена без опционального поля *Value*, она возвращает текущее значение встроенной в PUB/PRI переменной RESULT. Если же поле *Value* было задано, метод PUB или PRI вернет вместо RESULT заданное значение *Value*.

О стеке вызовов

При вызове методов путем обычной ссылки на них из других методов, должен присутствовать какой-то механизм для сохранения места, в которое необходимо вернуться после завершения вызванного метода. Этот механизм называется “стек”, здесь мы будем использовать термин “стек вызовов”. Он представляет собой просто область памяти, используемую для хранения адресов возврата, величин возврата, параметров и промежуточных результатов. По мере того, как вызываются метод за методом, стек вызовов логически удлиняется. По мере того, как метод за методом завершаются (по командам RETURN или достигая своего конца), стек вызовов становится короче. Это называется соответственно “заталкиванием” в стек и “выталкиванием” из стека.

Команда RETURN выталкивает наиболее свежие данные из стека вызовов для обеспечения возврата к непосредственно вызывающему методу, тому, который вызвал только что завершенный метод.

Использование RETURN

В следующем примере показано два варианта использования RETURN. Допустим, что метод DisplayDivByZeroError определен ранее.

```
PUB Add(Num1, Num2)
  Result := Num1 + Num2         'Add Num1 + Num2
  return

PUB Divide(Dividend, Divisor)
  if Divisor == 0               'Check if Divisor = 0
    DisplayDivByZeroError       'If so, display error
    return 0                    'and return with 0
  return Dividend / Divisor     'Otherwise return quotient
```

Метод Add устанавливает свою встроенную переменную RESULT равной Num1 плюс Num2, затем выполняет RETURN. Команда RETURN заставляет метод Add вернуть значение RESULT вызывающему методу. Отметьте, что этот RETURN на самом деле не был нужен, потому что компилятор *Propeller Tool* сам автоматически добавляет его в конце любого метода.

Метод Divide проверяет значение Divisor. Если Divisor равен нолю, он вызывает метод DisplayDivByZeroError и затем выполняет return 0, который приводит к немедленному выходу из метода с возвратом значения 0. Если, однако, Divisor не был равен нолю, он выполняет return Dividend / Divisor, что приводит к выходу из метода с возвратом результата деления. Это пример, где последний RETURN был использован для выполнения вычислений и возврата результата за один шаг, вместо отдельного вычисления и дальнейшего изменения значения встроенной переменной RESULT.

ROUND

Директива: Округлить float-константу к ближайшему целому.

((CON ¦ VAR ¦ OBJ ¦ PUB ¦ PRI ¦ DAT))
ROUND (*FloatConstant*)

Возвращает: Ближайшее к указанной float-константе целое значение.

- ***FloatConstant*** – float выражение-константа, которое должно быть округлено до ближайшего целого.

Описание

ROUND - это одна из трех директив (FLOAT, ROUND и TRUNC), используемых для float-выражений-констант. ROUND возвращает целую константу, которая представляет собой ближайшее целое значение к заданной float-константе. Дробные значения ½ (.5) и выше округляются вверх до ближайшего целого, а меньшие дробные отбрасываются.

Использование ROUND

ROUND может использоваться для округления float-констант вверх или вниз до ближайшего целого значения. Отметьте, что эта директива используется только для констант, вычисляемых во время компиляции, а не во время выполнения приложения. Например:

```
CON
  OneHalf = 0.5
  Smaller = 0.4999
  Rnd1    = round(OneHalf)
  Rnd2    = round(Smaller)
  Rnd3    = round(Smaller * 10.0) + 4
```

Приведенный код создает две float-константы, OneHalf и Smaller, равные соответственно 0.5 и 0.4999. Следующие три константы, Rnd1, Rnd2 и Rnd3 – это целые константы, созданные из OneHalf и Smaller с использованием директивы ROUND. Rnd1 = 1, Rnd2 = 0, и Rnd3 = 9.

О константах в формате с плавающей точкой

Компилятор Propeller рассматривает константы с плавающей точкой как вещественные числа одинарной точности, как описано в стандарте IEEE-754. Вещественные числа

одинарной точности хранятся в 32 битах, с 1 битом на знак, 8-битной экспонентой и 23-битной мантиссой (дробная часть). Это обеспечивает примерно 7.2 значащих десятичных разряда.

Для выполнения операций с float-числами, объекты FloatMath, FloatString, Float32 и Float32Full предоставляют математические функции, совместимые с числами одинарной точности. См. «Присваивание констант ‘=’» в секции «Операторы *Spin*» на стр. 296, FLOAT на стр. 351, и TRUNC на стр. 363, а также объекты FloatMath, FloatString, Float32 и Float32Full для более подробной информации.

SPR

Регистр: Массив Регистров Специальных Функций (РСФ); обеспечивает косвенный доступ к регистрам специальных функций процессора.

((PUB ¦ PRI))
SPR [*Index*]

Возвращает: Значение регистра специальных функций по индексу *Index*.

- ***Index*** – это выражение, задающее индекс (0-15) необходимого РСФ для доступа (от PAR до VSCL).

Описание

SPR – это массив из 16 РСФ в процессоре. Элемент 0 массива – это регистр PAR , а элемент 15 – регистр VSCL. См. Табл. 4-15, приведенную ниже. Регистр SPR обеспечивает косвенный доступ к РСФ процессора.

Табл. 4-15: Регистры Специальных Функций в ОЗУ

Имя	Индекс	Тип	Описание
PAR	0	Только чтение	Параметр загрузки
CNT	1	Только чтение	Системный счетчик
INA	2	Только чтение	Входные состояния P31 - P0
INB	3	Только чтение	Входные состояния P63- P32[1]
OUTA	4	Чтение/запись	Выходные состояния P31 - P0
OUTB	5	Чтение/запись	Выходные состояния P63 – P32[1]
DIRA	6	Чтение/запись	Направления P31 - P0
DIRB	7	Чтение/запись	Направления P63 - P32[1]
CTRA	8	Чтение/запись	Управление Счетчик A
CTRB	9	Чтение/запись	Управление Счетчик B
FRQA	10	Чтение/запись	Частота Счетчик A
FRQB	11	Чтение/запись	Частота Счетчик B
PHSA	12	Чтение/запись	Фаза Счетчик A
PHSB	13	Чтение/запись	Фаза Счетчик B
VCFG	14	Чтение/запись	Конфигурация Видео
VSCL	15	Чтение/запись	Масштаб Видео

Примечание 1:Зарезервировано для будущего использования

Использование SPR

SPR может использоваться как любой другой массив *long*-элементов. В следующем примере считается, что переменная Temp определена ранее.

```
spr[4] := %11001010             'Set outa register
Temp := spr[2]                  'Get ina value
```

В этом примере регистр OUTA (индекс 4 массива SPR) устанавливается в %11001010, после чего переменная Temp устанавливается равной значению регистра INA (индекс 2 массива SPR).

_STACK

Константа: Предопределенная, один раз устанавливаемая константа для задания размера области стека для приложения.

```
CON
  _STACK = Expression
```

- ***Expression*** – целое выражение, которое указывает количество резервируемых под область стека *long*-ов.

Описание

_STACK - это предопределенная, устанавливаемая один раз опциональная константа, которая задает для приложения необходимый размер области памяти под стек. Это значение добавляется к константе **_FREE,** если она задана, для определения общего объема стека/свободной памяти, необходимого для приложения. Используйте **_STACK,** если приложению чтобы правильно выполняться необходим минимальный заданный объем стека. Если результирующее откомпилированное приложение так велико, что не позволяет зарезервировать заданный объем памяти под стек, то отобразится сообщение об ошибке. Например:

```
CON
  _STACK = 3000
```

Объявление _STACK в приведенном блоке CON указывает на то, что приложению после компиляции необходимо иметь как минимум 3000 *long*-ов свободной памяти под стек. Если откомпилированное приложение не получает столько свободного места, в сообщении об ошибке будет указано, сколько памяти не хватает. Это правильный путь, позволяющий предотвратить успешно откомпилированные приложения от ошибок при выполнении из-за недостатка памяти.

Отметьте, что лишь в верхнем объектном файле можно устанавливать значение _STACK. Любые объявления STACK в дочерних объектах будут игнорироваться компилятором. Область стека, резервируемая этой константой, используется основным процессором приложения для сохранения временных данных, таких, как стек вызовов, параметров а также промежуточных результатов при вычислениях выражений.

STRCOMP

Команда: Сравнение двух строк на равенство.

((PUB ¦ PRI))
  STRCOMP (*StringAddress1*, *StringAddress2*)

Возвращает: TRUE, если обе строки одинаковы, иначе - FALSE.

- ***StringAddress1*** – выражение, задающее адрес начала первой строки для сравнения.
- ***StringAddress2*** – выражение, задающее адрес начала второй строки для сравнения.

Описание

STRCOMP - это одна из двух команд (STRCOMP и STRSIZE), которые предоставляют информацию о строке. Команда STRCOMP сравнивает содержимое строки по адресу *StringAddress1* с содержимым строки по адресу *StringAddress2*, вплоть до ноль-терминатора в каждой строке, и возвращает TRUE, если обе строки идентичны, и FALSE – если нет. Это сравнение чувствительно к буквенному регистру.

Использование STRCOMP

В следующем примере считается, что метод PrintStr создан ранее.

```
PUB Main
  if strcomp(@Str1, @Str2)
    PrintStr(string("Str1 and Str2 are equal"))
  else
    PrintStr(string("Str1 and Str2 are different"))

DAT
  Str1 byte "Hello World", 0
  Str2 byte "Testing.", 0
```

В этом примере, в блоке DAT, созданы две строки с ноль-терминаторами – Str1 и Str2. Метод Main вызывает STRCOMP для сравнения содержимого каждой строки. Считая, что PrintStr – это метод, который отображает строку, этот пример напечатает на дисплее “Str1 and Str2 are different”.

Строки с ноль-терминаторами

При использовании команды STRCOMP предполагается, что сравниваемые строки завершаются ноль-терминаторами: за каждой из строк следует байт, равный 0. Такой формат является общепринятым и рекомендуется к использованию, поскольку большинство методов, работающих со строками, предполагают наличие ноль-терминаторов.

STRING

Директива: Объявляет строковую *in-line* константу и получает ее адрес.

((PUB ¦ PRI))
STRING (***StringExpression***)

Возвращает: Адрес строковой *in-line* константы.

- ***StringExpression*** – это желаемое строковое выражение, которое должно использоваться для единоразовых целей.

Описание

Блок DAT обычно используется для создания строк либо строковых буферов, которые могут использоваться несколько раз и для различных целей; однако есть случаи, когда строка необходима во временных целях – для отладки либо единоразового использования в объекте. Директива STRING как раз предназначена для такого использования; она компилируется в памяти в *in-line* строку с ноль-терминатором и возвращает адрес этой строки.

Использование STRING

Директива STRING очень полезна для создания строк одноразового использования и передачи адреса такой строки другим методам. Например, считаем, что метод PrintStr создан ранее.

```
PrintStr(string(''This is a test string.''))
```

В этом примере директива STRING используется для компиляции строки “This is a test string.”в памяти и возврата ее адреса как параметра для фиктивного метода PrintStr.

Если строка должна использоваться в коде более одного раза, лучше объявить ее в блоке DAT, при этом ее адрес сможет использоваться несколько раз.

STRSIZE

Команда: Получить размер строки.

((PUB ¦ PRI))
STRSIZE (***StringAddress***)

Возвращает: Размер (в байтах) строки с ноль-терминатором.

- ***StringAddress*** – выражение, задающее адрес начала измеряемой строки.

Описание

STRSIZE – это одна из двух команд (STRCOMP и STRSIZE), которые предоставляют информацию о строке. Команда STRSIZE измеряет длину строки в байтах, начиная с адреса *StringAddress*, до (но не включая), байт ноль-терминатора.

Использование STRSIZE

В следующем примере считается, что метод Print создан ранее.

```
PUB Main
  Print(strsize(@Str1))
  Print(strsize(@Str2))

DAT
  Str1 byte "Hello World", 0
  Str2 byte "Testing.", 0
```

В этом примере, в блоке DAT имеется две строки с ноль-терминаторами, Str1 и Str2. Метод Main вызывает STRSIZE для получения длины каждой из строк. Считая, что метод Print отображает величину, на дисплее отобразится 11 и 8.

Строки с ноль-терминаторами

При использовании команды STRSIZE предполагается, что строки завершаются ноль-терминаторами: за каждой из строк следует байт, равный 0. Такой формат является общепринятым и рекомендуется к использованию, поскольку большинство методов, работающих со строками, предполагают наличие ноль-терминаторов.

Символы

Каждый из приведенных в Табл. 4-16 Символов выполняет одну или более специфических функций в коде на языке *Spin*. В таблице кратко описаны функции каждого символа с ссылками на другие секции руководства, в которых эти функции описаны подробно, либо используются в примерах. Описание символов языка *Propeller*-ассемблер приведено на стр. 522.

Табл. 4-16: Символы

Символ	Назначение
%	Указатель двоичного: используется для указания, что это значение приведено в двоичном формате (основание 2). См. «Представление величин» на стр. 183.
%%	Указатель четверичного: используется для указания, что это значение приведено в четверичном формате (основание 4). См. «Представление величин» на стр. 183.
$	Указатель шестнадцатеричного: используется для указания, что это значение приведено в шестнадцатеричном формате (основание 16). См. «Представление величин» на стр. 183.
''	Указатель строки: используется в начале и конце строки текстовых символов. Обычно используется в блоках Obj (стр. 288), Dat (стр. 243), или в Pub/Pri блоках с директивой STRING (стр. 358).
@	Указатель Адреса Идентификатора: вводится непосредственно перед идентификатором для указания, что используется адрес идентификатора, а не его величина. См. «Адрес Идентификатора», стр. 324.
@@	Адрес Объекта Плюс Идентификатор: вводится непосредственно перед идентификатором для указания, что используется его величина,прибавленная к базовому адресу объекта. См. «Адрес Объекта Плюс Идентификатор», стр. 324.
_	1) Разделитель: используется как разделитель групп чисел в константах (где запятая ',' или точка '.' используются как обычный разделитель групп чисел). См. «Представление величин» на стр. 183. 2) Подчеркивание: используется как часть идентификатора. См. «Правила Идентификаторов» на стр. 183.
#	1) Ссылка «Объект-Константа»: используется для ссылки на константы суб-объекта. См. «Область видимости » в секции CON, стр. 234, и OBJ, стр. 288. 2) Начало перечисления: используется в блоках CON для установки начала набора перечислимых идентификаторов. См секцию CON в «Перечисления (Синтаксис 2 и 3)», на стр. 231.
.	1) Ссылка «Объект-Метод»: используется для ссылки на метод суб-объекта. См. OBJ, стр. 288. 2) Десятичная точка: используется в float-константах. См. CON, стр. 228.
..	Указатель диапазона: для операторов CASE указывает диапазон выражения от одного значения до другого, либо для регистра B/B указывает индекс. См. OUTA, OUTB на стр. 326, INA, INB на стр. 264, и DIRA, DIRB на стр. 249.

Табл. 4-16: Символы (продолжение)	
Символ	**Назначение**
:	1) Разделитель значения возврата: вводится перед идентификатором значения возврата в объявлениях PUB или PRI. См. PUB на стр. 334, PRI на стр. 333, и RESULT на стр. 347. 2) Присваивание Объекта: вводится в объявлении ссылки на объект в блоке OBJ. См. OBJ, стр. 288. 3) Разделитель оператора Case: вводится сразу после выражений совпадения в структуре CASE. См. CASE, стр. 200.
|	1) Разделитель локальных переменных: вводится прямо перед перечнем локальных переменных в объявлениях PUB или PRI. См. PUB, стр. 334 и PRI, стр. 333. 2) Побитовое ИЛИ: используется в выражениях. См. «Побитовое ИЛИ (OR) '|', '|='» на стр. 315.
\	Ловушка Abort: вводится прямо перед вызовом метода, который потенциально может выйти по abort. См. ABORT на стр. 187.
,	Разделитель списка: используется для разделения элементов в списках. См. LOOKUP, LOOKUPZ на стр. 285, LOOKDOWN, LOOKDOWNZ на стр. 283, а также секцию DAT «Объявление Данных (Синтаксис 1)» на стр. 244.
( )	Указатели списка параметров: используются для заключения параметров метода. См. PUB, стр. 334 и PRI, стр. 333.
[ ]	Указатели индекса массива: используются для заключения индексов массива переменных или для ссылок на Основную Память. См. VAR, стр. 364; BYTE, стр. 192; WORD, стр. 382; и LONG, стр. 274.
'	Указатель комментария кода: используется для ввода внутристроковых комментариев кода (некомпилируемый текст) в целях просмотра текста. См. «Упражнение 3: Output.spin» на стр. 118.
''	Указатель комментария документации: используется для ввода внутристроковых комментариев документации (некомпилируемый текст) для просмотра документации. См. «Упражнение 3: Output.spin» на стр. 118.
{ }	Указатель одностроковых (In-line)/многостроковых (multi-line) комментариев кода: используется для ввода многостроковых комментариев кода (некомпилируемый текст) для просмотра кода.
{{ }}	Указатель одностроковых (In-line)/многостроковых (multi-line) комментариев: используется для ввода многостроковых комментариев документации (некомпилируемый текст) для просмотра документации. См. «Упражнение 3: Output.spin» на стр. 118.

TRUNC

Директива: Отбрасывание, “усечение,” дробной части float-константы.

((CON ¦ VAR ¦ OBJ ¦ PUB □ PRI □ DAT))
TRUNC (***FloatConstant***)

Возвращает: Целое, представляющее собой исходную float-константу с отсеченной дробной частью.

- ***FloatConstant*** – float-константа, отсекаемая до целого.

Описание

TRUNC - это одна из трех директив (FLOAT, ROUND и TRUNC), используемых в float-выражениях-константах. TRUNC возвращает целую константу, которая представляет собой исходную float-константу ***FloatConstant*** с отброшенной дробной частью.

Применение TRUNC

TRUNC используется для получения целой части float-константы. Например:

```
CON
  OneHalf = 0.5
  Bigger = 1.4999
  Int1 = trunc(OneHalf)
  Int2 = trunc(Bigger)
  Int3 = trunc(Bigger * 10.0) + 4
```

Этот код создает две float-константы, OneHalf и Bigger, равные 0.5 и 1.4999. Следующие три константы, Int1, Int2 и Int3, - это целые константы, полученные из OneHalf и Bigger с использованием директивы TRUNC. Int1 = 0, Int2 = 1, и Int3 = 18.

О константах в формате с плавающей точкой

В компиляторе Propeller константы с плавающей точкой представляются как вещественные числа одинарной точности, согласно стандарту IEEE-754. Вещественные числа одинарной точности хранятся в 32 битах, с 1 битом на знак, 8-битной экспонентой и 23-битной мантиссой (дробная часть). Это обеспечивает примерно 7.2 значащих десятичных разряда. Для выполнения операций с float-числами, объекты FloatMath, FloatString, Float32 и Float32Full предоставляют математические функции, совместимые с числами одинарной точности. Для более подробной информации см. «Присваивание констант ‘=’» в секции «Операторы *Spin*» на стр. 296, FLOAT на стр. 253, и TRUNC на стр. 363, а также объекты FloatMath , FloatString, Float32, и Float32Full.

VAR

Объявление: Объявляет Блок Переменных .

VAR
  Size Symbol ⟨[*Count*]⟩ ⟨ ↳*Size Symbol* ⟨ [*Count*]⟩⟩...

VAR
  Size Symbol ⟨[*Count*]⟩ ⟨ , *Symbol* ⟨ [*Count*]⟩⟩...

- ***Size*** – желаемый размер переменной: BYTE, WORD или LONG.
- ***Symbol*** – желаемое имя переменной.
- ***Count*** – опциональное выражение, заключенное в квадратные скобки, указывающее, что эта переменная – массив с количеством элементов *Count,* каждый размером, соответственно, байт, слово или двойное слово. При дальнейших обращениях к этим элементам необходимо учитывать, что они начинаются с индекса 0 и заканчиваются индексом *Count*-1.

Описание

Директива VAR – это объявление Блока Переменных. Блок Переменных представляет собой секцию исходного кода, в которой объявляются идентификаторы глобальных переменных. Это одна из шести специальных директив (CON, VAR, OBJ, PUB, PRI, и DAT), которые обеспечивают четкую структуру языка *Spin*.

Объявление переменных (Синтаксис 1)

Наиболее общая форма объявления переменных начинается с VAR с последующими одним или более объявлениями. Объявление VAR должно вводиться с первой (самой левой) колонки строки, а последующие объявления должны вводиться с отступом как минимум в один пробел.

```
VAR
  byte Str[10]
  word Code
  long LargeNumber
```

В этом примере определяется переменная Str как массив из 10 байтов, переменная Code как слово (два байта), и переменная LargeNumber – как двойное слово (четыре байта). Методы *public* и *private* этого же объекта могут обращаться к приведенным переменным следующим образом:

```
PUB SomeMethod
  Code := 60000
  LargeNumber := Code * 250
  GetString(@Str)
  if Str[0] == "A"
    <more code here>
```

Отметьте, что переменные Code и LargeNumber используются в выражениях напрямую. Обращение к Str в параметре метода GetString выглядит иначе: ей предшествует оператор Адреса идентификатора @. Это необходимо, потому как наш фиктивный метод GetString должен записать значение назад в переменную Str. Если бы мы просто указали GetString(Str), то в GetString передался бы только первый байт Str, т.е. элемент 0. При использовании оператора Адреса идентификатора, @, мы передаем в метод GetString адрес переменной Str; метод GetString может использовать этот адрес для записи в элементы Str. И в конце мы используем Str[0] в условии оператора IF, чтобы увидеть, равен ли первый байт символу "A". Запомните, что первый элемент любого массива всегда равен нолю (0).

Объявление переменных (Синтаксис 2)

Этот синтаксис представляет собой разновидность Синтаксиса 1, позволяющую объявлять переменные одинакового размера в виде списка идентификаторов, разделенных запятыми. Следующий пример похож на предыдущий, но здесь мы объявляем две переменных, каждая размером в слово: Code и Index.

```
VAR
  byte Str[10]
  word Code, Index
  long LargeNumber
```

Диапазоны значений переменных

Диапазон значений и размер переменной каждого типа представлены ниже:

- BYTE – 8 битов (беззнаковое); от 0 до 255.
- WORD – 16 битов (беззнаковое); от 0 до 65535.
- LONG – 32 бита (знаковое); от -2147483648 до +2147483647.

Более детально диапазоны и использование переменных описаны соответственно: BYTE на стр. 192, WORD на стр 382, и LONG – на стр. 274.

Организация переменных

Во время компиляции объекта все объявления в их общих Блоках Переменных объединяются вместе по своим типам. Переменные в ОЗУ располагаются согласно следующего порядка: сначала все двойные слова, затем слова и в самом конце – однобайтовые переменные. Это выполняется для того, чтобы наиболее эффективно упаковать переменные в ОЗУ и избежать ненужных «дыр» в памяти. Помните это, когда пишете код, в котором производится косвенный доступ к переменным, базируясь на их относительном расположении по отношению друг к другу.

Оптимизированная адресация

В откомпилированом Propeller-приложении первые восемь (8) *long*-ов, которые образуют параметры, переменную RESULT и локальные переменные, адресуются по оптимизированной схеме. Это значит, что доступ к этим первым восьми двойным словам (параметрам, RESULT, и локальным переменным) требует немного меньше времени, чем доступ к девятому и далее, *long*-ам. Для оптимизации скорости выполнения кода необходимо убедиться в том, что все локальные переменные, используемые в наиболее часто повторяющихся местах метода, находятся в числе этих восьми. Подобный механизм применим также и для локальных переменных; см. PUB, на стр. 334, для подробной информации.

Область видимости переменных

Переменные, определенные в Блоке Переменных, являются глобальными по отношению к своему объекту, но не за его пределами. Это значит, они доступны напрямую из любого *public*- либо *private*-метода в рамках объекта, и их имена не будут конфликтовать с такими же, объявленными в других, родительских либо дочерних, объектах.

Отметьте, что *public*- и *private*-методы могут объявлять свои собственные локальные переменные. См. PUB, стр. 334, и PRI, стр. 333.

Глобальные переменные не доступны извне объекта, за исключением случая передачи либо возврата адреса переменной в/из другого объекта при вызове метода.

Область видимости распространяется за границы процессора

Область видимости глобальных переменных не ограничивается каким-то единственным процессором. Методы *public* и *private* имеют естественный доступ к своим глобальным переменным не зависимо от того, в каком процессоре они выполняются. В случае, когда объект выполняет некоторые из своих методов в

различных *cog*-ах, каждый из этих методов, а, значит, и *cog*-ов, в которых они выполняются, имеет доступ к этим переменным.

Это свойство позволяет единственному объекту с легкостью использовать преимущества централизованного управления параллельной обработкой. К примеру, умело написанный объект может этим воспользоваться для мгновенного изменения «глобальных» установок, используемых каждым из ранее созданных и работающих параллельно программных экземпляров.

Несомненно, должно уделяться достаточное внимание, чтобы убедиться, что параллельная работа с одним и тем же блоком переменных не создаст нежелательных ситуаций. См. примеры в секции LOCKNEW на стр. 268.

VCFG

Регистр: Регистр Конфигурации Видео.

((PUB ¦ PRI))
VCFG

Возвращает: Текущее значение Регистра Конфигурации Видео процессора, если используется как переменная-источник.

Описание

VCFG — это один из двух регистров (VCFG и VSCL), которые влияют на работу Видео Генератора процессора. Каждый из процессоров имеет модуль видео-генератора, который обеспечивает передачу данных видео-изображения на постоянной скорости. Регистр VCFG содержит конфигурационные установки видео-генератора, как показано в Табл. 4-17.

Табл. 4-17: Регистр VCFG

Биты VCFG									
31	**30..29**	**28**	**27**	**26**	**25..23**	**22..12**	**11..9**	**8**	**7..0**
-	VMode	CMode	Chroma1	Chroma0	AuralSub	-	VGroup	-	VPins

Поля с VMode по AuralSub удобно записывать с помощью инструкции MOVI, поле VGroup – с помощью инструкции MOVD, а поле VPins – с помощью инструкции MOVS языка Propeller ассемблер.

Поле VMode

Двухбитовое поле VMode (video mode, режим видео) выбирает тип и размещение видео-выхода, если он используется, согласно Табл. 4-18.

Табл. 4-18: Поле VMode

VMode	Видео-режим
00	Выключено, видеосигнал не генерируется.
01	VGA -режим; 8-битный параллельный вывод на линиях VPins7:0
10	Композитный режим 1; радиосигнал на VPins 7:4, видеосигнал на VPins 3:0
11	Композитный режим 2; видеосигнал на VPins 7:4, радиосигнал на VPins 3:0

Поле CMode

Поле CMode (color mode, режим цветности) выбирает двухцветный либо четырехцветный режим. При CMode = 0 режим – двухцветный, пиксели – 32 бита на 1 бит и могут использоваться только цвета 0 и 1. При CMode = 1 – четырехцветный режим, данные пикселей – это 16 бит на 2 бита и используются цвета от 0 до 3.

Поле Chroma1

Бит Chroma1 (предавать сигнал цветности в радиосигнале) включает или выключает цветность в радиочастотном сигнале. 0 = выключено, 1 = включено.

Поле Chroma0

Бит Chroma0 (предавать сигнал цветности в видеосигнале) включает или выключает цветность в видео сигнале. 0 = выключено, 1 = включено.

Поле AuralSub

Поле AuralSub (aural sub-carrier, звуковая поднесущая) выбирает источник для ЧМ-модуляции звуковой поднесущей. Источником может быть ФАПЧ PLLA одного из процессоров, выбранный соответственно значению AuralSub.

Табл. 4-19: Поле AuralSub

AuralSub	Источник частоты поднесущей
000	PLLA *Cog* 0
001	PLLA *Cog* 1
010	PLLA *Cog* 2
011	PLLA *Cog* 3
100	PLLA *Cog* 4
101	PLLA *Cog* 5
110	PLLA *Cog* 6
111	PLLA *Cog* 7

Поле VGroup

Поле VGroup (группа линий видеовыхода) выбирает, какая из групп по 8 линий будет выводить видеосигнал.

Табл. 4-20: Поле VGroup

VGroup	Группа линий
000	Группа 0: P7..P0
001	Группа 1: P15..P8
010	Группа 2: P23..P16
011	Группа 3: P31..P24
100-111	<зарезервировано >

Поле VPins

Поле VPins (линии видео-выхода) представляет собой маску, применяемую к линиям VGroup, которая указывает, на каких линиях выводить видео сигнал.

Табл. 4-21: Поле VPins

VPins	Результат
00001111	Вывод видео на младшие 4 бита, композит
11110000	Вывод видео на старшие 4 бита, композит
11111111	Вывод видео на все 8 бит; режим VGA

Использование VCFG

Регистр `VCFG` может быть считан/записан как и любые другие регистры или предопределенные переменные. Например:

```
VCFG := %0_10_1_0_1_000_00000000000_001_0_00001111
```

Этот код устанавливает регистр конфигурации видео для включения видео в композитном режиме 1, с 4 цветами, хромо (цвет) в радиосигнале включен, на группе линий 1, нижние 4 линии (то есть линии P11:8).

VSCL

Регистр: Регистр Масштаба Видео.

((PUB ¦ PRI))
  VSCL

Возвращает: Текущее значение Регистра Масштаба Видео процессора, если используется как переменная-источник.

Описание

VSCL - это один из двух регистров (VCFG и VSCL), которые влияют на работу Видео Генератора процессора. Каждый *Cog* имеет модуль видео-генератора, который обеспечивает передачу данных видео-изображения на постоянной скорости. Регистр VSCL устанавливает скорость, на которой генерируются видео-данные.

Табл. 4-22: Регистр VSCL		
Биты VSCL		
31..20	19..12	11..0
–	PixelClocks	FrameClocks

Поле PixelClocks

Поле *PixelClocks* из 8 бит указывает количество тактов на пиксель: количество тактов, которое проходит перед тем, как модулем видео-генератора выдается каждый следующий пиксель. Эти такты – это такты PLLA, а не Системного Генератора.

Поле FrameClocks

Поле *FrameClocks* из 12 битов указывает количество тактов на фрейм: количество тактов, которое должно пройти перед тем, как модулем видео-генератора будет выдан новый фрейм. Эти такты – такты PLLA, а не Системного Генератора. Фрейм – это одно двойное слово данных для пикселей (поставляемое командой WAITVID). Поскольку данные пикселей могут быть либо 16 на 2 бит, либо 32 на 1 бит (то есть соответственно 16 бит в ширину с 4 цветами, или 32 пикселя в ширину с 2 цветами), то поле *FrameClocks* обычно в 16 или 32 раза больше величины поля *PixelClocks*.

Использование VSCL

Поле VSCL может быть прочитано/записано как и любые другие регистры или предопределенные переменные. Например:

```
VSCL := %000000000000_10100000_1010000000000
```

Этот код устанавливает регистр масштаба видео в значение 160 для поля *PixelClocks* и 2560 для поля *FrameClocks* (для фрейма в 16-пикселов на 2 бита цвета). Реальная скорость, на которой выводятся пиксели, зависит от частоты PLLA с заданным масштабным фактором.

WAITCNT

Команда: Временно приостанавливает работу процессора.

((PUB ¦ PRI))
 WAITCNT (*Value*)

- ***Value*** – желаемое 32-битное значение Системного Счетчика для ожидания.

Описание

WAITCNT, "Wait for System Counter," (ожидать Системный Счетчик) – это одна из четырех команд ожидания (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки работы процессора до выполнения заданного условия. WAITCNT приостанавливает *Cog* до достижения глобальным Системным Счетчиком значения *Value*.

При своем выполнении, команда WAITCNT активизирует специальный аппаратный механизм "ожидания" в процессоре, который не позволяет Системному Генератору выполнять дальнейший код в этом процессоре до тех пор, пока Системный Счетчик не достигнет значения *Value*. Аппаратный механизм ожидания проверяет системный Счетчик каждый цикл Системного Генератора, и потребление энергии процессором в это время уменьшается примерно на $7/8^{-x}$. Команда WAITCNT может использоваться в приложениях для уменьшения потребляемой энергии, особенно в таких программах, где тратится большое количество времени в циклах ожидания редких событий.

Существует два типа задержек, организуемых с помощью WAITCNT: фиксированные задержки и синхронизированные задержки. Они описаны ниже.

Фиксированные задержки

Фиксированные задержки – это такие, которые никак не зависят от конкретно взятой точки времени и служат только для приостановки выполнения кода на заданное количество тактов. Например, фиксированная задержка может использоваться для ожидания 10 миллисекунд после прихода какого-либо события перед выполнением других действий. Например:

```
CON
  _clkfreq = xtal1              'Set for slow crystal
  _xinfreq = 5_000_000          'Use 5 MHz accurate crystal
  repeat
   !outa[0]                     'Toggle pin 0
   waitcnt(50_000 + cnt)        'Wait for 10
```

В этом примере переключается состояние линии В/В P0, после чего ожидается 50000 тактов системной частоты перед следующим проходом цикла. Помните, что параметр *Value* – это 32-битная величина, с которой должно совпасть значение Системного Счетчика. Поскольку Системный Счетчик – это глобальный ресурс, который изменяет свое значение каждый такт системной частоты, то для задержки на заданное количество тактов от текущего положения, нам необходимо добавить это количество к текущему значению Системного Счетчика. Идентификатор `cnt` в "`50_000 + cnt`" – это переменная Регистра Системного Счетчика; она возвращает текущее значение Системного Счетчика в данный момент времени. Поэтому в нашем коде задается ожидание значения 50000 тактов плюс текущее значение Системного Счетчика; т.е.: ожидать 50000 тактов от текущего времени. При внешнем 5 МГц резонаторе, 50000 тактов – это около 10 мсек (1/100-я секунды).

ВАЖНО: Поскольку **`WAITCNT`** приостанавливает *Cog* до достижения Системным Счетчиком заданного значения, необходимо уделять внимание тому, чтобы это значение не было им уже пройдено. Если такое случится, и Системный Счетчик пройдет заданное значение перед активацией аппаратного механизма ожидания, то процессор будет вести себя как «подвисший», хотя на самом деле он будет ждать, пока Системный Счетчик переполнит 32-битное значение и дойдет до заданной величины. Даже при тактовой частоте в 80 МГц для переполнения 32-х разрядного регистра потребуется более 53 секунд!

Исходя из вышеизложенного, при использовании **`WAITCNT`** в коде *Spin* убеждайтесь, что записываете выражение с *Value* в том же виде, как сделали это мы: "`offset + cnt`", а не "`cnt + offset`." Это необходимо, поскольку интерпретатор *Spin* будет вычислять это выражение слева направо, а каждое промежуточное вычисление в выражении требует время на выполнение. Если бы `cnt` стояло в начале выражения, то сначала бы прочитался регистр Системного Счетчика, а затем выполнялось вычисление всего остального выражения, требуя неопределенное количество тактов и превращая наше значение `cnt` в довольно старое на момент вычисления финального результата. Однако, ввод `cnt` последним в выражение **`WAITCNT`** гарантирует фиксированное количество экстра тактов между чтением регистра Системного Счетчика и активизацией механизма ожидания. На самом деле, если команда записана в виде `waitcnt(offset + cnt)`, интерпретатор дополнительно затрачивает 381 такт. Это значит, что для исключения непредвиденно длинных задержек, значение `offset` должно всегда быть больше либо равно 381.

Синхронизированные задержки

Синхронизированные задержки – это такие задержки, которые напрямую зависят от заданной точки времени, т.н. базовой точки, и служат для "выравнивания по времени"

будущих событий относительно этой точки. К примеру, синхронизированные задержки могут использоваться для ввода или вывода сигнала с заданным интервалом, не зависимо от неопределенного количества времени, затраченного на выполнение самого кода. Чтобы понять, в чем здесь отличие от примера с Фиксированной Задержкой, рассмотрим его временную диаграмму.

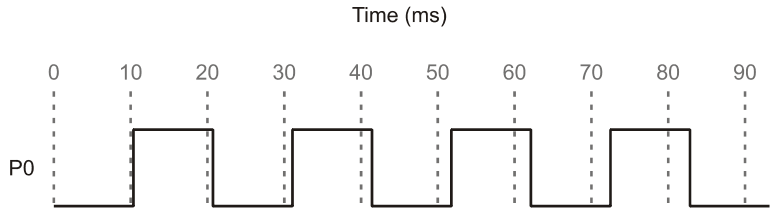


Рис. 4-1: Временная диаграмма для фиксированной задержки

На Рис. 4-1 показано состояние выхода для предыдущего примера, примера с фиксированной задержкой. Вы заметили, что линия В/В P0 переключается примерно, а не строго, каждые 10 миллисекунд? Действительно, здесь присутствует кумулятивная погрешность, которая приводит к тому, что последовательные переключения происходят все дальше и дальше от точек синхронизации по отношению к начальному моменту, 0 мсек. Длина задержки остается 10 мсек, однако возникает ошибка, так как в ней не учитывается время выполнения остальной части тела цикла. Операторы `repeat`, `!outa[0]` и **WAITCNT** каждый требуют хоть и небольшого, но времени для выполнения, и это «лишнее» время добавляется к 10 мсек задержке, задаваемой в **WAITCNT**.

Если использовать **WAITCNT** немного иначе, как синхронизированную задержку, мы можем исключить эту накапливающуюся ошибку. В следующем примере считаем, что используется внешний резонатор на 5 МГц.

```
CON
  _clkfreq = xtal1              'Set for slow crystal
  _xinfreq = 5_000_000          'Use 5 MHz accurate crystal

PUB Toggle | Time
  Time := cnt                   'Get current system counter value
  repeat
    waitcnt(Time += 50_000)     'Wait for 10 msec
    !outa[0] 'Toggle pin 0
```

В этом примере сначала сохраняется текущее значение Системного Счетчика, `Time := cnt`, затем запускается цикл `repeat`, в котором ожидается достижение Системным

Счетчиком значения Time + 50000, переключается состояние линии В/В P0 и выполняется возврат на начало цикла для следующего прохода. Оператор Time += 50_000 – это оператор присваивания, он складывает значение переменной Time с 50000, сохраняет его назад в Time, и затем запускает команду WAITCNT с полученным результатом. Отметьте, что мы запросили значение Системного Счетчика только один раз, в начале примера; это и есть наше базовое время. Затем мы ожидаем, пока значение Системного Счетчика станет равным значению базовой точки плюс 50000 и выполняем действия в цикле. Каждый последующий проход цикла мы ожидаем достижения значением Системного Счетчика следующего, кратного 50000 от базового времени, значения. Такой метод автоматически учитывает время, затрачиваемое на выполнение самих операторов цикла: repeat, !outa[0] и waitcnt. В результате на выходе будем иметь диаграмму, приведенную на Рис. 4-2.

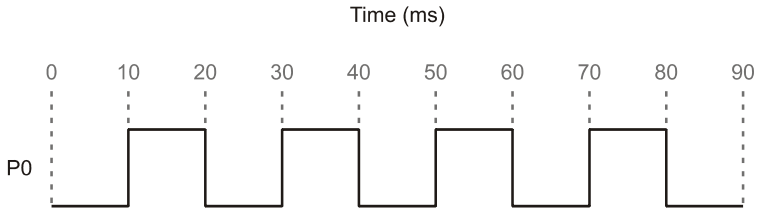


Рис. 4-2: Временная диаграмма для синхронизированной задержки

При использовании метода синхронизированных задержек, наш сигнал всегда точно выровнен по отношению к временной базе, кратно времени нашего интервала. Этот метод будет работать, пока мы имеем точную временную базу (внешний резонатор), а также пока время выполнения команд цикла не превышает длительности самого интервала. Мы ожидали перед первым переключением (при помощи WAITCNT), поэтому время между самым первым, вторым и остальными переключениями совпадают.

Вычисление времени

Объект может выполнять задержку на заданный интервал времени, даже если приложение будет изменять частоту Системного Генератора. Для реализации такой задержки используется команда WAITCNT, комбинированная с выражением, включающим значение текущей частоты Системного Генератора (CLKFREQ). Например, даже не зная, какая действительно частота будет в использующем Ваш объект приложении, задержка выполнения процессором кода на 1 миллисекунду (до тех пор, пока частота достаточно высока), может быть организована следующей строкой:

```
waitcnt(clkfreq / 1000 + cnt) 'delay Cog 1 millisecond
```

Для более детальной информации, см. CLKFREQ на стр. 204.

WAITPEQ

Команда: Приостановить процессор до получения заданной комбинации на линиях В/В.

((PUB ¦ PRI))
  WAITPEQ (*State*, *Mask*, *Port*)

- ***State*** – логические состояния, с которыми сравниваются состояния на линиях. Это 32-битное значение, которое указывает состояния до 32 линий В/В. *State* сравнивается либо с (INA & *Mask*), либо с (INB & *Mask*), в зависимости от *Port*.
- ***Mask*** – отслеживаемые линии. *Mask* – это 32-битное значение, которое содержит биты, установленные в '1' для каждой отслеживаемой линии, и сброшенные в '0' – для линий, которые не контролируются. *Mask* побитно умножается по И с 32-битным значением входных состояний, а результат сравнивается со значением *State*.
- ***Port*** – это 1-битное значение, указывающее номер порта В/В для контроля; 0 = Port A, 1 = Port B. В текущей версии кристалла ИМС Propeller (P8X32A) реализован только Port A.

Описание

WAITPEQ, ("Wait for Pin(s) to Equal", "ожидать совпадения пинов") – это одна из четырех команд ожидания (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки выполнения процессором кода до достижения заданных условий. WAITPEQ приостанавливает процессор до тех пор, пока значение состояний линий В/В порта, побитно умноженное по И на *Mask*, не совпадет со значением *State*.

При своем выполнении, команда WAITPEQ активизирует специальный аппаратный механизм "ожидания", который отключает системную частоту от данного процессора, запрещая дальнейшее выполнение кода, до тех пор, пока состояние на заданном пине (или группе пинов) не станет равным заданному. Аппаратный механизм ожидания проверяет заданные линии В/В каждый такт Системного Генератора, и потребление энергии процессором в это время уменьшается примерно на $7/8^{-x}$.

Использование WAITPEQ

Команда WAITPEQ предоставляет очень удобный механизм синхронизации кода со внешними событиями. Например:

```
waitpeq(%0100, %1100, 0)     'Wait for P3 & P2 to be low & high
```

```
outa[0] := 1 'Set P0 high
```

Приведенный выше код приостанавливает процессор до тех пор, пока линия B/B 3 не станет равной 0 и линия B/B 2 не станет 1, после чего устанавливает линию B/B P0 в 1.

Использование изменяемых номеров линий B/B

В объектах Propeller довольно часто возникает задача контролировать состояние отдельного пина, номер которого задается вне самого объекта. Простым путем преобразования этого номера пина в соответствующие 32-битные значения *State* и *Mask* является использование побитного оператора Дешифровать (Decode, "|<"), подробно описанного на стр. 309. Например, если номер пина был задан в переменной Pin, и нам нужно ожидать до тех пор, пока он не станет в состояние '1', мы можем использовать следующий код:

```
waitpeq(|< Pin, |< Pin, 0)      'Wait for Pin to go high
```

Параметр *Mask*, равный |< Pin, преобразует заданный номер Pin в 32-битное значение, в котором только один бит установлен в '1' – бит, соответствующий номеру пина Pin.

Ожидание переходов (фронтов импульсов)

Если нам необходимо ожидать перехода из одного состояния в другое (например, из высокого в низкий уровень), мы можем использовать следующий код:

```
waitpeq(%100000, |< 5, 0)          'Wait for Pin 5 to go high
waitpeq(%000000, |< 5, 0)          'Then wait for Pin 5 to go low
```

В этом примере сначала ожидается состояние высокого уровня на P5, затем на нем же ожидается состояние низкого уровня – это и есть переход высокое-низкое состояние. Если бы мы использовали только вторую строку, без первой, процессор бы вовсе не остановился, если бы на линии P5 изначально был низкий уровень.

WAITPNE

Команда: Приостановить процессор, пока на линиях B/B присутствует заданная комбинация.

((PUB ¦ PRI))
  WAITPNE (*State*, *Mask*, *Port*)

- ***State*** – логические состояния, с которыми сравниваются состояния на линиях. Это 32-битное значение, которое указывает состояния до 32 линий B/B. *State* сравнивается либо с (INA & *Mask*), либо с (INB & *Mask*), в зависимости от *Port*.
- ***Mask*** – отслеживаемые линии. *Mask* – это 32-битное значение, которое содержит биты, установленные в '1' для каждой отслеживаемой линии, и сброшенные в '0' – для линий, которые не контролируются. *Mask* побитно умножается по И с 32-битным значением входных состояний, а результат сравнивается со *State*.
- ***Port*** – это 1-битное значение, указывающее номер порта B/B для контроля; 0 = Port A, 1 = Port B. В текущей версии (P8X32A) реализован только Port A.

Описание

WAITPNE, ("Wait for Pin(s) Not Equal", "ожидать несовпадения пинов") – это одна из четырех команд ожидания (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки выполнения процессором кода до достижения заданных условий. WAITPNE – обратная форма WAITPEQ, она приостанавливает процессор, пока значение состояний линий B/B порта, побитно умноженное по И на *Mask*, совпадает со *State*.

При выполнении, команда WAITPNE активизирует аппаратный механизм "ожидания", отключающий системную частоту от процессора, запрещая выполнение в нем кода, пока состояние на заданном пине (или группе пинов) равно указанному. Аппаратный механизм ожидания проверяет заданные линии B/B каждый такт Системного Генератора, при этом потребление энергии процессором уменьшается примерно на 7/8.

Использование WAITPNE

WAITPNE дает удобный механизм синхронизации со внешними событиями. Например:

```
waitpeq(%0100, %1100, 0) 'Wait for P3 & P2 to be low & high
waitpne(%0100, %1100, 0) 'Wait for P3 & P2 to not match prev. state
outa[0] := 1             'Set P0 high
```

Этот код приостанавливает *Cog* до достижения P3 = 0 и P2 =1, затем приостанавливает *Cog*, пока один или оба этих пина не изменят состояние, и затем устанавливает P0 =1.

WAITVID

Команда: Приостановить процессор, пока его Видео Генератор не будет готов принять данные.

((PUB ¦ PRI))
  WAITVID (*Colors*, *Pixels*)

- ***Colors*** – величина *long*, представляющая четыре байтовых значения цвета, каждое из которых описывает четыре возможных цвета пикселей в *Pixels*.
- ***Pixels*** – это следующий для вывода набор 16-пикселов на 2 бита (или 32 на 1).

Описание

WAITVID, (“Wait for Video Generator”, “Ожидать Видео Генератор”) – это одна из четырех команд (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки работы процессора, пока не выполнится заданное условие. WAITVID приостанавливает *Cog*, пока его аппаратный Видео Генератор не будет готов принять следующие данные пикселей, после чего Видео Генератор принимает данные и процессор выполняет следующую строку кода.

При своем выполнении, команда WAITVID активизирует аппаратный механизм “ожидания”, отключающий системную частоту от процессора, запрещая выполнение в нем кода, пока Видео Генератор не будет готов. Аппаратный механизм ожидания проверяет статус Видео Генератора на каждом такте Системного Генератора, при этом потребление энергии процессором существенно уменьшается.

Использование WAITVID

WAITVID предоставляет простой механизм доставки данных аппаратному Видео Генератору процессора. Поскольку Видео Генератор работает независимо от самого процессора, то последний должен синхронизировать каждый момент, когда данные необходимы устройству отображения. Частота, на которой это происходит, зависит от типа дисплея и соответствующих установок для Видео Генератора, но, в любом случае, процессор должен всегда иметь готовые новые данные на тот момент, когда Видео Генератор будет готов их принять. Процессор использует команду WAITVID, чтобы дождаться необходимого момента и передать эти данные в Видео Генератор.

Параметр *Colors* – это 32-битное значение, которое содержит либо четыре 8-битных значений цвета (для 4-х цветного режима), либо два 8-ми битных значения в нижних 16-ти битах (для 2-х цветного режима). Для режима VGA, старшие 6 бит каждого

значения цвета – это 2-битный красный, 2-битный зеленый и 2-битный синий компоненты цвета, описывающие необходимый оттенок; младшие 2 бита – не используются. Каждое из значений цвета соответствует одному из четырех возможных цветов для 2-х битных пикселей (когда *Pixels* используется в наборе 16x2), или одному из двух возможных цветов для 1-битных пикселей (когда *Pixels* используется в наборе 32x1).

Pixels описывают набор пикселей для отображения – 16 либо 32 пикселя – в зависимости от цветовой глубины в конфигурации Видео Генератора.

Для получения примеров использования **WAITVID**, просмотрите объекты *TV* и *VGA*.

Перед тем, как выполнять команду **WAITVID**, необходимо убедиться в том, что Видео Генератор и Счетчик A процессора запущены, иначе можно получить бесконечный цикл ожидания. См. **VCFG** на стр. 368 и **VSCL** на стр. 371, а также **CTRA, CTRB** на стр. 239.

WORD

Объявление: Объявляет либо переменную размером *word* (слово), либо данные размером *word* и выровненные по границе *word*, либо читает/записывает *word* основной памяти.

```
VAR
   WORD Symbol {[Count]}
```

```
DAT
   {Symbol}  WORD Data {[Count]}
```

```
((PUB | PRI))
   WORD [BaseAddress] {[Offset]}
```

```
((PUB | PRI))
   Symbol.WORD {[Offset]}
```

- ***Symbol*** – желаемое имя переменной (синтаксис 1), блока данных (синтаксис 2) или существующее имя переменной (Синтаксис 4).
- ***Count*** – опциональное выражение, указывающее количество элементов размером *word* для идентификатора *Symbol* (синтаксис 1), либо количество элементов ***Data***, размером слово, для хранения в виде таблицы данных (синтаксис 2).
- ***Data*** – выражение-константа либо список выражений, разделенных запятыми.
- ***BaseAddress*** – это выражение, представляющее собой выровненный по границе слова адрес в основной памяти для чтения или записи. Если *Offset* опущен, то *BaseAddress* – это реальный адрес данных. Если *Offset* задан, реальный адрес данных равен *BaseAddress* + *Offset*.
- ***Offset*** – это опциональное выражение, указывающее смещение до данных относительно адреса *BaseAddress*, либо смещение от байта 0 *Symbol*. Величина *Offset* представляется в значениях размером слово.

Описание

WORD - это одно из трех объявлений (BYTE, WORD, и LONG), которые объявляют данные либо оперируют с памятью. Объявление WORD может использоваться для:

1) объявления идентификатора переменной размером *word* (слово, 16-бит) либо массива из таких идентификаторов в блоке VAR, или
2) объявления *word* -выровненных и/или *word* -размерных данных в блоке DAT, или
3) чтения или записи слова в основной памяти по базовому адресу со смещением,
4) доступ к отдельным словам переменной размера *long* (двойное слово, 32 бита).

Диапазон значений Word

Ячейка памяти размером в слово (*word*, 16 бит) может содержать значение, равное одной из 2^{16} возможных комбинаций битов (то есть одной из 65536 комбинаций). Это обеспечивает для величин с размером в одно слово диапазон значений от 0 до 65535. Поскольку язык *Spin* выполняет все математические операции, используя 32-битную арифметику со знаком, то любые величины размера *word* имеют внутреннее представление как положительные величины размера *long*. Однако число, которое содержится в *long*, в известной степени зависит от представления компьютера и пользователя о нем. К примеру, в *Spin*-выражении, для перевода значения *word,* рассматриваемого как величина со знаком (от -32768 до +32767), в соответствующую величину со знаком размера *long*, можно использовать оператор «Распространение Знака 15 или Пост- Установка '~~'», стр. 306.

Объявление переменной типа Word (Синтаксис 1)

В блоках **VAR,** для объявления глобальных переменных, которые либо имеют размер в слово, либо являются массивом из слов, используется синтаксис 1 объявления **WORD.** Например:

```
VAR
  word Temp                 'Temp is a word (2 bytes)
  word List[25]             'List is a word array
```

В этом примере объявлены две переменные с идентификаторами `Temp` и `List`. `Temp` – это просто одиночная переменная размером слово. В строке под объявлением переменной `Temp` используется опциональное поле *Count* для создания массива с именем `List` из 25 элементов –переменных, каждая размером слово. Как `Temp`, так и `List`, доступны из любого метода **PUB** или **PRI** в рамках того объекта, в котором объявлен их блок **VAR**; они глобальны по отношению к объекту. Например:

```
PUB SomeMethod
  Temp := 25_000            'Set Temp to 25,000
  List[0] := 500            'Set first element of List to 500
  List[1] := 9_000          'Set second element of List to 9,000
  List[24] := 60_000        'Set last element of List to 60,000
```

Для дополнительной информации о таком формате использования **WORD**, см. «Объявление переменных (Синтаксис 1)» секции **VAR** на стр. 364, и помните, что в поле *Size* в этом описании используется **WORD**.

Объявление данных Word (Синтаксис 2)

В блоках DAT для объявления данных, выровненных по размеру *word* и/или с размером *word*, которые компилируются в главной памяти как константы, используется синтаксис 2 объявления WORD. Блоки DAT в таком объявлении позволяют иметь предваряющий его опциональный идентификатор, который может быть использован для дальнейших ссылок на эти данные (см. DAT, стр. 243). Например:

```
DAT
  MyData word 640, $AAAA, 5_500      'Word-aligned/word-sized data
  MyList byte word $FF99, word 1_000 'Byte-aligned/word-sized data
```

В этом примере объявлено два идентификатора: MyData и MyList. MyData указывает на начало *word* -выровненных и *word* -размерных данных в основной памяти. Значения MyData в основной памяти – это соответственно 640, $AAAA и 5500. MyList использует в блоке DAT особый синтаксис объявления WORD и создает набор *byte*-выровненных , но *word* -размерных данных в основной памяти. Значения MyList в основной памяти – это соответственно $FF99 и 1000. При побайтном доступе, MyList содержит $99, $FF, 232 и 3, поскольку данные хранятся в формате little-endian.

Эти данные компилируются в объект и в конечное приложение как часть выполнимого кода и доступны для чтения/записи при использовании синтаксиса 3 объявления WORD (см.ниже). Для более детальной информации об использовании WORD в этом синтаксисе, обратитесь к DAT-секции «Объявление Данных (Синтаксис 1)» на стр. 244, и помните, что в поле *Size* в этом описании нужно использовать WORD.

Элементы данных могут повторяться при использовании опционального поля *Count*. Например:

```
DAT
  MyData word 640, $AAAA[4], 5_500
```

В приведенном выше примере объявляется таблица выровненных по размеру слова данных с размером в слово, с именем MyData, состоящей из шести следующих значений: 640, $AAAA, $AAAA, $AAAA, $AAAA, 5500. Величина $AAAA встречается в таблице четыре раза, поскольку в объявлении сразу после нее установлено '[4]'.

Чтение/Запись Word-величин основной памяти (Синтаксис 3)

В блоках PUB и PRI, для чтения или записи величин основной памяти размера *word*, используется синтаксис 3 объявления WORD. Это осуществляется при помощи

выражения, которое обращается к основной памяти, используя формат: word[*BaseAddress*][*Offset*]. Приведем пример.

```
PUB MemTest | Temp
  Temp := word[@MyData][1]              'Read word value
  word[@MyList][0] := Temp + $0123      'Write word value

DAT
  MyData  word  640, $AAAA, 5_500       'Word-sized/aligned data
  MyList  byte  word $FF99, word 1_000  'Byte-sized/aligned word data
```

В нижней части приведенного примера блок DAT размещает свои данные в памяти согласно Рис. 4-5. Первый элемент MyData находится в памяти по адресу $18, последний – по адресу $1C, со следующим сразу за ним первым элементом MyList по адресу $1E. Отметьте, что начальный адрес ($18) произволен и может изменяться компилятором при изменении программы или при включении самого объекта в другое приложение.

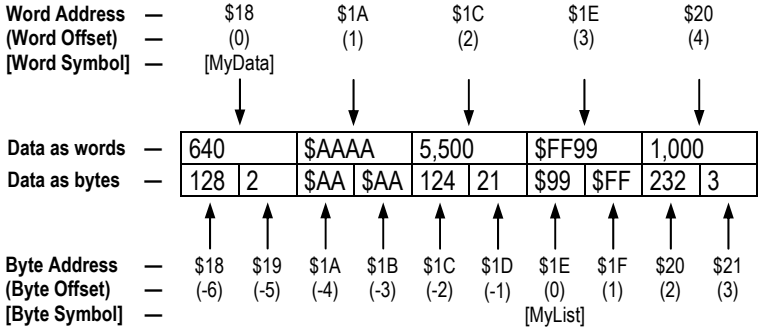


Рис. 4-5: Структура и адресация данных размера *Word* в основной памяти

В верхней части примера, в первой исполнимой строке метода MemTest, Temp := WORD[@MyData][1], производится чтение из основной памяти величины размером *word*. Локальная переменная Temp при этом устанавливается равной $AAAA, то есть значению, прочитанному из основной памяти по адресу $1A. Адрес $1A был определен по адресу идентификатора MyData ($18) плюс смещение в 1 слово (2 байта). Этот процесс показан на упрощенной диаграмме:

word[@MyData][1] → word[$18][1] → word[$18 + (1*2)] → word[$1A]

В следующей строке, `word[@MyList][0] := Temp + $0123`, производится запись в основную память величины размера *word*. При этом данные в основной памяти по адресу $1E устанавливаются в значение $ABCD. Адрес $1E был расчитан по адресу идентификатора `MyList` ($1E) плюс смещение в 0 слов (0 байтов):

word[@MyList][0] → word[$1E][0] → word[$1E + (0*2)] → word[$1E]

Величина $ABCD была получена как текущее значение `Temp` плюс $0123; $AAAA + $0123 равно $ABCD.

Адресация основной памяти

Как отображено на Рис.4-5, основная память представляет собой набор последовательно расположенных байтов (см. строку “data as bytes”), которые при правильном подходе могут быть прочитаны как *word*-значения (2-байтные величины). На самом деле, в приведенном выше примере показано, что адреса также вычисляются в позициях байтов. Такой подход действителен для всех команд, использующих адресацию.

Адресация основной памяти всегда байтовая, независимо от размера переменной, к которой осуществляется доступ, ─ будь она однобайтовая, одинарное либо двойное слово. Это упрощает понимание взаимного расположения байтов, слов и двойных слов, но в то же время вносит некоторые сложности при рассмотрении нескольких элементов одинакового размера, таких как слова.

С этой точки зрения, директива **WORD** имеет очень удобное свойство обеспечивать адрессацию в масштабе *word*. При использовании поля *BaseAddress* совместно с опциональным полем *Offset*, действия производятся в базисе, соответствующем размеру данных.

Представьте себе доступ к словам в памяти с известной начальной точки (*BaseAddress*). Естественно думать, что следующий *word* или *word*-ы будут находиться от этой точки на определенном смещении (*Offset*). Хотя эти слова на самом деле находятся за этой точкой по смещению на определенное количество байт, все же будет проще понять, если говорить о смещении на некоторое количество *word*-ов (то есть: 4^{й} *word*, вместо: *word* который начинается с 6го байта). Идентификатор **WORD** манипулирует данными именно таким образом, умножая величину *Offset* (предоставляемую в *word*-ах), на 2 (количество байт в одном *word*), и прибавляя результат к *BaseAddress* для определения адреса необходимой области памяти для чтения. Он также очищает два младших бита *BaseAddress* для обеспечения выравнивания адреса по размеру двойного слова.

Таким образом, при чтении значений из массива MyData, команда word[@MyData][0] читает первую *word*-величину, word[@MyData][1] читает вторую, а word[@MyData][2] – третью.

Если бы поле *Offset* не использовалось, приведенные выше операторы выглядели бы соответственно как word[@MyData] и word[@MyData+2]. Результат бы оказался таким же, но запись бы не воспринималась так легко, как ранее.

Для дополнительной информации о размещении данных в памяти, см. «Объявление Данных (Синтаксис 1)» блока DAT на стр.244.

Альтернативный доступ к памяти

Существует еще один метод доступа к данным, альтернативный использованному в предыдущем примере; в нем можно обращаться к данным напрямую по имени. К примеру, в следующем выражении производится чтение первого слова массива MyData:

```
Temp := MyData[0]
```

А в этих строках производится чтение второго и третьего слов массива MyData:

```
Temp := MyData[1]
Temp := MyData[2]
```

Так почему бы постоянно не использовать прямое обращение по имени? Рассмотрим следующий случай:

```
Temp := MyList[0]
Temp := MyList[1]
```

Обращаясь к рассмотренному ранее Рис. 4-5, можно ожидать, что эти два оператора прочтут первый и второй *word*-ы массива MyList: соответственно $FF99 и 1000. Однако, напротив, они прочтут лишь первый и второй байты MyList, то есть соответственно $99 и $FF.

Что произошло? В отличие от MyData, идентификатор MyList был задан как байт-размерные и байт-выровненные данные. Хотя данные и состоят из величин *word*, так как каждый элемент предварен директивой **WORD,** но поскольку размерность была задана байтовая, все прямые обращения будут возвращать отдельные байты.

Однако в этом случае для получения значений *word* все же можно использовать указатель **WORD**, поскольку данные, следуя за MyData, получились выровненными по границе слов.

```
Temp := word[@MyList][0]
Temp := word[@MyList][1]
```

Здесь из MyList прочтется первый *word*, $FF99, а затем – второй, 1000. Это свойство очень удобно в случае, когда к одним и тем же данным в одно время необходимо обращаться как к байтам, а в другое – как к словам.

Другие особенности доступа

Оба приведенных метода доступа к памяти, — как директива WORD, так и прямое обращение по имени, — могут использоваться для доступа к любой области основной памяти, независимо от того, как она относится к конкретным структурам данных. Несколько примеров:

```
Temp := word[@MyList][-1]  'Read last word of MyData (before MyList)
Temp := word[@MyData][3]   'Read first word of MyList (after MyData)
Temp := MyList[-6]         'Read first byte of MyData
Temp := MyData[-2]         'Read word that is two words before MyData
```

В этих примерах производится чтение данных за рамками логических границ (начальной и конечной точек) структуры объявленных данных. Это может оказаться и полезным трюком, однако чаще такое происходит по ошибке; будьте внимательны при адресации памяти, особенно когда вы выполняете операции записи.

Доступ к отдельным словам переменных большего размера (Синтаксис 4)

Для записи или чтения отдельных компонентов *long*-переменных в блоках PUB и PRI используется синтаксис 4 объявления WORD. Например:

```
VAR
  long LongVar

PUB Main
  LongVar.word := 65000         'Set first word of LongVar to 65000
  LongVar.word[0] := 65000      'Same as above
  LongVar.word[1] := 1          'Set second word of LongVar to 1
```

В этом примере производится доступ к отдельным словам *long*-переменной LongVar. В комментариях отражено, что делает каждая из строк. В конце метода Main переменная LongVar будет равна 130536.

Аналогичный подход может использоваться при обращении к *word*-составляющим данных с размером в двойное слово.

```
PUB Main | Temp
  Temp := MyList.word[0]          'Read low word of MyList long 0
  Temp := MyList.word[1]          'Read high word of MyList long 0
  MyList.word[1] := $1234         'Write high word of MyList long 0
  MyList.word[2] := $FFEE         'Write low word of MyList long 1

DAT
  MyList  long  $FF998877, $DDDDEEEE  'Long-sized/aligned data
```

В первой и второй строках метода Main производится чтение значений из `MyList`, соответственно $8877 и $FF99. В третьей строке производится запись значения $1234 в старшее слово двойного слова элемента 0 массива `MyList`, изменяющая значение этого элемента на $12348877. В четвертой строке в слово с индексом 2 массива `MyList` (младшее слово двойного слова элемента номер 1 массива `MyList`), записывается $FFEE, изменяя значение элемента на $DDDDFFEE.

WORDFILL

Команда: Заполняет слова в основной памяти заданной величиной .

((PUB ¦ PRI))
 WORDFILL (*StartAddress*, *Value*, *Count*)

- ***StartAddress*** – выражение, указывающее на адрес первого слова памяти для заполнения значением *Value*.
- ***Value*** – выражение, указывающее значение, которым необходимо заполнять слова памяти.
- ***Count*** – выражение, указывающее количество слов для заполнения, начиная с адреса *StartAddress*.

Описание

WORDFILL – это одна из трех команд (BYTEFILL, WORDFILL, и LONGFILL), используемых для заполнения блоков основной памяти заданным значением. Команда LONGFILL заполняет *Count* слов основной памяти значением *Value*, начиная с адреса *StartAddress*.

Использование WORDFILL

WORDFILL – это мощное средство для очистки больших блоков *word* -размерной памяти. Например:

```
VAR
  word Buff[100]

PUB Main
  wordfill(@Buff, 0, 100) 'Clear Buff to 0
```

Первая строка метода Main, очищает весь массив Buff из 100 слов (200 байт), присваивая им ноли. Для таких задач WORDFILL работает быстрее соответствующих циклов REPEAT.

WORDMOVE

Команда: Копирует слова из одной области памяти в другую.

((PUB ¦ PRI))
  WORDMOVE (*DestAddress*, *SrcAddress*, *Count*)

- ***DestAddress*** – выражение, задающее адрес области в основной памяти, куда будет скопирован первый *word* из источника.
- ***SrcAddress*** – выражение, задающее адрес области в основной памяти, где находится первый копируемый *word*.
- ***Count*** – выражение, отображающее количество *word*-ов в области источника для копирования в область приемника

Описание

WORDMOVE - это одна из трех команд (BYTEMOVE, WORDMOVE, и LONGMOVE), используемых для копирования блоков данных основной памяти из одной области в другую. Команда WORDMOVE копирует *Count* слов из области основной памяти, начинающейся с адреса *SrcAddress,* в область основной памяти, начинающуюся с *DestAddress*.

Использование WORDMOVE

WORDMOVE – это мощный способ, используемый для копирования больших блоков *word* - размерной памяти. Например:

```
VAR
  word Buff1[100]
  word Buff2[100]

PUB Main
  wordmove(@Buff2, @Buff1, 100)    'Copy Buff1 to Buff2
```

Первая строка метода Main копирует весь массив Buff1 из 100 *word*-ов (200 байт) в массив Buff2. Для таких задач WORDMOVE работает быстрее соответствующих циклов REPEAT.

_XINFREQ

Константа: Предопределенная, устанавливаемая один раз константа для задания частоты внешнего резонатора.

```
CON
  _XINFREQ = Expression
```

- ***Expression*** – целое выражение, указывающее частоту внешнего резонатора, то есть частоту на пине XI. Это значение используется при начальном пуске приложения.

Описание

_XINFREQ задает значение частоты внешнего резонатора, которое используется совместно со значением режима работы генератора для определения системной частоты при начальной загрузке. Это идентификатор предопределенной константы, значение которой определяется в верхнем объектном файле приложения. Константа **_XINFREQ** либо устанавливается приложением либо напрямую, либо косвенно, как результат установок **_CLKMODE** и **_CLKFREQ**.

Верхний объектный файл приложения (с которого начинается компиляция) может установить **_XINFREQ** в своем блоке **CON**. Это значение, совместно со значением режима работы генератора, определяет частоту, на которую переключится Системный Генератор, как только приложение загрузится и будет запущено на выполнение.

Приложение, в своем блоке **CON,** может задавать либо значение **_XINFREQ,** либо **_CLKFREQ**; эти параметры взаимно-вычисляемые, и незаданное значение одного из них автоматически вычисляется из заданного другого.

В следующих примерах считается, что строки содержатся в верхнем объектном файле. Любые установки **_XINFREQ** в дочерних объектах компилятором игнорируются.

Например:

```
CON
   _CLKMODE = XTAL1 + PLL8X
   _XINFREQ = 4_000_000
```

Первое объявление в приведенном выше блоке **CON** устанавливает режим генератора на работу со внешним низкочастотным резонатором и коэффициентом ФАПЧ, равным 8. Второе объявление указывает, что частота внешнего резонатора равна 4 МГц, что

значит, что частота Системного Генератора будет равна 32 МГц (4 МГц * 8 = 32 МГц). Исходя из этих объявлений, значение _CLKFREQ автоматически устанавливается на 32 МГц.

```
CON
  _CLKMODE = XTAL2
  _XINFREQ = 10_000_000
```

Эти два объявления устанавливают режим генератора на работу со внешним средне-частотным резонатором, без использования цепей ФАПЧ, и частоту внешнего резонатора 10 МГц. Исходя из этих объявлений, значение _CLKFREQ, а, следовательно, и частота Системного Генератора, автоматически также устанавливается на 10 МГц.

Глава 5: Справочник по языку ассемблера

В этой главе описываются все элементы языка ассемблер ИМС Propeller, и ее очень удобно использовать как справочник по каждому конкретному элементу языка. Многие инструкции имеют соответствующие команды в языке *Spin*, поэтому для дополнительной информации рекомендуется обращаться к Справочному пособию по языку *Spin*.

Справочник по языку ассемблера состоит из трех основных частей:

1) **Структура языка Propeller Ассемблер**. Код на языке Propeller Ассемблер является опциональной частью Объектов Propeller. В этой секции описывается общая структура кода на языке ассемблер, и как он располагается внутри объектов.

2) **Перечень элементов языка Propeller *Spin* по категориям**. Все элементы, включая операторы, группируются в соответствии с выполняемыми функциями. Это удобный способ для быстрого осознания возможностей языка и того, какие функции доступны для конкретной задачи. Каждый перечисленный элемент имеет справочную страницу с подробной информацией. Некоторые элементы обозначены индексом “s”, означающим, что они также доступны в *Spin*, хотя их синтаксис может и отличаться. Отмеченные таким образом элементы включены и в Главу 5: Справочник по языку *Spin*.

3) **Элементы языка ассемблера.** Все инструкции перечислены в Главной Таблице в начале секции, и большинство элементов имеет свои собственные подсекции, сортированные по алфавиту для обеспечения их легкого поиска. Те отдельные элементы, которые не имеют подсекций, ─ такие, как Операторы, ─ сгруппированы в рамках других соответствующих подсекций, но могут также быть легко найдены при переходе на страницу со справочной информацией из Перечня по категориям.

Структура ассемблера Propeller

Объекты Propeller состоят из кода, написанного на языке *Spin* и, возможно, ассемблере, а также данных. Код *Spin* формирует структуру объекта, организуя специальные блоки. Данные и код на языке ассемблера (если он имеется), располагаются в специальном блоке DAT (блок данных). См. DAT, стр. 243.

Код на языке *Spin* выполняется *Cog*-ом из Основной Памяти, с использованием интерпретатора *Spin*, в то же время код на ассемблере выполняется в своем

первоначальном виде непосредственно из самого процессора. По этой причине код на языке *Propeller*-ассемблер и любые принадлежащие ему данные должны для выполнения полностью загружаться в процессор. В этом случае и код на ассемблере, и данные рассматриваются в процессе загрузки как одно и то же. В следующем примере показан объект, *Spin*-код которого в методе `Main` блока **PUB** запускает другой процессор для выполнения ассемблерной подпрограммы `Toggle` блока **DAT**.

```
{{ AssemblyToggle.spin }}
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

PUB Main
{Launch Cog to toggle P16 endlessly}
   Cognew(@Toggle, 0) 'Launch new Cog

DAT
{Toggle P16}
                    ORG     0               'Begin at Cog RAM addr 0
Toggle              mov     dira, Pin       'Set Pin to output
                    mov     Time, cnt       'Calculate delay time
                    add     Time, #9        'Set minimum delay here
:loop               waitcnt Time, Delay     'Wait
                    xor     outa, Pin       'Toggle Pin
                    jmp     #:loop          'Loop endlessly

Pin       long      |< 16                   'Pin number
Delay     long      6_000_000               'Clock cycles to delay
Time                res 1                   'System Counter Workspace
```

При выполнении команды **COGNEW** метода `Main`, новый *cog* начинает заполнять своё Cog ОЗУ 496-ю последовательно расположенными *long*-значениями из Основной Памяти, начиная с инструкции по адресу `Toggle`. Затем новый *cog* инициализирует свои Регистры Специальных Функций и начинает выполнение кода, стартуя с регистра 0 в Cog ОЗУ.

Инструкции ассемблера могут перемежаться с данными в рамках этого блока **DAT**, однако необходимо уделять внимание тому, чтобы они были расположены в таком порядке, чтобы все критические элементы загружались для выполнения в *Cog* в правильном порядке. Рекомендуется записывать их в следующем порядке: 1)

ассемблерный код, 2)проинициализированные данные (т.е. LONG), 3) зарезервированные ячейки памяти (т.е.: RES). Это заставит процессор загрузить сначала код ассемблера, сопровождаемый проинициализированными данными и уже затем любые данные приложения, не зависимо от того, требует ли этого сам код. Для более детальной информации см. секции, в которых обсуждается ORG (стр. 488), RES (стр.501), и DAT (стр. 243).

Память Процессора

Память ОЗУ Cog похожа на Основное ОЗУ следующим:

- Каждая может содержать как инструкции, так и данные.
- Каждая может быть изменена при выполнении (напр., переменные в ОЗУ).

ОЗУ Cog отличается от Основной Памяти следующим:

- Память процессора меньше и быстрее Основного ОЗУ.
- Память процессора – это набор регистроов, адресуемых только как *long*-и (4 байта), в то время как Основное ОЗУ – это набор ячеек, адресуемых как байты, слова и двойные слова.
- Ассемблерный код выполняется непосредственно из ОЗУ процессора, в то время как код Spin извлекается и выполняется из Основного ОЗУ.
- ОЗУ процессора доступно только в пределах самого этого *cog*-а, в то время как Основное ОЗУ доступно для всех процессоров.

После того, как в ОЗУ *Cog*-а был загружен ассемблерный код, этот процессор начинает его выполнение, читая код операции (32-битный *long*) из регистра (начиная с регистра 0), определяя свои источники и приемники данных, выполняя инструкцию (возможно, записывая результат в другой регистр) и затем переходя далее по адресу следующего регистра для повторения этого процесса. Регистры ОЗУ *Cog*-а могут содержать как инструкции, так и чистые данные, и все они могут быть изменены в результате выполнения других инструкций.

Где инструкция берет свои данные?

Большинство инструкций включают в себя два операнда – адрес приемника и адрес источника данных. Например, формат инструкции ADD такой:

```
add  destination, {#}source
```

Операнд *destination* – это 9-битный адрес регистра, содержащего необходимую величину, с которой будет производиться операция. Операнд *source* – это либо 9-

битная константа, либо 9-битный адрес регистра, содержащего необходимую величину. Значение операнда *source* зависит от того, был ли использован символ константы "#". Например:

```
   add  X, #25           'Add 25 to X
   add  X, Y             'Add Y to X

X  long  50
Y  long  10
```

Первая инструкция прибавляет константу 25 к значению, содержащемуся в регистре X. Во второй же производится сложение величины из регистра Y с величиной из X. В обоих случаях результат операции сохраняется назад в регистр X.

Последние две строки определяют идентификаторы данных X и Y как *long*-величины, соответственно 50 и 10. Поскольку запуск ассемблерного кода в процессоре привел к записи этих данных в ОЗУ *Cog* сразу за инструкциями, то естественно, что X – это идентификатор, указывающий на регистр со значением 50, а Y – указывающий на регистр со значением 10.

Таким образом, результат выполнения первой инструкции ADD равен 75 (т.е.: X + 25 → 50 + 25 = 75) и это значение, 75, сохранено назад в регистр X. Аналогично, результат выполнения второй инструкции ADD равен 85 (т.е.: X + Y → 75 + 10 = 85), так что в конце X установлен в 85.

Не забывайте символ константы '#'

Не забывайте вводить символ '#', если приведенное в инструкции значение используется как константа. Изменение первой строки в приведенном выше примере путем исключения символа '#' (т.е.: ADD X, 25) приведет к тому, что величина из регистра 25 будет добавлена к X, вместо того, чтобы к X добавить число 25.

Другая возможная ошибка возникает при игнорировании символа '#' в инструкциях перехода, таких как JMP или DJNZ. Если необходимое место перехода отмечено меткой MyRoutine, то инструкция JMP должна выглядеть как JMP #MyRoutine, а не JMP MyRoutine. Последнее приведет к использованию сохраненной в регистре MyRoutine величины как адреса перехода; это удобно для выполнения косвенных переходов, но обычно это не то, что задумывал разработчик.

Константы должны умещаться в 9-ти битах

Операнд-источник имеет длину всего 9 битов; он может содержать значения от 0 до 511 (от $000 до $1FF). Имейте это в виду при задании констант. Если величина слишком большая, чтобы уместиться в 9 битах, она должна быть помещена в регистр и доступ к ней осуществлен по адресу этого регистра. Например:

```
   add  X, BigValue      'Add BigValue to X

X         long  50
BigValue  long  1024
```

Глобальные и локальные метки

Для наименования отдельных процедур, *Propeller*-ассемблер может использовать два вида меток: глобальные и локальные.

Глобальные метки выглядят точно так же, как и остальные идентификаторы и подчиняются тем же правилам; они начинаются с подчеркивания '_' либо буквы и сопровождаются другими буквами, подчеркиваниями и/или цифрами. Для дополнительной информации см. «Правила Идентификаторов», стр.183.

Локальные метки похожи на глобальные, за исключением того, что они начинаются с двоеточия ':' и должны быть отделены от других одноименных локальных меток как минимум одной глобальной меткой. Например:

```
Addition            mov     Count, #9       'Set up 'Add' loop
counter
:loop               add     Temp, X         'Iteratively do Temp+X
                    djnz    Count, #:loop   'Dec counter, loop back

Subtraction         mov     Count, #15      'Set up 'Sub' loop
counter
:loop               sub     Temp, Y         'Iteratively do Temp-Y
                    djnz    Count, #:loop   'Dec counter, loop back

                    jmp     #Addition       'Go add more
```

В этом коде присутствует две глобальные метки: `Addition` и `Subtraction`, а также две локальные, обе с именем `:loop`. Эти локальные метки могут иметь абсолютно одинаковые имена, `:loop`, поскольку их разделяет как минимум одна глобальная метка. На самом деле, в этом преимущество локальных меток: они указывают на

общие, характерные вещи, такие как циклы, не требуя уникального имени для каждой из них.

Две инструкции **DJNZ** абсолютно одинаковы, однако каждая из них осуществляет переход в различные места. Инструкция **DJNZ** подпрограммы `Addition` переходит назад на локальную метку `:loop` в рамках `Addition`, а инструкция **DJNZ** подпрограммы `Subtraction` — на локальную метку `:loop` в рамках `Subtraction`.

Отметьте, что инструкции **DJNZ** используют символ константы '#', и точное имя локальной метки, включая двоеточие. Без символа '#' этот код выполнялся бы неверно (выполнялся бы косвенный переход вместо прямого), а без двоеточия при компиляции была бы выдана ошибка.

Для детальной информации о формате *Propeller*-ассемблера, см. «Общие элементы синтаксиса», стр. 408.

Перечень элементов Propeller ассемблер по категориям

Директивы

ORG	Установить при компиляции указатель адреса в *Cog*-е; стр.488.
FIT	Проверить, что предыдущие инструкции/данные полностью помещаются в *Cog*; стр. 453.
RES	Резервировать следующий *long*(и) для идентификатора; стр. 499.

Конфигурация

CLKSET[s]	Установить режим генератора при выполнении; стр. 429.

Управление процессором

COGID[s]	Получить *ID*-номер текущего процессора; стр. 441.
COGINIT[s]	Запустить, или перезапустить *Cog* по его *ID*; стр. 442.
COGSTOP[s]	Остановить *Cog* по его *ID*; стр. 444.

Управление процессами

LOCKNEW[s]	Проверить новый запрет; стр. 465.
LOCKRET[s]	Отменить запрет; стр. 466.
LOCKCLR[s]	Снять запрет по *ID*; стр. 463.
LOCKSET[s]	Установить запрет по *ID*; стр. 272.
WAITCNT[s]	Ожидать пока Системный Счетчик достигнет значения; стр. 373.
WAITPEQ[s]	Ожидать, пока вывод(ы) станут равны значению; стр. 377.
WAITPNE[s]	Ожидать, пока вывод(ы) перестанут равны значению; стр. 379.
WAITVID[s]	Ожидать видео и освободить следующие цвет/пиксель; стр. 380

Условные операторы

IF_ALWAYS	Всегда; стр. 456.
IF_NEVER	Никогда; стр. 456.
IF_E	Если равно (Z = 1); стр. 456.
IF_NE	Если не равно (Z = 0); стр. 456.

IF_A	Если больше (!C & !Z = 1); стр. 456.
IF_B	Если меньше (C = 1); стр. 456.
IF_AE	Больше либо равно (C = 0); стр. 456.
IF_BE	Меньше либо равно (C | Z = 1); стр. 456.
IF_C	Если C установлен; стр. 456.
IF_NC	Если C сброшен; стр. 456.
IF_Z	Если Z установлен; стр. 456.
IF_NZ	Если Z сброшен; стр. 456.
IF_C_EQ_Z	Если C равен Z; стр. 456.
IF_C_NE_Z	Если C не равен Z; стр. 456.
IF_C_AND_Z	Если C установлен и Z установлен; стр. 456.
IF_C_AND_NZ	Если C установлен и Z сброшен; стр. 456.
IF_NC_AND_Z	Если C сброшен и Z установлен; стр. 456.
IF_NC_AND_NZ	Если C сброшен и Z сброшен; стр. 456.
IF_C_OR_Z	Если C установлен или Z установлен; стр. 456.
IF_C_OR_NZ	Если C установлен или Z сброшен; стр. 456.
IF_NC_OR_Z	Если C сброшен или Z установлен; стр. 456.
IF_NC_OR_NZ	Если C сброшен или Z сброшен; стр. 456.
IF_Z_EQ_C	Если Z равен C; стр. 456.
IF_Z_NE_C	Если Z не равен C; стр. 456.
IF_Z_AND_C	Если Z установлен и C установлен; стр. 456.
IF_Z_AND_NC	Если Z установлен и C сброшен; стр. 456.
IF_NZ_AND_C	Если Z сброшен и C установлен; стр. 456.
IF_NZ_AND_NC	Если Z сброшен и C сброшен; стр. 456.
IF_Z_OR_C	Если Z установлен или C установлен; стр. 456.
IF_Z_OR_NC	Если Z установлен или C сброшен; стр. 456.
IF_NZ_OR_C	Если Z сброшен или C установлен; стр. 456.
IF_NZ_OR_NC	Если Z сброшен или C сброшен; стр. 456.

Управление потоком

CALL	Переход по адресу с дальнейшим возвратом на следующую инструкцию; стр. 427.
DJNZ	Декремент значения и переход по адресу, если не ноль; стр. 447.
JMP	Безусловный переход по адресу; стр. 458.
JMPRET	Переход по адресу с дальнейшей возможностью “возврата” по другому адресу; стр. 459.
TJNZ	проверить значение и перейти по адресу, если не ноль; стр. 523.
TJZ	Проверить значение и перейти по адресу, если ноль; стр. 526.
RET	Возврат по сохраненному адресу; стр. 504.

Воздействия

NR	Нет результата (не записывать результат); стр. 450.
WR	Записать результат; стр. 450.
WC	Записать статус C; стр. 450.
WZ	Записать статус Z; стр. 450.

Доступ к Основной Памяти

RDBYTE	Прочитать байт основной памяти; стр. 495.
RDWORD	Прочитать слово основной памяти; стр. 498.
RDLONG	Прочитать двойное слово основной памяти; стр. 496.
WRBYTE	Записать байт в основную память; стр. 534.
WRWORD	Записать слово в основную память; стр. 535.
WRLONG	Записать двойное слово в основную память; стр. 535.

Общие операции

ABS	Получить абсолютное значение; стр. 416.
ABSNEG	Получить отрицательное от абсолютного значения числа; стр. 417.
NEG	Получить инвертированное значение числа; стр. 479.
NEGC	Получить значение или аддитивное инверсное в зависимости от C; с. 480.
NEGNC	Получить значение или аддитивное инверсное в зависимости от!C; с. 481.

NEGZ	Получить значение или аддитивное инверсное в зависимости от Z; стр. 483.
NEGNZ	Получить значение или аддитивное инверсное в зависимости от !Z; стр. 482.
MIN	Ограничение по минимуму величины без знака к другой без знака; стр. 470.
MINS	Ограничение по минимуму величины со знаком к другой со знаком; с. 470.
MAX	Ограничение по максимуму величины без знака к другой величине без знака; стр. 468.
MAXS	Ограничение по максимуму величины со знаком к другой величине со знаком; стр. 469.
ADD	Сложение двух величин без знака; стр. 417.
ADDABS	Сложение абсолютного значения величины с другой величиной; стр. 418.
ADDS	Сложение двух величин со знаком; стр. 421.
ADDX	Сложение двух величин без знака плюс C; стр. 423.
ADDSX	Сложение двух величин со знаком плюс C; стр. 421.
SUB	Вычитание двух величин без знака; стр. 510.
SUBABS	Вычитание абсолютного значения величины из другого значения; стр. 512.
SUBS	Вычитание двух величин со знаком; стр. 513.
SUBX	Вычитание величины без знака плюс C из другой величины без знака; с. 516.
SUBSX	Вычитание величины со знаком плюс C из другой величины со знаком; с. 514.
SUMC	Сумма величины со знаком с другой с влиянием C на знак; стр. 517.
SUMNC	Сумма величины со знаком с другой с влиянием !C на знак; стр. 519.
SUMZ	Сумма величины со знаком с другой с влиянием Z на знак; стр. 521.
SUMNZ	Сумма величины со знаком с другой с влиянием !Z на знак; стр. 520.
MUL	<зарезервировано для использования в будущем>
MULS	< зарезервировано для использования в будущем >
AND	Побитовое И двух величин; стр. 425.
ANDN	Побитовое И величины с инверсным (НЕ) значением другой; стр. 426.
OR	Побитовое ИЛИ двух величин; стр. 488.
XOR	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ двух величин; стр. 537.
ONES	< зарезервировано для использования в будущем >
ENC	< зарезервировано для использования в будущем >
RCL	Циклический сдвиг C влево на заданное количество битов; стр. 494.

RCR	Циклический сдвиг C вправо на заданное количество битов; стр. 494.
REV	Реверс LSB значения и дополнение нолями; стр. 504.
ROL	Циклический сдвиг значения влево на заданное количество бит; стр. 505.
ROR	Циклический сдвиг значения вправо на заданное количество бит; стр. 506.
SHL	Сдвиг значения влево на заданное количество битов; стр. 509.
SHR	Сдвиг значения вправо на заданное количество битов; стр. 509.
SAR	Арифметический сдвиг вправо на заданное количество бит; стр. 507.
CMP	Сравнение двух величин без знака; стр. 431.
CMPS	Сравнение двух величин со знаком; стр. 432.
CMPX	Сравнение двух величин без знака плюс C; стр. 438.
CMPSX	Сравнение двух величин со знаком плюс C; стр. 434.
CMPSUB	Сравнение величин без знака, вычитание второй, если меньше либо равна; стр. 433.
TEST	Побитовое И двух величин с влиянием лишь на флаги; стр. 523.
TESTN	Побитовое И величины с NOT от другой; влияет лишь на флаги; р 460.
MOV	Поместить величину в регистр; стр. 471.
MOVS	Установить поле источник регистра равным значению; стр. 473.
MOVD	Установить поле приемник регистра равным значению; стр. 472.
MOVI	Установить поле инструкции регистра равным значению; стр. 472.
MUXC	Установить дискретные биты величины в состояние флага C; стр. 475.
MUXNC	Установить дискретные биты величины в состояние флага !C; стр. 476.
MUXZ	Установить дискретные биты величины в состояние флага Z; стр. 478.
MUXNZ	Установить дискретные биты величины в состояние флага !Z; стр. 477.
HUBOP	Выполнить *Hub*-функцию; стр. 454.
NOP	Нет операции, простой на четыре такта; стр. 483.

Регистры

DIRA[s]	Регистр направления 32-битного порта A; стр. 499.
DIRB[s]	Регистр направления 32-битного порта B (будущее); стр. 499.
INA[s]	Входной регистр 32-битного порта A (только чтение); стр. 499.
INB[s]	Входной регистр 32-битного порта B (чтение) (будущее); стр. 499.

OUTA[s]	Выходной регистр 32-битного порта A; стр. 499.
OUTB[s]	Выходной регистр 32-битного порта B (будущее); стр. 499.
CNT[s]	32-битный регистр системного счетчика (только чтение); стр. 499.
CTRA[s]	Регистр управления счетчика A; стр. 499.
CTRB[s]	Регистр управления счетчика B; стр. 499.
FRQA[s]	Регистр частоты счетчика A; стр. 499.
FRQB[s]	Регистр частоты счетчика B; стр. 499.
PHSA[s]	Регистр ФАПЧ счетчика A; стр. 499.
PHSB[s]	Регистр ФАПЧ счетчика B; стр. 499.
VCFG[s]	Регистр конфигурации видео; стр. 499.
VSCL[s]	Регистр масштаба видео; стр. 499.
PAR[s]	Регистр параметра начальной загрузки cog (только чтение); с. 499.

Константы

ПРИМЕЧАНИЕ: Обратитесь к секции «Предопределенные Константы», Глава 4: Справочник по языку Spin .

TRUE[s]	Логическая ИСТИНА: -1 ($FFFFFFFF); стр. 237.
FALSE[s]	Логическая ЛОЖЬ: 0 ($00000000); стр. 237.
POSX[s]	Максимальное положительное целое: 2147483647 ($7FFFFFFF); стр. 238.
NEGX[s]	Максимальное отрицательное целое: -2147483648 ($80000000); стр. 238.
PI[s]	Вещественное значение Пи PI: ~3.141593 ($40490FDB); стр. 238.

Унарные операторы

ПРИМЕЧАНИЕ: Приведенные операторы используются в выражениях-константах.

+	Положительное (+X) – унарная форма Сложения; стр. 487.
-	Отрицательное (−X) – унарная форма Вычитания; стр. 487.
^^	Квадратный корень; стр. 487.
||	Абсолютное значение; стр. 487.
|<	Дешифровать величину (0-31) в двойное слово с одним старшим битом; стр. 487.

>|	Шифровать *long* в (0 … 32) с приоритетом старшего бита; стр. 487.
!	Побитовое НЕ; стр. 487.
@	Адрес идентификатора; стр. 487.

Бинарные операторы

ПРИМЕЧАНИЕ: Приведенные операторы используются в выражениях-константах.

+	Сложение; стр. 487.
-	Вычитание; стр. 487.
*	Умножить и вернуть младшие 32 бита (знаковое); стр. 487.
**	Умножить и вернуть старшие 32 бита (знаковое); стр. 487.
/	Разделить и вернуть целое (знаковое); стр. 487.
//	Разделить и вернуть остаток (знаковое); стр. 487.
#>	Ограничение минимума (знаковое); стр. 487.
<#	Ограничение максимума (знаковое); стр. 487.
~>	Арифметический сдвиг вправо; стр. 487.
<<	Побитовое: Сдвиг влево; стр. 487.
>>	Побитовое: Сдвиг вправо; стр. 487.
<-	Побитовое: Циклический сдвиг влево; стр. 487.
->	Побитовое: Циклический сдвиг вправо; стр. 487.
><	Побитовое: Реверсирование; стр. 487.
&	Побитовое: И; стр. 487.
|	Побитовое: ИЛИ; стр. 487.
^	Побитовое: ИСКЛЮЧАЮЩЕЕ ИЛИ; стр. 487.
AND	Логическое: И (представляет не-0 как -1); стр. 487.
OR	Логическое: OR (представляет не-0 как -1); стр. 487.
==	Логическое: Равно; стр. 487.
<>	Логическое: Не равно; стр. 487.
<	Логическое: Меньше чем (знаковое); стр. 487.
>	Логическое: Больше чем (знаковое); стр. 487.
=<	Логическое: Меньше или равно (знаковое); стр. 487.
=>	Логическое: Больше или равно (знаковое); стр. 487.

Элементы языка ассемблер

Определения синтаксиса

Вдобавок к подробному описанию, дальнейшие страницы содержат определения синтаксиса для многих элементов языка, описывающие в краткой форме все возможные варианты использования каждого конкретного элемента. В определениях синтаксиса используются специальные символы, указывающие когда и как каждый конкретный элемент должен использоваться.

`BOLDCAPS`	Элементы, указанные заглавными символами утолщенного шрифта должны вводиться точно так, как указано.
Bold Italics	Элементы, отображаемые утолщенным курсивом, заменяются текстом пользователя: идентификаторами, операторами, выражениями и т.д.
`. : , #`	Точки, двоеточия, запятые и символы решетки должны вводиться там, где они указаны.
〈 〉	Угловые скобки заключают опциональные элементы. Вводите заключенные элементы при необходимости. Сами скобки не вводите.
Двойная линия	Отделяет инструкции от значения результата.

Общие элементы синтаксиса

При чтении определений синтаксиса в этой главе помните, что инструкции ассемблера *Propeller* имеют три общих, опциональных элемента: метка, условие и воздействия. Каждая инструкция языка *Propeller*-ассемблер имеет следующий базовый синтаксис:

〈***Метка***〉 〈***Условие***〉 ***Инструкция Операнды*** 〈***Воздействия***〉

- ***Метка*** – опциональная метка. *Метка* может быть глобальной (начинаясь с подчеркивания ‘_’ или буквы), либо может быть локальной (начинаясь с двоеточия ‘:’). Локальные метки должны быть отделены от других одноименных локальных меток как минимум одной глобальной меткой. *Метка* используется такими инструкциями, как `JMP`, `CALL` и `COGINIT` для обозначения места перехода. Для дополнительной информации см. «Глобальные и локальные метки» на стр.Глобальные и локальные метки 398.

- ***Условие*** – это опциональное условие выполнения (IF_C, IF_Z, и т.д.), которое приводит (либо нет) к выполнению инструкции. См. «Условия (IF_x)» на стр. 446 для более детальной информации.
- ***Инструкция*** и ***Операнды*** это инструкция языка *Propeller*-ассемблер (MOV, ADD, COGINIT, и т.д.) и ее операнды, которых может быть ноль, один или два, что определяется самой инструкцией.
- ***Воздействия*** – это опциональный перечень от одного до трех воздействий выполнения (WZ, WC, WR, и NR) для применения к инструкции при ее выполнении. Они приводят к изменению ***Инструкцией*** флагов Z, C, и, соответственно, к записи или нет результата выполнения инструкции в регистр приемника. См. «Воздействия» на стр. 450 для более детальной информации.

Поскольку каждая инструкция может включать все три опциональных поля (***Метка***, ***Условие***, и ***Воздействия***), для упрощения эти общие поля преднамеренно опущены из описания синтаксиса инструкции.

Поэтому, если Вы читаете описание синтаксиса, подобное такому:

WAITCNT ***Target***, ⟨#⟩ ***Delta***

...помните, что на самом деле синтаксис такой:

⟨***Метка***⟩ ⟨***Условие***⟩ **WAITCNT** ***Target***, ⟨#⟩ ***Delta*** ⟨***Воздействия***⟩

Это правило применяется только для инструкций *Propeller*-ассемблера; оно не применяется для директив этого языка.

В определениях синтаксиса операндам всегда даются описательные имена, такие как *Target* и *Delta* для инструкции WAITCNT, в приведенном выше примере. Эти имена используются в детальных описаниях синтаксиса, однако в таблицах кодов операций и таблицах истинности используются их собирательные названия (D, ПРМ, Приемник, и S, ИСТ, Источник) для указания на отдельные биты инструкции, в которых содержатся соответствующие величины.

Коды операций и таблицы кодов операций

Большинство описаний синтаксиса инструкций содержит таблицу кода операции, похожую на приведенную ниже. В этой таблице приведен 32-х битный код операции (opcode) данной инструкции, результаты её выполнения, а также количество затрачиваемых на выполнение тактов.

В первой колонке таблицы содержится код операции для данной инструкции Propeller-ассемблера, состоящий из следующих полей:

ИНСТР (биты 31:26) - определяет выполняемую инструкцию.

ZCRI (биты 25:22) - определяет статус воздействий и значение поля SRC.

УСЛ (биты 21:18) - определяет условие выполнения инструкции.

ПРМ (биты 17:9) - содержит адрес регистра приемника.

ИСТ (биты 8:0) - содержит адрес источника, либо 9-бит константу.

Каждый бит поля **ZCRI** равен 1 либо 0 для указания, должен ли быть записан флаг '**Z**', флаг '**C**' и результат '**R**'esult, а также содержит ли поле **ИСТ** константу (вместо адреса регистра). Биты **Z** и **C** поля **ZCRI** по умолчанию сброшены (0), и устанавливаются (1), если в инструкции было указано воздействие **WZ** и/или **WC**. См. «Воздействия» на стр. 450. Состояние по умолчанию бита **R** зависит от типа инструкции, и изменяется, если инструкция была введена с воздействием **WR** или **NR**. Состояние по умолчанию бита **I** зависит от типа инструкции, и изменяется, если инструкция была введена с символом константы (**#**) в поле источника.

Биты поля **CON** обычно по умолчанию равны всем единицам (1111), но изменяются, если инструкция вводилась с условием. См. «Условия (IF_x)» на стр. 446.

Последние четыре колонки таблицы кода операции показывают значение выходных флагов **Z** и **C**, будет ли произведена запись величины результата, а также количество тактов, необходимых для выполнения инструкции.

Таблица кода операции инструкции CLKSET:

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0001 1111 ddddddddd ------000	---	---	Не записан	7..22

Краткие таблицы истинности

После таблицы кода операции идет краткая таблица истинности. Эта таблица отображает пробные входные и результирующие выходные воздействия для соответствующей инструкции. Вместо того, чтобы показывать все возможные варианты входов/выходов, в таблице сосредотачивается внимание на использовании инструкций на их числовых или логических границах, что приводит к соответствующим изменениям в состоянии флагов и значимом изменении выхода. Эта информация может быть полезной при изучении либо проверке соответствующего функционирования и поведения инструкций.

В большинстве случаев, краткие таблицы истинности должны быть внимательно прочитаны с первой по последнюю строку. При возможном возникновении нескольких граничных условий, соответствующие строки группируются вместе, для акцентирования внимания, и отделяются от остальных жирной горизонтальной линией.

Действуют следующие соглашения:

$FFFF_FFFE; -2	Числа представляют собой шестнадцатиричные (слева от ';') и десятичные (справа от ';') величины.
%0_00000011; 3	Числа представляют собой двоичные (слева от ';') и десятичные (справа от ';') величины.
0 –или– 1	Отдельный ноль либо единица обозначают двоичный 0 или 1.
wr, wz, wc	Воздействия обозначают состояние выполнения: запись-r(результата), запись-z, запись-c.
x	Малая литера "x" обозначает, что возможны любые значения.
---	Дефисы означают невозможные либо неважные пункты.

Хороший пример – таблица истинности для инструкции **ADDS**, приведенная ниже.

Таблица истинности инструкции ADDS:

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$FFFF_FFFF; -1	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	–	–	wz wc	$FFFF_FFFF; -1	0	0
$7FFF_FFFE; 2147483646	$0000_0001; 1	–	–	wz wc	$7FFF_FFFF; 2147483647	0	0
$7FFF_FFFE; 2147483646	$0000_0002; 2	–	–	wz wc	$8000_0000; -2147483648	0	1
$8000_0001; -2147483647	$FFFF_FFFF; -1	–	–	wz wc	$8000_0000; -2147483648	0	0
$8000_0001; -2147483647	$FFFF_FFFE; -2	–	–	wz wc	$7FFF_FFFF; 2147483647	0	1

В таблице истинности для инструкции **ADDS** имеется восемь строк данных, сгруппированых в четыре пары. Каждая группа испытывает различные граничные условия. Первые пять столбцов каждой строки отображают входные, а последние три – выходные величины.

- В первой паре строк показаны соответственно простое сложение со знаком (-1 + 1), которое дает в результате ноль (установка флага **Z**), и другой пример, (-1 + 2) который приводит к ненулевому результату (очистка флага **Z**).
- Во второй паре строк показана та же ситуация, но с инвертированными знаками значений: (1 + -1) и (1 + -2).

- В третьей паре строк рассматривается случай сложения вблизи верхней границы знакового целого (2,147,482,646 + 1), и следующий за ним пример перехода через эту границу (2,147,482,646 + 2), приводящий к знаковому переполнению (установке флага C).
- В четвертой паре строк рассмотрена похожая ситуация, но при подходе к и пересечении границы с отрицательной стороны, также приводящей к знаковому переполнению (установке флага C).

Отметьте, что на самом деле поле приемника в инструкции содержит адрес регистра, содержащего величину необходимого операнда, и поле источника часто рассматривается аналогичным образом, однако в таблицах истинности эти детали всегда упрощаются и отображаются непосредственно сами величины в источнике и приемнике.

Сводная таблица инструкций языка Propeller ассемблер

На следующих страницах приведена сводная таблица инструкций языка Propeller ассемблер. В этой таблице литеры D и S указывают на поля инструкции «приемник» и «источник», также известные как соответственно d- и s-поле. Для записей, помеченных звездочками в колонке Циклы, необходимо прочесть Примечания к Сводной Таблице на стр. 415.

Инструкция	-ИНСТР- ZCRI -УСЛ- -ПРМ- -ИСТ-	Результат Z	Результат C	Результат	Тактов
ABS D, S	101010 001i 1111 ddddddddd sssssssss	Результат = 0	S[31]	Записан	4
ABSNEG D, S	101011 001i 1111 ddddddddd sssssssss	Результат = 0	S[31]	Записан	4
ADD D, S	100000 001i 1111 ddddddddd sssssssss	D + S = 0	Беззнаков. перенос	Записан	4
ADDABS D, S	100010 001i 1111 ddddddddd sssssssss	D + |S| = 0	Беззнаков. перенос[3]	Записан	4
ADDS D, S	110100 001i 1111 ddddddddd sssssssss	D + S = 0	Знаковый переполн.	Записан	4
ADDSX D, S	110110 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Знаковый переполн.	Записан	4
ADDX D, S	110010 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Беззнаков. перенос	Записан	4
AND D, S	011000 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4
ANDN D, S	011001 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4
CALL #S	010111 0011 1111 ????????? sssssssss	Результат = 0	---	Записан	4
CLKSET D	000011 0001 1111 ddddddddd ------000	---	---	Не записан	7..22[1]
CMP D, S	100001 000i 1111 ddddddddd sssssssss	D = S	Беззнаковый (D<S)	Не записан	4
CMPS D, S	110000 000i 1111 ddddddddd sssssssss	D = S	Знаковый (D<S)	Не записан	4
CMPSUB D, S	111000 000i 1111 ddddddddd sssssssss	D = S	Беззнаков. (D => S)	Не записан	4
CMPSX D, S	110001 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Знаков. (D<S+C)	Не записан	4
CMPX D, S	110011 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Беззнаков. (D<S+C)	Не записан	4
COGID D	000011 0011 1111 ddddddddd ------001	ID = 0	0	Записан	7..22[1]
COGINIT D	000011 0001 1111 ddddddddd ------010	ID = 0	Нет свободн. *COG*	Не записан	7..22[1]
COGSTOP D	000011 0001 1111 ddddddddd ------011	Остановлен ID = 0	Нет свободн. *COG*	Не записан	7..22[1]
DJNZ D, S	111001 001i 1111 ddddddddd sssssssss	Результат = 0	Беззнаковый заем	Записан	4 или 8[2]
HUBOP D, S	000011 000i 1111 ddddddddd sssssssss	Результат = 0	---	Не записан	7..22[1]
JMP S	010111 000i 1111 --------- sssssssss	Результат = 0	---	Не записан	4
JMPRET D, S	010111 001i 1111 ddddddddd sssssssss	Результат = 0	---	Записан	4
LOCKCLR D	000011 0001 1111 ddddddddd ------111	ID = 0	Предыд. сост. Lock	Не записан	7..22[1]
LOCKNEW D	000011 0011 1111 ddddddddd ------100	ID = 0	Нет своб. Lock	Записан	7..22[1]
LOCKRET D	000011 0001 1111 ddddddddd ------101	ID = 0	Нет своб. Lock	Не записан	7..22[1]
LOCKSET D	000011 0001 1111 ddddddddd ------110	ID = 0	Предыд. сост. Lock	Не записан	7..22[1]
MAX D, S	010011 001i 1111 ddddddddd sssssssss	S = 0	Беззнаковый (D < S)	Записан	4
MAXS D, S	010001 001i 1111 ddddddddd sssssssss	S = 0	Знаковый (D < S)	Записан	4
MIN D, S	010010 001i 1111 ddddddddd sssssssss	S = 0	Беззнаковый (D < S)	Записан	4
MINS D, S	010000 001i 1111 ddddddddd sssssssss	S = 0	Знаковый (D < S)	Записан	4
MOV D, S	101000 001i 1111 ddddddddd sssssssss	Результат = 0	S[31]	Записан	4
MOVD D, S	010101 001i 1111 ddddddddd sssssssss	Результат = 0	---	Записан	4
MOVI D, S	010110 001i 1111 ddddddddd sssssssss	Результат = 0	---	Записан	4
MOVS D, S	010100 001i 1111 ddddddddd sssssssss	Результат = 0	---	Записан	4
MUXC D, S	011100 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4
MUXNC D, S	011101 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4
MUXNZ D, S	011111 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4

5: Справочник по языку ассемблер: сводная таблица

Инструкция	-ИНСТР- ZCRI	-УСЛ-	-ПРМ-	-ИСТ-	Результат Z	Результат C	Результат	Тактов
MUXZ D, S	011110 001i	1111	dddddddddd	ssssssssss	Результат = 0	Четность результата	Записан	4
NEG D, S	101001 001i	1111	dddddddddd	ssssssssss	Результат = 0	S[31]	Записан	4
NEGC D, S	101100 001i	1111	dddddddddd	ssssssssss	Результат = 0	S[31]	Записан	4
NEGNC D, S	101101 001i	1111	dddddddddd	ssssssssss	Результат = 0	S[31]	Записан	4
NEGNZ D, S	101111 001i	1111	dddddddddd	ssssssssss	Результат = 0	S[31]	Записан	4
NEGZ D, S	101110 001i	1111	dddddddddd	ssssssssss	Результат = 0	S[31]	Записан	4
NOP	------ ----	0000	---------	---------	---	---	---	4
OR D, S	011010 001i	1111	dddddddddd	ssssssssss	Результат = 0	Четность результата	Записан	4
RCL D, S	001101 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[31]	Записан	4
RCR D, S	001100 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[0]	Записан	4
RDBYTE D, S	000000 001i	1111	dddddddddd	ssssssssss	Результат = 0	---	Записан	7..22 [1]
RDLONG D, S	000010 001i	1111	dddddddddd	ssssssssss	Результат = 0	---	Записан	7..22 [1]
RDWORD D, S	000001 001i	1111	dddddddddd	ssssssssss	Результат = 0	---	Записан	7..22 [1]
RET	010111 0001	1111	---------	---------	Результат = 0	---	Не записан	4
REV D, S	001111 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[0]	Записан	4
ROL D, S	001001 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[31]	Записан	4
ROR D, S	001000 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[0]	Записан	4
SAR D, S	001110 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[0]	Записан	4
SHL D, S	001011 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[31]	Записан	4
SHR D, S	001010 001i	1111	dddddddddd	ssssssssss	Результат = 0	D[0]	Записан	4
SUB D, S	100001 001i	1111	dddddddddd	ssssssssss	D – S = 0	Беззнаковый заем	Записан	4
SUBABS D, S	100011 001i	1111	dddddddddd	ssssssssss	D – |S| = 0	Беззнаковый заем [4]	Записан	4
SUBS D, S	110101 001i	1111	dddddddddd	ssssssssss	D – S = 0	Знаковый переполн.	Записан	4
SUBSX D, S	110111 001i	1111	dddddddddd	ssssssssss	Z & (D-(S+C) = 0)	Знаковый переполн.	Записан	4
SUBX D, S	110011 001i	1111	dddddddddd	ssssssssss	Z & (D-(S+C) = 0)	Беззнаковый заем	Записан	4
SUMC D, S	100100 001i	1111	dddddddddd	ssssssssss	D ± S = 0	Знаковый переполн.	Записан	4
SUMNC D, S	100101 001i	1111	dddddddddd	ssssssssss	D ± S = 0	Знаковый переполн.	Записан	4
SUMNZ D, S	100111 001i	1111	dddddddddd	ssssssssss	D ± S = 0	Знаковый переполн.	Записан	4
SUMZ D, S	100110 001i	1111	dddddddddd	ssssssssss	D ± S = 0	Знаковый переполн.	Записан	4
TEST D, S	011000 000i	1111	dddddddddd	ssssssssss	D = 0	Четность результата	Не записан	4
TESTN D, S	011001 000i	1111	dddddddddd	ssssssssss	D = 0	Четность результата	Не записан	4
TJNZ D, S	111010 000i	1111	dddddddddd	ssssssssss	D = 0	0	Не записан	4 или 8 [2]
TJZ D, S	111011 000i	1111	dddddddddd	ssssssssss	D = 0	0	Не записан	4 или 8 [2]
WAITCNT D, S	111110 001i	1111	dddddddddd	ssssssssss	Результат = 0	Беззнаков. перенос	Записан	5+
WAITPEQ D, S	111100 000i	1111	dddddddddd	ssssssssss	Результат = 0	---	Не записан	5+
WAITPNE D, S	111101 000i	1111	dddddddddd	ssssssssss	Результат = 0	---	Не записан	5+
WAITVID D, S	111111 000i	1111	dddddddddd	ssssssssss	Результат = 0	---	Не записан	5+
WRBYTE D, S	000000 000i	1111	dddddddddd	ssssssssss	---	---	Не записан	7..22 [1]

Инструкция	-ИНСТР- ZCRI -УСЛ- -ПРМ- -ИСТ-	Результат Z	Результат C	Результат	Тактов
WRLONG D, S	000010 000i 1111 ddddddddd sssssssss	---	---	Не записан	7..22[1]
WRWORD D, S	000001 000i 1111 ddddddddd sssssssss	---	---	Не записан	7..22[1]
XOR D, S	011011 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4

Примечания к Сводной Таблице

Примеч.1. Количество тактов для Hub-инструкций

Hub-инструкциям для своего выполнения необходимо от 7 до 22 тактов, в зависимости от временного взаиморасположения между окном доступа процессора к *Hub*-у и моментом выполнения инструкции. Каждые 16 тактов *Hub* предоставляет каждому процессору "окно доступа к *Hub*". Поскольку каждый из процессоров работает независимо от *Hub*, он должен синхронизироваться с *Hub* при выполнении *Hub*-инструкции. Первой *Hub*-инструкции в очереди потребуется от 0 до 15 тактов для синхронизации с окном доступа к *Hub* и затем еще 7 тактов для выполнения; таким образом, получаем от 7 до 22 (15 + 7) тактов для выполнения. После первой *Hub*-инструкции у данного *Cog* будет 9 (16 – 7) свободных тактов перед приходом следующего окна доступа – достаточное количество времени для выполнения двух 4-тактовых инструкций без потери синхронизации со следующим окном доступа. Для минимизации траты тактов, Вы можете вставлять две обычных инструкции между любыми двумя последовательно расположенными *Hub*-инструкциями без увеличения времени выполнения. Остерегайтесь того, что *Hub*-инструкции могут привести к возникновению неопределенности во времени выполнения, особенно первая *Hub*-инструкция в последовательности.

Примеч.2. Количество тактов для инструкций «Модифицировать-Перейти»

Инструкции, модифицирующие значение, после чего, возможно, выполняющие переход в зависимости от результата, требуют различного количества тактов, что зависит от того, будет ли выполняться переход. Такие инструкции требуют 4 такта, если переход необходим, и 8 тактов – если нет. Поскольку программные циклы, использующие эти инструкции, обычно должны выполняться быстро, они таким образом оптимизируются по скорости.

Примеч. 3. ADDABS выход C: Если S отрицат., C = инверсное беззн. заема (для D-S).
Примеч. 4. SUBABS выход C: Если S отрицат., C = инверсное беззн. переноса (для D+S).

ABS

Инструкция: Получить абсолютное значение числа.

ABS *AValue*, ⟨#⟩ *SValue*

Результат: Абсолютное значение *SValue* сохранено в *AValue*.

- ***AValue*** (d-поле) – регистр, в который записывается абсолютное значение *SValue*.
- ***SValue*** (s-поле) – регистр, либо 9-битная константа, чье абсолютное значение будет записано в *AValue*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
101010 001i 1111 ddddddddd sssssssss	Результат = 0	S[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$7FFF_FFFF; 2147483647	-	-	wz wc	$7FFF_FFFF; 2147483647	0	0
$----_----; -	$8000_0000; -2147483648	-	-	wz wc	$8000_0000; -2147483648[1]	0	1
$----_----; -	$8000_0001; -2147483647	-	-	wz wc	$7FFF_FFFF; 2147483647	0	1

[1] **Наименьшее отрицательное число (-2147483648) не имеет соответствующего положительного в 32-битном допкоде.**

Описание

ABS вычисляет абсолютное значение *SValue* и записывает результат в *AValue*.

При заданном воздействии **WZ,** флаг Z устанавливается в 1, если *SValue* равно нолю. При заданном воздействии **WC**, флаг C устанавливается в 1, если *SValue* отрицательное, либо очищается, (0) если *SValue* положительное. Результат записывается в *AValue*, если не задано воздействие **NR**.

Константы *Svalue* расширяются нолями, поэтому лучше использовать ABS с регистровыми величинами *SValue*.

ABSNEG

Инструкция: Получить отрицательное от величины абсолютного значения числа.

ABSNEG *NValue*, ⟨#⟩ *SValue*

Результат: Отрицательное значение от абсолютного *SValue* сохранено в *NValue*.

- ***NValue*** (d-поле) – регистр для записи отрицательного значения абсолютной величины *SValue*.
- ***SValue*** (s-поле) – регистр или 9-битная константа, отрицательное от абсолютной величины которого будет записано в *NValue*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
101011 001i	1111	ddddddddd sssssssss	Результат = 0	S[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$7FFF_FFFF; 2147483647	-	-	wz wc	$8000_0001; -2147483647	0	0
$----_----; -	$8000_0000; -2147483648	-	-	wz wc	$8000_0000; -2147483648	0	1
$----_----; -	$8000_0001; -2147483647	-	-	wz wc	$8000_0001; -2147483647	0	1

Описание

ABSNEG вычисляет отрицательное значение абсолютной величины *SValue* и записывает результат в *NValue*.

При заданном воздействии WZ, флаг Z устанавливается в 1, если *SValue* равно нолю. При заданном воздействии WC, флаг C устанавливается в 1, если *SValue* отрицательное, либо очищается (0), если *SValue* положительное. Результат записывается в *NValue*, если не задано воздействие NR.

Константы *Svalue* расширяются нолями, поэтому лучше использовать ABSNEG с регистровыми величинами *Svalue.*

ADD

Инструкция: Сложить две беззнаковые величины.

ADD *Value1*, ⟨#⟩ *Value2*

Результат: Сумма беззнакового *Value1* и беззнакового *Value2* сохранена в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину, складываемую с *Value2* и являющийся приемником для записи результата.
- ***Value2*** (s-поле) – регистр или 9-битная константа, величина которого складывается с величиной *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
100000 001i 1111 ddddddddd sssssssss	D + S = 0	Беззнаков. перенос	Written	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник	Z	C
$FFFF_FFFE; 4294967294	$0000_0001; 1	–	–	wz wc	$FFFF_FFFF; 4294967295	0	0
$FFFF_FFFE; 4294967294	$0000_0002; 2	–	–	wz wc	$0000_0000; 0	1	1
$FFFF_FFFE; 4294967294	$0000_0003; 3	–	–	wz wc	$0000_0001; 1	0	1

[1] **Источник и Приемник рассматриваются как величины без знака.**

Описание

ADD складывает две беззнаковые величины *Value1* и *Value2* и сохраняет результат в регистр *Value1*.

При заданном воздействии WZ, флаг Z устанавливается в 1, если результат *Value1* + *Value2* равен нолю. При заданном воздействии WC, флаг C устанавливается в 1, если в результате суммирования получен беззнаковый перенос старшего бита (32-битное переполнение). Результат записывается в *Value1*, если не задано воздействие NR.

Для сложения двух беззнаковых значений размером в несколько *long*, используйте инструкцию ADD сопровождаемую инструкцией ADDX. Для подробной информации см. ADDX на стр. 423.

ADDABS

Инструкция: Сложить абсолютное значение с другой величиной.

ADDABS *Value*, ⟨#⟩ *SValue*

Результат: Сумма величины *Value* и абсолютного значения знаковой величины *SValue* сохранена в *Value*.

- ***Value*** (d-поле) – регистр, содержащий величину для суммирования с абсолютным значением *SValue* и являющийся приемником для записи результата.
- ***SValue*** (s-поле) – регистр или 9-битная константа, абсолютное значение величины которого суммируется с *Value*. Константы *SValue* расширяются нолями (всегда положительны), поэтому лучше использовать ADDABS с регистровыми величинами *SValue.*

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
100010 001i 1111	ddddddddd	sssssssss	D + |S| = 0	Беззнаков. перенос[1]	Записан	4

1: Если S отрицательное, C = инверсное беззнакового заема (для D-S).

Вход					Выход		
Приемник[1]	Источник	Z	C	Возд.	Приемник	Z	C
$FFFF_FFFD; 4294967293	$0000_0004; 4	–	–	wz wc	$0000_0001; 1	0	1
$FFFF_FFFD; 4294967293	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	1
$FFFF_FFFD; 4294967293	$0000_0002; 2	–	–	wz wc	$FFFF_FFFF; 4294967295	0	0
$FFFF_FFFD; 4294967293	$FFFF_FFFF; -1	–	–	wz wc	$FFFF_FFFE; 4294967294	0	1
$FFFF_FFFD; 4294967293	$FFFF_FFFE; -2	–	–	wz wc	$FFFF_FFFF; 4294967295	0	1
$FFFF_FFFD; 4294967293	$FFFF_FFFD; -3	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFD; 4294967293	$FFFF_FFFC; -4	–	–	wz wc	$0000_0001; 1	0	0

[1] Приемник рассматривается как величина без знака.

Описание

ADDABS суммирует величину *Value* и абсолютное значение величины *SValue* и сохраняет результат в регистр *Value*.

При заданном воздействии **WZ,** флаг Z устанавливается в 1, если результат *Value* + |*SValue*| равен нолю. При заданном воздействии **WC**, флаг C устанавливается в 1, если в результате суммирования получен беззнаковый перенос старшего бита (32-битное переполнение). Результат записывается в *Value*, если не задано воздействие **NR**.

ADDS

Инструкция: Суммировать две знаковых величины.

ADDS *SValue1*, ⟨#⟩ *SValue2*

Результат: Сумма знакового *SValue1* и знакового *SValue2* сохранена в *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину, складываемую с *SValue2* и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) – регистр или 9-битная константа, величина которого складывается с величиной *SValue1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
110100 001i 1111 ddddddddd sssssssss	D + S = 0	Знаковое переполн.	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$FFFF_FFFF; -1	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	–	–	wz wc	$FFFF_FFFF; -1	0	0
$7FFF_FFFE; 2147483646	$0000_0001; 1	–	–	wz wc	$7FFF_FFFF; 2147483647	0	0
$7FFF_FFFE; 2147483646	$0000_0002; 2	–	–	wz wc	$8000_0000; -2147483648	0	1
$8000_0001; -2147483647	$FFFF_FFFF; -1	–	–	wz wc	$8000_0000; -2147483648	0	0
$8000_0001; -2147483647	$FFFF_FFFE; -2	–	–	wz wc	$7FFF_FFFF; 2147483647	0	1

Описание

ADDS суммирует две знаковые величины *SValue1* и *SValue2,* и сохраняет результат в регистр *SValue1*.

При заданном воздействии WZ, флаг Z устанавливается в 1, если результат *SValue1* + *SValue2* равен нолю. При заданном воздействии WC, флаг C устанавливается в 1, если в результате суммирования получен знаковый перенос старшего бита. Результат записывается в *SValue1*, если не задано воздействие NR.
Для сложения двух знаковых величин размером в несколько *long*-ов, используется инструкция ADD, затем, возможно, ADDX, и в конце – ADDSX. Для подробной информации см. ADDSX на стр. 421.

ADDSX

Инструкция: Сложить две знаковые величины плюс C.

ADDSX *SValue1*, ⟨#⟩ *SValue2*

Результат: Сумма знакового *SValue1* и знакового *SValue2* плюс C сохранена в *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину, складываемую с *SValue2* плюс C, и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) – регистр или 9-битная константа, величина которого складывается с величиной *SValue1* плюс C.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
110110 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Знаковое переполн.	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$FFFF_FFFE; -2	$0000_0001; 1	x	0	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFE; -2	$0000_0001; 1	0	1	wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$0000_0001; 1	1	1	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	x	0	wz wc	$FFFF_FFFF; -1	0	0
$0000_0001; 1	$FFFF_FFFE; -2	0	1	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$FFFF_FFFE; -2	1	1	wz wc	$0000_0000; 0	1	0
$7FFF_FFFE; 2147483646	$0000_0001; 1	x	0	wz wc	$7FFF_FFFF; 2147483647	0	0
$7FFF_FFFE; 2147483646	$0000_0001; 1	x	1	wz wc	$8000_0000; -2147483648	0	1
$7FFF_FFFE; 2147483646	$0000_0002; 2	x	0	wz wc	$8000_0000; -2147483648	0	1
$8000_0001; -2147483647	$FFFF_FFFF; -1	x	0	wz wc	$8000_0000; -2147483648	0	0
$8000_0001; -2147483647	$FFFF_FFFE; -2	x	0	wz wc	$7FFF_FFFF; 2147483647	0	1
$8000_0001; -2147483647	$FFFF_FFFE; -2	x	1	wz wc	$8000_0000; -2147483648	0	0

Описание

ADDSX (Сложить Знаковое, Расширенное), складывает две знаковые величины *SValue1* и *SValue2* плюс C, и сохраняет результат в регистре *SValue1*. Используйте инструкцию ADDSX для выполнения суммирования со знаком величин размерностью в несколько *long*, например, для 64-битного сложения.

При выполнении знаковых операций со значениями размером более одного *long*, первая инструкция — беззнаковая, (напр.: ADD), любая промежуточная инструкция —

беззнаковая расширенная (напр.: ADDX), и последняя – знаковая расширенная (напр.: ADDSX). Используйте воздействия WC, и опционально WZ, с предшествующими инструкциями ADD и ADDX.

Например, знаковое сложение двойных *long*-ов (64-битное) может выглядеть так:

```
add   XLow, YLow   wc wz  'Add low longs together; save C and Z
addsx XHigh,YHigh         'Add high longs together
```

После выполнения приведенного выше, 64-битный результат находится в *long*-регистрах *XHigh*:*XLow*. Если в начале *XHigh*:*XLow* было равно $0000_0001:0000_0000 (4294967296) и *YHigh*:*YLow* было $FFFF_FFFF:FFFF_FFFF (-1), результат в регистрах *XHigh*:*XLow* будет равен $0000_0000:FFFF_FFFF (4294967295). Это показано ниже.

```
                        Hexadecimal              Decimal
                      (high)     (low)
   (XHigh:XLow)     $0000_0001:0000_0000       4,294,967,296
 + (YHigh:YLow)   + $FFFF_FFFF:FFFF_FFFF     +            -1
                  ----------------------     ---------------
                  = $0000_0000:FFFF_FFFF     = 4294967295
```

Знаковое сложение тройных *long*-ов (96-битное) будет выглядеть похоже, но с инструкцией ADDX, вставленной между ADD и ADDSX:

```
add     XLow,  YLow   wc wz   'Add low longs; save C and Z
addx    XMid,  YMid   wc wz   'Add middle longs; save C and Z
addsx   XHigh, YHigh          'Add high longs
```

Естественно, что для того, чтобы определить нулевой результат либо знаковое переполнение, в последней инструкции, **ADDSX,** необходимо указать воздействия WC и WZ. Отметьте, что в процессе выполнения этой многошаговой операции, флаг Z всегда сигнализирует, равен ли результат нулю, но флаг C сигнализирует беззнаковый переход лишь до последней инструкции, ADDSX, в которой он уже становится признаком знакового переполнения.

Для **ADDSX,** при заданном воздействии WZ, флаг Z устанавливается в 1, если Z был установлен ранее и *SValue1* + *SValue2* + C равно нолю (используйте WC и WZ в предыдущих инструкциях ADD или ADDX). При указанном воздействии WC, флаг C устанавливается в 1, если в результате суммирования получен знаковый перенос старшего бита. Результат записывается в *SValue1*, если не задано воздействие NR.

ADDX

Инструкция: Сложить две беззнаковые величины плюс C.

ADDX *Value1*, ⟨#⟩ *Value2*

Результат: Сумма знакового *Value1* и знакового *Value2* плюс C сохранена в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину, складываемую с *Value2* плюс C, и являющийся приемником для записи результата.
- ***Value2*** (s-поле) – регистр или 9-битная константа, величина которого складывается с величиной *Value1* плюс C.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
110010 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Беззнаков. перенос	Записан	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник	Z	C
$FFFF_FFFE; 4294967294	$0000_0001; 1	x	0	wz wc	$FFFF_FFFF; 4294967295	0	0
$FFFF_FFFE; 4294967294	$0000_0001; 1	0	1	wz wc	$0000_0000; 0	0	1
$FFFF_FFFE; 4294967294	$0000_0001; 1	1	1	wz wc	$0000_0000; 0	1	1

[1] **Источник и Приемник рассматриваются как величины без знака.**

Описание

ADDX (Сложить Расширенное), складывает две беззнаковые величины *Value1* и *Value2* плюс C, и сохраняет результат в регистре *Value1*.

Инструкция ADDX используется для выполнения сложения величин размером более одного *long*; например, сложения 64-битных величин.

При выполнении знаковых операций с величинами размером более одного *long* (*multi-long* величинами) первая инструкция беззнаковая, (напр.: ADD), любая промежуточная инструкция — беззнаковая расширенная (напр.: ADDX), и последняя – беззнаковая расширенная (напр.:ADDX) либо знаковая расширенная, в зависимости от природы самих *multi-long* величин. Здесь мы обсудим беззнаковые величины; пример со знаковыми *multi-long* величинами см. **ADDSX,** на стр. 421. Убедитесь, что воздействие WC и, опционально, WZ заданы для предшествующих инструкций ADD и ADDX.

Например, беззнаковое сложение двойных *long*-ов (64-битное) может выглядеть так:

```
add  XLow,  YLow   wc wz   'Add low longs together; save C and Z
addx XHigh, YHigh          'Add high longs together
```

После выполнения приведенного выше, 64-битный результат находится в *long*-регистрах *XHigh*:*XLow*. Если в начале *XHigh*:*XLow* было равно $0000_0000:FFFF_FFFF (4294967295) и *YHigh*:*YLow* было $0000_0000:0000_0001 (1), результат в регистрах *XHigh*:*XLow* будет равен $0000_0001:0000_0000 (4,294,967,296). Это показано ниже.

```
                          Hexadecimal                 Decimal
                        (high)     (low)
   (XHigh:XLow)      $0000_0000:FFFF_FFFF        4294967295
 + (YHigh:YLow)    + $0000_0000:0000_0001      +          1
                   ---------------------      ---------------
                   = $0000_0001:0000_0000      = 4,294,967,296
```

Естественно, что для того, чтобы определить нулевой результат либо беззнаковое переполнение, в последней инструкции, **ADDX,** необходимо указать воздействия **WC** и **WZ**.

Для **ADDX**, при заданном воздействии **WZ,** флаг **Z** устанавливается в 1, если ранее был установлен **Z**, и *Value1* + *Value2* + **C** равно нолю (используйте **WC** и **WZ** в предыдущих инструкциях **ADD** или **ADDX**). При заданном воздействии **WC**, флаг **C** устанавливается в 1, если в результате суммирования получен беззнаковый перенос старшего бита (32-битн. переполнение). Результат записывается в *Value1*, если не задано воздействие **NR**.

AND

Инструкция: Побитовое И двух величин.

AND *Value1*, ⟨#⟩ *Value2*

Результат: *Value1* И *Value2* сохранено в регистре *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину для побитного И с величиной *Value2* и являющийся приемником для записи результата.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого побитно умножается по И на величину *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
011000 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
s0000_000A; 10	s0000_0005; 5	–	–	wz wc	s0000_0000; 0	1	0
s0000_000A; 10	s0000_0007; 7	–	–	wz wc	s0000_0002; 2	0	1
s0000_000A; 10	s0000_000F; 15	–	–	wz wc	s0000_000A; 10	0	0

Описание

AND (Побитовое И) выполняет Побитовое И значений *Value2* и *Value1*, результат сохраняется в *Value1*.

Флаг Z устанавливается в 1, если при заданном воздействии WZ результат операции *Value1* И *Value2* равен нолю. Флаг C устанавливается в 1, если при заданном воздействии WC в результате содержится нечетное количество установленных в 1 битов. Результат записывается в *Value1*, если не задано воздействие NR.

ANDN

Инструкция: Побитовое И величины и инверсного значения другой величины.

ANDN *Value1*, ⟨#⟩ *Value2*

Результат: *Value1* И !*Value2* сохранено в регистре *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину для побитного И с величиной !*Value2* и являющийся приемником для записи результата.
- ***Value2*** (s-поле) – регистр или 9-битная константа, величина которого инвертируется (Побитовое НЕ) и побитно умножается по И с значением *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
011001 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$F731_125A; -147778982	$FFFF_FFFA; -6	–	–	wz wc	$0000_0000; 0	1	0
$F731_125A; -147778982	$FFFF_FFF8; -8	–	–	wz wc	$0000_0002; 2	0	1
$F731_125A; -147778982	$FFFF_FFF0; -16	–	–	wz wc	$0000_000A; 10	0	0

Описание

ANDN (Побитовое И НЕ) выполняет побитную операцию И величины *Value1* с инвертированным значением (Побитовое НЕ) величины *Value2,* и сохраняет результат в *Value1*.

Флаг Z устанавливается в 1, если при заданном воздействии WZ результат операции *Value1* И !*Value2* равен нолю. Флаг C устанавливается в 1, если при заданном воздействии WC в результате содержится нечетное количество установленных в 1 битов. Результат записывается в *Value1*, если не задано воздействие NR.

CALL

Инструкция: Переход на адрес с дальнейшим возвратом на следующую инструкцию.

CALL #*Symbol*

Результат: PC + 1 записан в s-поле регистра, указанного в d-поле.

- ***Symbol*** (s-поле) – это 9-битная константа, значение которой является адрес перехода. Это поле должно содержать DAT-символику, указанную как константа (#символика), а соответствующий код должен в завершении выполнить инструкцию RET, отмеченную такой же символикой плюс суффикс "_ret" (символика_ret RET).

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
010111 0011 1111 ????????? sssssssss	Результат = 0	---	Записано	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C[2]
s----_----; -	s----_----; -	-	-	wz wc	31:9 неизм., 8:0 = PC+1	0	1

[1] **s-поле приемника (младшие 9 бит) переписывается адресом возврата (PC+1) во время выполнения.**

[2] **Флаг C устанавливается в 1, кроме случая когда PC+1 равен 0, что очень маловероятно, поскольку при этом CALL будет выполняться с самого начала cog RAM ($1FF, регистр специальных функций VSCL).**

Описание

CALL записывает адрес следующей инструкции (PC + 1), после чего выполняет переход на ***Symbol***. Подпрограмма по адресу ***Symbol*** в завершении должна выполнить инструкцию RET для возврата на записанный адрес (т.е. на инструкцию, следующую за CALL). Для правильной компиляции и выполнения команды CALL, инструкция RET подпрограммы по адресу *Symbol* должна быть обозначена в формате *Symbol,* с символами "_ret", добавленными к ней. Для чего это выполняется, описано ниже.

Аппаратная часть ИМС Propeller не использует стек вызовов, поэтому адрес возврата должен сохраняться в другом виде. Во время компиляции ассемблер определяет необходимую подпрограмму и ее инструкцию RET (обозначенные соответственно как Symbol и Symbol_ret) и преобразовывает эти адреса в поля s- и d- инструкции CALL. Это даст инструкции CALL знать, куда нужно переходить и откуда будет нужно возвращаться.

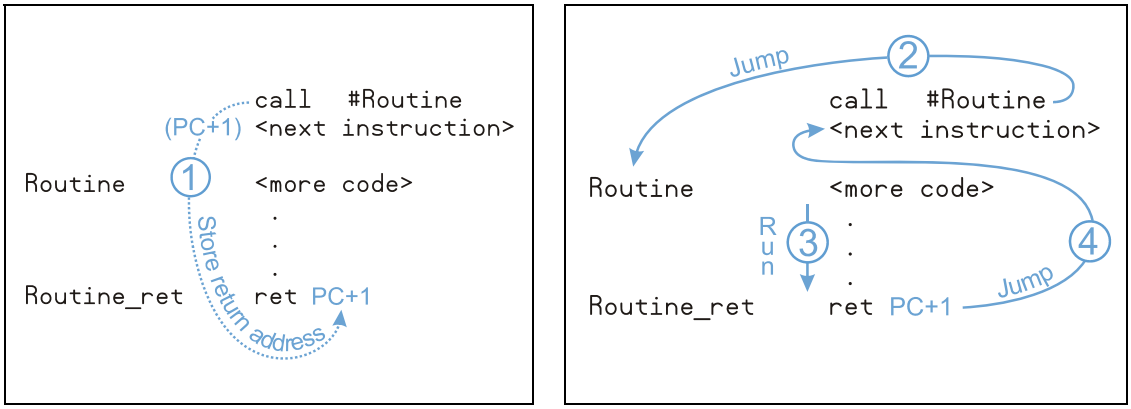
Во время выполнения, инструкция CALL в первую очередь сохраняет свой адрес возврата (PC+1) в место по адресу, откуда в дальнейшем будет выполнен возврат, то

есть по адресу инструкции “Symbol_ret RET”. В действительности инструкция RET представляет собой инструкцию JMP без предустановленного адреса для перехода, и такое действие обеспечивает её таким адресом для дальнейшего возврата. После сохранения адреса возврата, CALL переходит по адресу приемника, то есть Symbol.

В приведенной ниже диаграмме используется короткая программа для демонстрации поведения инструкции CALL во время выполнения: опреация схранения (слева) и операции перехода-выполнения-возврата (справа).

Работа инструкции CALL

Сохранение **Переход, выполнение и возврат**



В этом примере, при выполнении инструкции CALL, происходит следующее:

① Cog сохраняет адрес возврата (PC+1, то есть адрес <next instruction>) в поле источника (s-поле) в ячейке по адресу Routine_ret (левый рисунок).

② Cog переходит по адресу Routine (правый рисунок).

③ Выполняются инструкции подпрограммы Routine, доходя до строки Routine_ret.

④ Поскольку по адресу Routine_ret находится инструкция RET с обновленным полем источника (s-поле), представляющим собой адрес возврата, записанный на первом шаге, Cog возвращается, или переходит назад, на строку <next instruction>.

Такая природа инструкции CALL определяет следующие ограничения:

- В каждой подпрограмме может использоваться лишь одна ассоциированная с ней инструкция RET. Если в подпрограмме необходимо выполнить более чем одну точку выхода, выполните одну из них инструкцией RET, а в остальных используйте инструкции перехода (JMP) на эту, установленную ранее, инструкцию RET.
- Вызов подпрограммы не может быть реккурсивным. Выполнение вложенного вызовова подпрограммы перезапишет адрес возврата предыдущего вызова.

CALL – это в действительности подмножество инструкции JMPRET; на самом деле у нее такой же код операции, как и у JMPRET, но с установленным i-полем (поскольку CALL использует только прямые величины), а также d-полем, установленным на адрес метки с именем Symbol_ret.

Адрес возврата (PC + 1) записывается в поле источника (s-поле) регистра по адресу Symbol_ret, если не указано воздействие NR. Естественно, что указание NR для инструкции CALL не рекомендуется, поскольку это преобразует ее в JMP или RET.

CLKSET

Инструкция: Установить режим генератора во время выполнения.

CLKSET ***Mode***

- ***Mode*** (d-поле) – регистр, содержащий 8-битный набор для записи в регистр CLK.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0001 1111 ddddddddd ------000	---	---	Not Written	7..22

Вход					Выход		
Приемник	Источник[1]	Z	C	Возд.	Приемник[2]	Z	C
$0000_006F; 111	%0_00000000; 0	–	–	wr wz wc	$0000_0007; 7	0	0

[1] Значение источника автоматически устанавливается ассемблером в 0, чтобы обозначить, что это hub-инструкция CLKSET
[2] Приемник не перезаписывается, если не задано воздействие WR.

Описание

CLKSET изменяет режим работы генератора во время выполнения программы. Инструкция CLKSET выполняется аналогично команде *Spin* с таким же именем (см. CLKSET на стр. 213), за исключением того, что она устанавливает только режим генератора, а не его частоту.

После выдачи инструкции CLKSET важно обновить частоту Системного Генератора записью её величины в его адрес в Основном ОЗУ (*long* 0): WRLONG freqaddr, #0. Если значение частоты Системного Генератора не было обновлено, другие объекты будут вести себя непредсказуемо в связи с неверным значением частоты.

CLKSET – это *Hub*-инструкция. *Hub*-инструкция требует от 7 до 22 тактов для выполнения, в зависимости от нахождения окна предоставления доступа к *Hub* в момент прихода инструкции. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

CMP

Инструкция: Сравнить две беззнаковые величины.

CMP *Value1*, ⟨#⟩ *Value2*

Результат: Опционально: признак равенства, больше/меньше записаны в флаги Z и C.

- ***Value1*** (d-поле) – регистр, содержащий величину для сравнения с *Value2*.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *Value1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
100001 000i 1111	ddddddddd	sssssssss	D = S	Беззнаков. (D < S)	Не записан	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник[2]	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$8000_0000; 2147483648	$7FFF_FFFF; 2147483647	–	–	wr wz wc	$0000_0001; 1	0	0[4]
$7FFF_FFFF; 2147483647	$8000_0000; 2147483648	–	–	wr wz wc	$FFFF_FFFF; -1[3]	0	1[4]
$FFFF_FFFE; 4294967294	$FFFF_FFFF; 4294967295	–	–	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4294967294	$FFFF_FFFE; 4294967294	–	–	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; 4294967294	$FFFF_FFFD; 4294967293	–	–	wr wz wc	$0000_0001; 1	0	0

[1] Источник и Приемник рассматриваются как величины без знака.

[2] Приемник не перезаписывается, если не задано воздействие WR.

[3] Приемник на выходе (записанное значение) может рассматриваться как величина с- или без знака; здесь она показана со знаком лишь в описательных целях.

[4] Значение флага C в результате выполнения инструкции CMP (Compare Unsigned) может отличаться от значения после CMPS (Compare Signed), где «интерпретируемые знаки» Источника и Приемника различны. В приведенном выше первом примере второй группы видно, что инструкция CMP очищает C, потому как беззнаковое $8000_0000 (2147483648) не меньше, чем беззнаковое $7FFF_FFFF (2147483647). Однако CMPS установила бы C, потому как знаковое $8000_0000 (-2147483648) меньше, чем знаковое $7FFF_FFFF (2147483647). Второй пример является таким же случаем, но когда знаки Источника и Приемника поменяны местами.

Описание

CMP (Сравнить Беззнаковое) сравнивает беззнаковые величины *Value1* и *Value2*. Флаги Z и C, при записи, указывают соответственно отношения равенства и больше либо меньше между величинами.

При указанном воздействии **WZ,** флаг Z устанавливается в 1, если *Value1* равно *Value2*. При указанном воздействии WC, флаг C устанавливается в 1, если *Value1* меньше *Value2*.

Этот результат не записывается, если не указано воздействие WR. При указанном WR, инструкция CMP работает точно так, как инструкция SUB.

Для сравнения беззнаковых, *multi-long* величин, используйте команду CMP, сопровождаемую CMPX. Для более подробной информации см. CMPX на стр. 438.

CMPS

Инструкция: Сравнить две знаковые величины.

CMPS *SValue1*, ⟨#⟩ *SValue2*

Результат: Опционально признак равенства, больше/меньше записаны в флаги Z и C.

- ***SValue1*** (d-поле) – регистр, содержащий величину для сравнения с *SValue2*.
- ***SValue2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *SValue1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
110000 000i 1111	ddddddddd	sssssssss	D = S	Беззнаков. (D < S)	Не записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wr wz wc	$FFFF_FFFF; -1	0	1
$8000_0000; -2147483648	$7FFF_FFFF; 2147483647	–	–	wr wz wc	$0000_0001; 1	0	1[2]
$7FFF_FFFF; 2147483647	$8000_0000; -2147483648	–	–	wr wz wc	$FFFF_FFFF; -1	0	0[2]
$8000_0000; -2147483648	$0000_0001; 1	–	–	wr wz wc	$7FFF_FFFF; 2147483647[3]	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	–	–	wr wz wc	$8000_0000; -2147483648[3]	0	0
$FFFF_FFFE; -2	$FFFF_FFFF; -1	–	–	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFE; -2	–	–	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	–	–	wr wz wc	$0000_0001; 1	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR.**

[2] **Значение флага C в результате выполнения инструкции CMPS (Compare Signed) может отличаться от значения после CMP (Compare Unsigned), где «интерпретируемые знаки» Источника и Приемника различны. В приведенном выше первом примере второй группы видно, что инструкция CMPS установила C, потому как знаковое $8000_0000 (-2147483648) меньше, чем знаковое $7FFF_FFFF (2147483647). Однако CMP бы очистила C, потому как беззнаковое $8000_0000 (2147483648) не меньше, чем беззнаковое $7FFF_FFFF (2147483647). Второй пример этой группы является таким же случаем, но когда знаки Источника и Приемника поменяны местами.**

[3] **Примеры третьей группы показывают случаи, когда сравнение правильно отражается на флагах, но значение Приемника на выходе пересекает разрядную сетку знакового числа (ошибка переполнения со знаком), в положительном и**

отрицательном направлениях. Эта ситуация переполнения со знаком не может отразиться на состоянии флагов. В этом случае важно, чтобы приложение выполняло инструкцию CMPS без воздействия WR, отмечало состояние флага C (знаковый заем), а затем выполняло инструкцию SUBS и отмечало состояние флага C (знаковое переполнение).

Описание

CMPS (Сравнить Знаковое) сравнивает знаковые величины *SValue1* и *SValue2*. Флаги Z и C, при записи, указывают соответственно отношения равенства и больше либо меньше между величинами.

При указанном воздействии WZ, флаг Z устанавливается в 1, если *SValue1* равно *SValue2*. При указанном воздействии WC, флаг C устанавливается в 1 если *SValue1* меньше *SValue2*.

Для сравнения знаковых, multi-long величин, используйте команду CMP, возможно сопровождаемую CMPX. Для более подробной информации см. CMPX на стр. 438.

CMPSUB

Инструкция: Сравнить две беззнаковые величины и вычесть вторую, если она меньше либо равна.

CMPSUB *Value1*, ⟨#⟩ *Value2*

Результат: Опционально, *Value1 = Value1 – Value2*, флаги Z, C = результат сравнения.

- ***Value1*** (d-поле) – регистр, содержащий величину для сравнения с *Value2* и являющийся приемником для записи результата при вычитании.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается и возможно вычитается из *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
111000 001i 1111 ddddddddd sssssssss	D = S	Беззнак. (D => S)	Записан	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0003; 3	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	1
$0000_0003; 3	$0000_0004; 4	–	–	wz wc	$0000_0003; 3	0	0

[1] Источник и Приемник рассматриваются как величины без знака.

Описание

CMPSUB сравнивает беззнаковые величины *Value1* и *Value2*, и если *Value2* равна или меньше *Value1* , она вычитается из *Value1* (при указанном WR). Флаги Z и C, при записи, указывают соотношения равенства и больше либо меньше между величинами.

При указанном воздействии WZ, флаг Z устанавливается в 1, если *Value1* равно *Value2*. При указанном воздействии WC, флаг устанавливается в 1, если вычитание возможно (*Value1* больше либо равно *Value2*). Результат, если есть, записывается в *Value1*, если не указано воздействие NR.

CMPSX

Инструкция: Сравнить две знаковые величины плюс C.

CMPSX *SValue1*, ⟨#⟩ *SValue2*

Результат: Опционально записывается статус равенства и больше или меньше в флагах Z и C.

- ***SValue1*** (d-поле) – регистр, содержащий величину для сравнения с *SValue2*.
- ***SValue2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *SValue1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
110001 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Знаковый (D < S+C)	Не записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
$0000_0003; 3	$0000_0002; 2	x	0	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0002; 2	0	1	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0002; 2	1	1	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	0	0	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0003; 3	1	0	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	x	1	wr wz wc	$FFFF_FFFF; -1	0	1
$0000_0003; 3	$0000_0004; 4	x	0	wr wz wc	$FFFF_FFFF; -1	0	1
$0000_0003; 3	$0000_0004; 4	x	1	wr wz wc	$FFFF_FFFE; -2	0	1
$8000_0000; -2147483648	$7FFF_FFFF; 2147483647	0	0	wr wz wc	$0000_0001; 1	0	1[2]
$7FFF_FFFF; 2147483647	$8000_0000; -2147483648	0	0	wr wz wc	$FFFF_FFFF; -1	0	0[2]
$8000_0000; -2147483648	$0000_0001; 1	0	0	wr wz wc	$7FFF_FFFF; 2147483647[3]	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	0	0	wr wz wc	$8000_0000; -2147483648[3]	0	0

$FFFF_FFFE; -2	$FFFF_FFFF; -1	x	0	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFF; -1	x	1	wr wz wc	$FFFF_FFFE; -2	0	1
$FFFF_FFFE; -2	$FFFF_FFFE; -2	0	0	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$FFFF_FFFE; -2	1	0	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; -2	$FFFF_FFFE; -2	x	1	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFD; -3	x	0	wr wz wc	$0000_0001; 1	0	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	0	1	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	1	1	wr wz wc	$0000_0000; 0	1	0

[1] Приемник не перезаписывается, если не задано воздействие WR.

[2] Значение флага C в результате выполнения инструкции CMPSX (Compare Signed, Extended) может отличаться от значения после CMPX (Compare Unsigned, Extended), где «интерпретируемые знаки» Источника и Приемника различны. В приведенном выше первом примере второй группы видно, что инструкция CMPSX устанавливает C, потому как знаковое $8000_0000 ((-2147483648) меньше, чем знаковое $7FFF_FFFF (2147483647). Однако CMPX очистила бы C, потому как беззнаковое $8000_0000 (2147483648) не меньше, чем беззнаковое $7FFF_FFFF (2147483647). Второй пример группы является таким же случаем, но когда знаки Источника и Приемника поменяны местами. Отметьте, что примеры с различающимися Z и C не приведены, но ожидаемые результаты такие же, как и для других примеров.

[3] Примеры третьей группы показывают случаи, когда сравнение правильно отражается на флагах, но значение Приемника на выходе пересекает разрядную сетку знакового числа (ошибка переполнения со знаком), в положительном и отрицательном направлениях. Эта ситуация переполнения со знаком не может отразиться на состоянии флагов. В этом случае важно, чтобы приложение само определяло подобную ситуацию другими методами.

Описание

CMPSX (Сравнить Знаковое, Расширенная) сравнивает знаковые величины *SValue1* и *SValue2* плюс C. Флаги Z и C, при записи, указывают соответственно отношения равенства и больше либо меньше между величинами. Инструкция CMPSX используется для выполнения сравнения со знаком величин размерностью в несколько *long*-ов, например, 64-х битных.

В операциях со знаком с числами размерностью в несколько *long*-ов, первая операция является беззнаковой (например, CMP), все промежуточные инструкции являются беззнаковыми расширенными (например, CMPX), а последняя инструкция – знаковая расширенная (например, CMPSX). Не забывайте устанавливать флаги WC, и, при необходимости, WZ, для всех инструкций в операциях сравнения.

Например, сравнение величин со знаком, размером в два слова (64-битных) выглядит следующим образом:

```
cmp     XLow,  YLow   wc wz 'Compare low longs; save C and Z
cmpsx   XHigh, YHigh  wc wz 'Compare high longs; save C and Z
```

После выполнения приведенного кода, флаги C и Z будут указывать на отношение между двумя 64-битными величинами. Если XHigh:XLow было равно $FFFF_FFFF:FFFF_FFFF (-1), а YHigh:YLow было $0000_0000:0000_0001 (1), то в результате флаги будут равны: Z = 0 и C = 1 (*Value1* < *Value2*). Это объясняется ниже.

Имейте в виду, что сравнение – это на самом деле лишь вычитание без записи результата; однако флаги Z и C при этом все равно важны.

```
                          Hexadecimal                 Decimal        Flags
                        (high)     (low)
   (XHigh:XLow)      $FFFF_FFFF:FFFF_FFFF                   -1       n/a
 - (YHigh:YLow)    - $0000_0000:0000_0001       -            1       n/a
                   ----------------------       ---------------    --------
                   = $FFFF_FFFF:FFFF_FFFE       =           -2     Z=0, C=1
```

Сравнение 96-битных величин со знаком будет выглядеть так же, но со вставленной между CMP и CMPSX инструкцией CMPX:

```
  cmp     XLow,  YLow   wc wz  'Compare low longs; save C and Z
  cmpx    XMid,  YMid   wc wz  'Compare middle longs; save C and Z
  cmpsx   XHigh, YHigh  wc wz  'Compare high longs; save C and Z
```

Для **CMPSX**, при указанном воздействии **WZ,** флаг **Z** будет в 1, если он был установлен ранее и *SValue1* равно *SValue2* + C. (используйте **WC** и **WZ** в предыдущих инструкциях **CMP** и **CMPX**). При указанном воздействии **WC**, флаг **C** устанавливается в 1, если *SValue1* меньше *SValue2* (как величина размером в несколько *long*-ов).

CMPX

Инструкция: Сравнить две беззнаковые величины плюс C.

CMPX *Value1*, ⟨#⟩ *Value2*

Результат: Опционально статус равенства, больше или меньше записан в флаги Z и C.

- ***Value1*** (d-поле) – регистр, содержащий величину для сравнения с *Value2*.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
110011 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	(D < S+C)	Не записан	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник[2]	Z	C
$0000_0003; 3	$0000_0002; 2	x	0	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0002; 2	0	1	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0002; 2	1	1	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	0	0	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0003; 3	1	0	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	x	1	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$0000_0003; 3	$0000_0004; 4	x	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$0000_0003; 3	$0000_0004; 4	x	1	wr wz wc	$FFFF_FFFE; -2[3]	0	1
$8000_0000; 2147483648	$7FFF_FFFF; 2147483647	0	0	wr wz wc	$0000_0001; 1	0	0[4]
$7FFF_FFFF; 2147483647	$8000_0000; 2147483648	0	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1[4]
$FFFF_FFFE; 4294967294	$FFFF_FFFF; 4294967295	x	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4294967294	$FFFF_FFFF; 4294967295	x	1	wr wz wc	$FFFF_FFFE; -2[3]	0	1
$FFFF_FFFE; 4294967294	$FFFF_FFFE; 4294967294	0	0	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; 4294967294	$FFFF_FFFE; 4294967294	1	0	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; 4294967294	$FFFF_FFFE; 4294967294	x	1	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4294967294	$FFFF_FFFD; 4294967293	x	0	wr wz wc	$0000_0001; 1	0	0
$FFFF_FFFE; 4294967294	$FFFF_FFFD; 4294967293	0	1	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; 4294967294	$FFFF_FFFD; 4294967293	1	1	wr wz wc	$0000_0000; 0	1	0

[1] Источник и Приемник рассматриваются как величины без знака.

[2] Приемник не перезаписывается, если не задано воздействие WR.

[3] Приемник на выходе (его записанное значение) может рассматриваться как величина с- или без знака; здесь, для примера, она показана со знаком.

[4] Значение флага C в результате выполнения инструкции CMPX (Compare Unsigned, Extended) может отличаться от значения после CMPSX (Compare Signed, Extended), где «интерпретируемые знаки» Источника и Приемника различны. В приведенном

выше первом примере второй группы видно, что инструкция CMPX очистила C, потому как беззнаковое $8000_0000 (2147483648) не меньше, чем беззнаковое $7FFF_FFFF (2147483647). Однако CMPSX устанавила бы C, потому как знаковое $8000_0000 (-2147483648) меньше, чем знаковое $7FFF_FFFF (2147483647). Второй пример группы является таким же случаем, но когда знаки Источника и Приемника поменяны местами. Отметьте, что примеры с различающимися Z и C не приведены, но ожидаемые результаты такие же, как и для других примеров.

Описание

CMPX (Сравнить, Расширенная) сравнивает значения беззнаковых величин *Value1* и *Value2* плюс C. Флаги Z и C, при записи, указывают соответственно отношения равенства и больше либо меньше между величинами. Инструкция CMPX используется для выполнения сравнения величин размером в несколько *long*-ов, например, 64-битных беззнаковых.

При работе с числами размерностью в несколько *long*-ов, первая операция является беззнаковой (например, CMP), все промежуточные инструкции являются беззнаковыми расширенными (например, CMPX), а последняя инструкция – беззнаковая расширенная (CMPX) либо знаковая расширенная (CMPSX), в зависимости от природы самих чисел. Здесь мы рассмотрим беззнаковые величины; примеры работы с числами размерностью несколько слов со знаком приведены в секции CMPSX на стр. 364. Не забывайте устанавливать флаги WC, и, при необходимости, WZ, для всех инструкций в операциях сравнения

Например, сравнение величин без знака, размером в два слова (64-битных), выглядит следующим образом:

```
cmp     XLow,  YLow   wc wz  'Compare low longs; save C and Z
cmpx    XHigh, YHigh  wc wz  'Compare high longs
```

После выполнения приведенного примера, флаги C и Z будут показывать отношение между двумя 64-битными величинами. Если XHigh:XLow было равно $0000_0001:0000_0000 (4,294,967,296), а YHigh:YLow было $0000_0000:0000_0001 (1), то в результате флаги примут значения: Z = 0 и C = 0; (Value1 > Value2). Это объясняется на рисунке ниже. Имейте в виду, что сравнение – это на самом деле лишь вычитание без записи результата; однако флаги Z и C при этом все равно важны.

```
                        Hexadecimal                 Decimal        Flags
                          (high)     (low)
   (XHigh:XLow)      $0000_0001:0000_0000        4,294,967,296       n/a
 - (YHigh:YLow)    - $0000_0000:0000_0001      -             1       n/a
                     ----------------------      --------------     --------
                     = $0000_0000:FFFF_FFFF      = 4294967295     Z=0, C=0
```

Для CMPX, при установленом воздействии WZ, флаг Z будет равен 1, если он был установлен ранее и *Value1* равно *Value2* + C (используйте WC и WZ в предыдущих инструкциях CMP и CMPX). При указанном воздействии WC, флаг C устанавливается в 1 если *Value1*меньше *Value2* (как величина размером в несколько *long*-ов).

CNT

Регистр: Регистр системного счетчика

```
DAT
    ⟨Метка⟩ ⟨Условие⟩ Инструкция   ОперандПрм, CNT ⟨Воздействия⟩
```

- ***Метка*** – опциональная метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – опциональное условие выполнения инструкции. См. «Общие элементы синтаксиса», стр. 408
- ***Инструкция*** – необходимая ассемблерная инструкция. CNT– регистр «только для чтения», поэтому может использоваться только как операнд-источник.
- ***ОперандПрм*** – это выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра CNT из поля операнда-источника.

Описание

Регистр CNT содержит текущее значение 32-битного глобального Системного Счетчика. Системный счетчик выполняет роль общего источника синхронизации для всех процессоров; он увеличивает свое 32-битное значение на единицу каждый такт Системной Частоты.

CNT является псевдорегистром, с доступом только для чтения; при его использовании в инструкции как операнда-источника, происходит чтение текущего значения Системного Счетчика. Не используйте CNT как операнд-приемник, это приведет лишь к чтению и модификации теневого регистра, чей адрес совпадает с адресом CNT.

CNT обычно используется для определения начального значения переменной в задержках, выполненных при помощи инструкции WAITCNT. В следующем примере производится циклическое действие каждую ¼ секунды. Для дополнительной информации см. Регистры, стр. 499, и Язык Spin CNT, стр.215.

```
DAT
                org 0                           'Reset assembly
pointer
AsmCode         rdlong  Delay, #0               'Get clock frequency
                shr     Delay, #2               'Divide by 4
                mov     Time,  cnt              'Get current time
                add     Time,  Delay            'Adjust by 1/4 second
Loop            waitcnt Time,  Delay            'Wait for 1/4 second
                '<more code here>               'Perform operation
                jmp     #Loop                   'loop back

Delay   res     1
Time    res     1
```

COGID

Инструкция: Получить *ID*-номер текущего процессора.

COGID *Приемник*

Результат: Номер *ID* (0-7) текущего процессора записан в *Приемник*.

- ***Приемник*** (d-поле) – регистр для записи номера *ID* процессора.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0011 1111 ddddddddd ------001	ID = 0	0	Записан	7..22

Вход					Выход		
Приемник	Источник[1]	Z	C	Возд.	Приемник[2]	Z	C
$----_----; -	%0_00000001; 1	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	%0_00000001; 1	-	-	wz wc	$1; 1 .. $7; 7	0	0

[1] **Значение Источника автоматически устанавливается ассемблером в 1, чтобы указать, что это hub-инструкция COGID.**

[2] **Значение Приемника на выходе (его записанное значение) будет от 0 до 7, в зависимости от того, какой cog выполнил инструкцию.**

Описание

COGID возвращает *ID* процессора, который выполнил команду. Инструкция COGID ведет себя аналогично команде языка *Spin* с таким же именем; см. COGID на стр. 217.

При установленном воздействии WZ, флаг Z устанавливается, если *ID* номер равен нолю. Результат записывается в *Приемник* , если не указано воздействие NR.

COGID – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

COGINIT

Инструкция: Запустить или перезапустить процессор, опционально по его номеру *ID*, для выполнения кода Propeller Ассемблера или *Spin*.

COGINIT *Приемник*

Результат: Опционально номер запущенного/перезапущенного *Cog*-а записывается в *Приемник*.

- *Приемник* (d-поле) – регистр, содержащий информацию для начального старта выбранного процессора и опционально являющимся приемником номера *ID* запущенного процессора, если запущен новый *Cog*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0001 1111 ddddddddd ------010	ID = 0	Нет своб. *Cog*-а	Не записан	7..22

Вход					Выход		
Приемник[1]	Источник[2]	Z	C	Возд.	Приемник[3]	Z	C
$0000_0000; 0	%0_00000010; 2	–	–	wr wz wc	$0000_0000; 0[4]	1	0
$0000_0001; 1	%0_00000010; 2	–	–	wr wz wc	$0000_0001; 1[4]	0	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$0000_0000; 0[5]	1	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$1; 1 .. $7; 7[5]	0	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$0000_0007; 7[5]	0	1

[1] Значение регистра Приемника на входе должно состоять из величины регистра PAR (биты 31:18), адреса инструкции (биты 17:4), и номера процессора / нового процессора (биты 3:0).

[2] Источник автоматически устанавливается транслятором в значение 2, чтобы указать, что это - hub-инструкция COGINIT.

[3] Приемник не перезаписывается, если не задано воздействие WR.

[4] Когда в значении Приемника на входе указан признак запуска нового процессора с заданным номером, в значении Приемника на выходе (его записанном значении) будет отображен ID-номер этого процессора; заданный процессор запускается или перезапускается в любом сучае.

[5] Когда в значении Приемника на входе указан признак запуска нового (следующего доступного) процессора, в значении Приемника на выходе (его записанном значении) будет отображен ID-номер запущенного процессора, либо значение 7 (с установленным флагом C), если нет доступных процессоров.

Описание

Инструкция COGINIT выполняется аналогично двум командам *Spin*, COGNEW и COGINIT, совмещенным вместе. Инструкция COGINIT ассемблера может использоваться для запуска нового процессора либо перезапуска уже активного *Cog*-а. Регистр *Приемник* имеет четыре поля, которые определяют, какой из процессоров запущен, где в Основной Памяти начинается его программа, и что будет содержать его регистр PAR. Приведенная ниже таблица описывает эти поля.

Табл. 5-1: Поля Регистра Приемник

31:18	17:4	3	2:0
14-битный адрес для регистра PAR	14-битный адрес кода для загрузки	Новый	*Cog* ID

Первое поле, биты 31:18, будет записано в биты 15:2 регистра PAR запущенного процессора. Все эти 14 бит предназначены для записи старшими битами 16-битного адреса. Аналогично полю *Parameter* аналога команды COGINIT из языка *Spin*, это первое поле регистра *Приемник* используется для передачи 14-битного адреса согласованной ячейки памяти или структуры для запущенного *Cog*-а.

Второе поле, биты 17:4, содержит старшие 14 битов 16-битного адреса, указывающего на необходимую ассемблерную программу для загрузки в *Cog*. Регистры процессора с $000 по $1EF будут последовательно загружены кодом начиная с указанного адреса, регистры специальных функций будут обнулены, и процессор начнет выполнение кода с регистра $000.

Третье поле, бит 3, должно быть установлено в 1, если должен быть запущен новый процессор, либо обнулено, если должен быть запущен или перезапущен выбранный процессор.

Если бит третьего поля установлен (1), *Hub* запустит следующий доступный (неактивный с наименьшим номером) процессор и вернет его номер *ID* в регистре *Приемник* (если указан эффект WR).

Если же бит третьего поля очищен (0), *Hub* запустит или перезапустит процессор, определенный номером в четвертом поле регистра *Приемник*, в битах 2:0.

При указанном воздействии WZ, флаг Z будет установлен (1), если возвращенный номер *ID* процессора равен 0. Если указано воздействие WC, флаг C будет установлен (1), если нет доступных процессоров. При указанном воздействии WR, в регистр *Приемник* записывается *ID*-номер *Cog*-а, запущенного *Hub*-ом, если Вы укажете *Hub*-у найти его.

Не забывайте указывать воздействия WC, WZ, и/или WR при использовании инструкции COGINIT, если Вы хотите, чтобы во флаги и регистр *Приемник* записались результаты.

Запускать код *Spin* из кода Propeller ассемблер не практично, и с использованием этой инструкции мы рекомендуем запускать только код ассемблера.

COGINIT – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

COGSTOP

Инструкция: Остановить процессор по его ID-номеру.

COGSTOP *CogID*

- ***CogID*** (d-поле) – регистр, содержащий ID-номер (0 – 7) останавливаемого cog-а.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0001 1111	ddddddddd	------011	Останов. ID = 0	Нет своб. *Cog*-а	Не записан	7..22

Вход					Выход		
Приемник	Источник[1]	Z	C	Возд.	Приемник[2,3]	Z	C[4]
$0000_0000; 0	%0_00000011; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0005; 5	%0_00000011; 3	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0008; 8[5]	%0_00000011; 3	–	–	wr wz wc	$0000_0000; 0	1	0

[1] **Источник автоматически устанавливается транслятором в значение 3, чтобы указать, что это - hub-инструкция COGSTOP.**
[2] **Приемник не перезаписывается, если не задано воздействие WR.**
[3] **Значение Приемника на выходе (его записанное значение) указывает номер процессора, который был остановлен.**
[4] **Флаг C устанавливается в 1, если перед выполнением инструкции COGSTOP уже были запущены все процессоры.**
[5] **Используются только младшие три бита значения Приемника на входе, поэтому величина 8, к примеру, будет указывать на *cog* номер 0.**

Описание

Инструкция COGSTOP останавливает процессор, *ID*-номер которого находится в регистре *CogID*, переводя этот процессор в спящий режим. В спящем режиме на процессор не подаются импульсы системной частоты, поэтому потребляемая им мощность значительно уменьшается.

COGSTOP – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub*

и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

При заданном воздействии **WZ**, флаг Z будет установлен в 1, если ID остановленного процессора равен нолю (0). При заданном воздействии **WC**, флаг C будет установлен в 1, если перед выполнением этой инструкции были запущены все процессоры. В случае задания воздействия **WR**, в регистр *Приемник* будет записан ID остановленного *COG*-а.

Условия (IF_x)

Каждая инструкция ассемблера Propeller имеет опциональное поле “условие”, которое используется для динамического определения необходимости исполнения инструкции при её достижении во время выполнения программы. Базовый синтаксис инструкций ассемблера Propeller такой:

⟨***Метка***⟩ ⟨***Условие***⟩ ***Инструкция*** ⟨***Воздействия***⟩

Опциональное поле ***Условие*** может содержать одно из 32 условий (см. IF_x Условия), стр. 446), и по умолчанию равно **IF_ALWAYS**, если не указано ни одно из них. Значение **Value** (4-битное), указанное для каждого условия – это значение, используемое в поле **–УСЛ–** в опкоде инструкции при компиляции.

Это свойство инструкций, совместно с правильным использованием опционального поля ***Воздействия***, делают *Propeller*-ассемблер очень мощным языком. К примеру, флаги Z и C могут изменяться как угодно, и в дальнейшем выполнение инструкции будет зависеть от состояний этих флагов.

Когда условие выполнения инструкции вычисляется как **FALSE**, инструкция динамически превращается в **NOP**, занимая 4 такта, но не влияя на флаги и регистры. Таким образом обеспечивается четкая определенность во времени выполнения кода со многими вариантами решений, поскольку при этом используется один и тот же путь (одинаковое время исполнения).

Для дополнительной информации см. IF_x (Условия), на стр. 446.

CTRA, CTRB

Регистр: Регистры управленния Счетчика A и Счетчика B.

```
DAT
   〈Метка〉 〈Условие〉 Инструкция   CTRA ОперандИст, 〈Воздействия〉
DAT
   〈Метка〉 〈Условие〉 Инструкция   ОперандПрм  CTRA, 〈Воздействия〉
DAT
   〈Метка〉 〈Условие〉 Инструкция   CTRB ОперандИст, 〈Воздействия〉
DAT
   〈Метка〉 〈Условие〉 Инструкция   ОперандПрм  CTRB, 〈Воздействия〉
```

Результат: Обновление регистра управления счетчика.

- ***Метка*** – опциональная метка. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистры CTRA и CTRB могут использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандИст*** – выражение, используемое инструкцией для выполнения действия, и при необходимости – записи в регисры CTRA или CTRB в ***ОперандПрм***.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра CTRA либо CTRB из поля операнда-источника.

Описание

CTRA и CTRB – это два из шести регистров (CTRA, CTRB, FRQA, FRQB, PHSA, и PHSB), которые влияют на работу модулей счетчиков процессора. В каждом *cog-е* имеется два одинаковых модуля счетчиков (A и B), которые могут выполнять множество повторяющихся задач. Регистры CTRA и CTRB содержат настройки для соответственно модулей Счетчика A и Счечика B.

CTRA и CTRB – это регистры с доступом для чтения/записи, и могут использоваться в ассемблерной инструкции в полях как операнда-источника, так и опреанда-приемника. В следующем примере Счетчик A настраивается в режим NCO на линии B/B пин 2. См. Регистры, стр. 499, и Язык Spin, секция CTRA, CTRB стр. 239.

```
                mov     ctra, CtrCfg
CtrCfg          long    %0_00100_000_0000000_000000_000_000010
```

DIRA, DIRB

Register: Регистры направления для 32-битных портов A и B.

```
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   DIRA ОперандИст, ⟨Воздействия⟩
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   ОперандПрм  DIRA, ⟨Воздействия⟩
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   DIRB ОперандИст, ⟨Воздействия⟩  (зарезервирован)
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   ОперандПрм  DIRB, ⟨Воздействия⟩ (зарезервирован)
```

Result: Обновление регистра направления.

- ***Метка*** – опциональная метка. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистры CTRA и CTRB могут использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандИст*** – выражение, используемое инструкцией для выполнения действия, и при необходимости – записи в регисры DIRA или DIRB в ***ОперандПрм***.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра DIRA либо DIRB из поля операнда-источника.

Описание

DIRA и DIRB – это два из шести регистров специального назначения (DIRA, DIRB, INA, INB, OUTA and OUTB), которые напрямую влияют на линии B/B. Биты регистров DIRA и DIRB отражают направление каждой из 32 линий соответственно в портах A и B. Регистр DIRB зарезервирован для использования в будущем; в ИМС Propeller P8X32A нет линий порта B, поэтому дальнейшее обсуждение будет ограничено регистром DIRA.

DIRA – это регистр для чтения/записи и может использоваться в ассемблерной инструкции в полях как операнда-источника, так и опреанда-приемника. Сброшенный в ноль бит устанавливает направление соответствующей линии на ввод, а установленный в единицу – на вывод. В следующем примере направление линий от P0 до P3 устанавливаются на вывод.

```
mov     dira, #$0F
```

Для дополнительной информации см. Регистры, стр. 499, и Язык Spin. Секция DIRA, DIRB, стр.249. Имейте в виду, что в языке *Propeller*-асемблера, в отличие от Spin, доступ ко всем 32 битам регистра DIRA осуществляется одновременно, если только не используются инструкции MUXx.

DJNZ

Инструкция: Декремент значения и переход на адрес, если не ноль.

DJNZ *Value*, ⟨#⟩ *Address*

Результат: *Value*-1 записывается в *Value*.

- ***Value*** (d-поле) – регистр для декремента и проверки.
- ***Address*** (s-поле) – регистрis или 9-ти битная константа, значение которой является адресом перехода, если декрементированное *Value* не равно нолю.

–ИНСТР– ZCRI –УСЛ–	–ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
111001 001i 1111	ddddddddd sssssssss	Результат = 0	Беззнаков. заем	Записан	4 или 8

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0002; 2	$----_----; -	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$----_----; -	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$----_----; -	-	-	wz wc	$FFFF_FFFF; -1	0	1

Описание

DJNZ декрементирует *Value* и переходит по *Address*, если результат не равен нолю.

При установленном воздействии WZ, флаг Z устанавливается в 1, если декрементированное значение *Value* равно нолю. При установленном воздействии WC, флаг C устанавливается в 1, если декрементирование значения привело к беззнаковому заему (32-битному переполнению). Результат декрементирования записывается в *Value*, если не указано воздействие NR.

DJNZ требует для исполнения различное количество тактов, в зависимости от того, необходимо ли выполнять переход. При переходе ей необходимо 4 такта, если переход не выполняется, ей необходимо 8 тактов. Поскольку циклы, использующие DJNZ, должны быть быстрыми, она таким образом оптимизирована по скорости.

Воздействия

Каждая инструкция языка Propeller ассемблер имеет опциональное поле “воздействия”, которое приводит к воздействию на флаг либо регистр при ее выполнении. Базовый синтаксис инструкций ассемблера Propeller такой:

⟨***Метка***⟩ ⟨***Условие***⟩ ***Инструкция*** ⟨***Воздействия***⟩

Опциональное поле *Воздействия* может содержать одно или более из четырех воздействий, приведенных ниже. Для любого не указанного рядом с инструкцией воздействия, поведение по умолчанию остается тем, которое задается соответствующими флагами (Z, C, or R) в поле **ZCRI** опкода инструкции.

Табл. 5-2: Воздействия	
Воздействие	**Результат**
WC	Обновляет флаг C, см. **WC**, стр. 532
WZ	Обновляет флаг Z, см. **WZ**, стр. 536
WR	Обновляет регистр приемника, см. **WR**, стр. 533
NR	Регистр приемника не изменяется, см. **NR**, стр. 484

Следующие за инструкцией от одного до трех разделенных пробелами воздействий приводят к влиянию этой инструкции на указанные ресурсы. Например:

```
                and     temp1,    #$20     wc
                andn    temp2,    #$38     wz, nr
  if_c_and_z    jmp     #MoreCode
```

Первая инструкция выполняет побитовое И значения в регистре `temp1` с $20, сохраняет результат в `temp1` и изменяет флаг C для отображения четности результата. Вторая инструкция выполняет побитовое И-НЕ значения регистра `temp2` с $38, изменяет флаг Z в соответствии с тем, равен результат нолю или нет, и не записывает результат в `temp2`. Во время выполнения первой инструкции флаг Z не изменяется. Если бы эти инструкции не включали воздействий **WC** и **WZ**, указанные флаги не изменились бы вовсе. Третья инструкция, с *Условием*, переходит на метку `MoreCode` (не показана), но лишь в том случае, когда оба флага C и Z установлены, иначе инструкция **JMP** будет вести себя как **NOP**.

Использование в инструкциях *Воздействий*, совместно с *Условиями* в последующих инструкциях, позволяет получить намного более мощный код, чем это позволяют сделать обычные инструкции ассемблера. См. Условия (IF_x) на стр. 446 для дополнительной информации.

FRQA, FRQB

Регистр: Регистры частоты Счетчиков A и B.

```
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   FRQA ОперандИст, ⟨Воздействия⟩
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   ОперандПрм  FRQA, ⟨Воздействия⟩
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   FRQB ОперандИст, ⟨Воздействия⟩
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   ОперандПрм  FRQB, ⟨Воздействия⟩
```

Результат: Обновление регистра частоты счетчика.

- ***Метка*** – метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистры CTRA и CTRB могут использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандИст*** – выражение, используемое инструкцией для выполнения действия, и при необходимости – записи в регисры DIRA или DIRB в ***ОперандПрм***.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра DIRA либо DIRB из поля операнда-источника.

Описание

FRQA и FRQB – это два из шести регистров (CTRA, CTRB, FRQA, FRQB, PHSA, и PHSB), которые влияют на работу Модулей Счетчиков процессора. В каждом *cog* имеется два одинаковых модуля счетчиков (A и B), которые могут выполнять множество повторяющихся задач. Регистры FRQA и FRQB содержат величину, которая накапливается в соответствующих регистрах

PHSA и PHSB, в соответствии с режимом данного модуля и входными воздействиями. Для дополнительной информации см. Язык Spin, секцию CTRA, CTRB, стр. 239.

FRQA и FRQB – регистры для чтения/записи и могут использоваться в ассемблерной инструкции в полях как операнда-источника, так и опреанда-приемника. В следующем примере регистр FRQA устанавливается в значение $F. Для дополнительной информации см. Регистры, стр. 499, и Язык Spin, секцию FRQA, FRQB, стр. 256.

```
mov     frqa, #$F
```

FIT

Директива: Проверить, что предыдущие инструкции/данные полностью размещены ниже заданного адреса.

FIT ⟨*Address*⟩

Результат: Ошибка на этапе компиляции, если предыдущие инструкции/данные превысили *Address*-1.

- ***Address*** – это опциональный адрес в ОЗУ *Cog* (0-$1F0), значения которого предыдущий код ассемблера не должен достигнуть. Если *Address* не указан, используется значение $1F0 (адрес первого регистра специальных функций, РСФ).

Описание

Директива FIT при компиляции проверяет текущий указатель адреса процессора и сообщает ошибку, если его значение ниже *Address*-1 либо ниже $1EF (конец ОЗУ общего назначения у *Cog*). Эта директива может использоваться для того, чтобы убедиться, что предыдущие инструкции и данные умещаются в ОЗУ *Cog*, либо в его заданной области. Заметьте: любые инструкции, которые не вмещаются в ОЗУ *Cog* будут игнорироваться при запуске кода ассемблера в процессоре. Обратимся к следующему примеру:

```
DAT
              ORG     492
Toggle        mov     dira, Pin
:Loop         mov     outa, Pin
              mov     outa, #0
              jmp     #:Loop

Pin  long     $1000

              FIT
```

Этот код был искусственно расположен в верхней части пространства ОЗУ *Cog* оператором ORG, что привело к перекрытию кодом первого РСФ ($1F0) и к сообщению об ошибке от директивы FIT при компиляции кода.

HUBOP

Инструкция: Выполнить *Hub* операцию.

HUBOP ***Приемник***, ⟨#⟩ ***Operation***

Результат: Изменяется в зависимости от выполняемой операции.

- ***Приемник*** (d-поле) – регистр, содержащий величину для использования в *Operation*.
- ***Operation*** (s-поле) – регистр или 3-битная константа, указывающая Переключателю операцию для выполнения.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 000i 1111	ddddddddd	sssssssss	Результат = 0	---	Не записан	7..22

<ТАБЛИЦА ИСТИННОСТИ НЕ ПРИВОДИТСЯ, ПОСКОЛЬКУ ОНА РАЗЛИЧНА ДЛЯ КАЖДОЙ ИЗ HUB-ИНСТРУКЦИЙ; СМ. **CLKSET**, **COGID**, **COGINIT**, **COGSTOP**, **LOCKNEW**, **LOCKRET**, **LOCKSET**, И **LOCKCLR**>

Описание

HUBOP - это шаблон для каждой инструкции, выполняемой *Hub*-ом в ИМС Propeller: CLKSET, COGID, COGINIT, COGSTOP, LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR. Инструкции, выполняющие *Hub*-операции, устанавливают поле *Operation* (s-поле в опкоде) в 3-битное значение, которое представляет желаемую операцию (для более детальной информации см. опкод в описании синтаксиса для каждой *Hub*-инструкции). Сама инструкция HUBOP используется очень редко, но в некоторых ситуациях может быть удобной.

HUBOP – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

IF_x

Каждая инструкция ассемблера Propeller имеет опциональное поле “условие”, которое используется для динамического определения необходимости исполнения инструкции при её достижении во время выполнения программы. Базовый синтаксис инструкций ассемблера Propeller такой:

⟨***Метка***⟩ ⟨***Условие***⟩ ***Инструкция*** ⟨***Воздействия***⟩

Опциональное поле ***Условие*** может содержать одно из 32 условий (см. IF_x Условия), стр. 446), и по умолчанию равно **IF_ALWAYS**, если не указано ни одно из них. Значение **Value** (4-битное), указанное для каждого условия – это значение, используемое в поле **–УСЛ–** в опкоде инструкции при компиляции.

Это свойство инструкций, совместно с правильным использованием опционального поля ***Воздействия***, делают *Propeller*-ассемблер очень мощным языком. К примеру, флаги Z и C могут изменяться как угодно, и в дальнейшем выполнение инструкции будет зависеть от состояний этих флагов. Например:

```
            test   _pins, #$20      wc
            and    _pins, #$38
            shl    t1, _pins
            shr    _pins, #3
            movd   vcfg, _pins
if_nc       mov    dira, t1
if_nc       mov    dirb, #0
if_c        mov    dira, #0
if_c        mov    dirb, t1
```

Первая инструкция, `test _pins, #$20 wc`, выполняет свое действие и задает состояние флага C, поскольку указано воздействие **WC**. Следующие четыре инструкции выполняют действия, которые могут изменить этот флаг, но они не влияют на него, поскольку воздействия **WC** для них не указаны. Это значит, что состояние флага C сохранилось с момента его задания в первой инструкции. Последние четыре инструкции выполняются последовательно в зависимости от состояния флага C, которое было задано на пять инструкций раньше. Среди последних четырех инструкций, первые две инструкции `mov` имеют условия `if_nc`, которые указывают, что выполнение возможно только если флаг C = 0 (“if not C”). Последние же две инструкции `mov` имеют условия `if_c`, которые указывают, что

выполнение возможно только при C = 1 (“if C”). В этом случае две пары инструкций mov выполняются во взаимо-исключающей манере.

Когда условие выполнения инструкции вычисляется как FALSE, инструкция динамически превращается в NOP, занимая 4 такта, но не влияя на флаги и регистры. Таким образом обеспечивается четкая определенность во времени выполнения кода со многими вариантами решений, поскольку при этом используется один и тот же путь (одинаковое время исполнения).

Табл. 5-3: Условия

Условие	Выполнение инструкции	Знач.	Синонимы
IF_ALWAYS	всегда	1111	
IF_NEVER	никогда	0000	
IF_E	если равно (Z = 1)	1010	IF_Z
IF_NE	если не равно (Z = 0)	0101	IF_NZ
IF_A	если больше (!C & !Z = 1)	0001	IF_NC_AND_NZ –и– IF_NZ_AND_NC
IF_B	если меньше (C = 1)	1100	IF_C
IF_AE	если больше или равно (C = 0)	0011	IF_NC
IF_BE	если меньше или равно (C | Z = 1)	1110	IF_C_OR_Z –и– IF_Z_OR_C
IF_C	если C установлен	1100	IF_B
IF_NC	если C сброшен	0011	IF_AE
IF_Z	если Z установлен	1010	IF_E
IF_NZ	если Z сброшен	0101	IF_NE
IF_C_EQ_Z	если C равен Z	1001	IF_Z_EQ_C
IF_C_NE_Z	если C не равен Z	0110	IF_Z_NE_C
IF_C_AND_Z	если C устан. и Z устан.	1000	IF_Z_AND_C
IF_C_AND_NZ	если C устан. и Z сброш.	0100	IF_NZ_AND_C
IF_NC_AND_Z	если C сброш. и Z устан.	0010	IF_Z_AND_NC
IF_NC_AND_NZ	если C сброш. и Z сброш.	0001	IF_A –и– IF_NZ_AND_NC
IF_C_OR_Z	если C устан. или Z устан.	1110	IF_BE –и– IF_Z_OR_C
IF_C_OR_NZ	если C устан. или Z сброш.	1101	IF_NZ_OR_C
IF_NC_OR_Z	если C сброш. или Z устан.	1011	IF_Z_OR_NC
IF_NC_OR_NZ	если C сброш. или Z сброш	0111	IF_NZ_OR_NC
IF_Z_EQ_C	если Z равен C	1001	IF_C_EQ_Z

IF_Z_NE_C	если Z не равен C	0110	IF_C_NE_Z
IF_Z_AND_C	если Z устан. и C устан.	1000	IF_C_AND_Z
IF_Z_AND_NC	если Z устан. и C сброш.	0010	IF_NC_AND_Z
IF_NZ_AND_C	если Z сброш. и C устан.	0100	IF_C_AND_NZ
IF_NZ_AND_NC	если Z сброш. и C сброш.	0001	IF_A –и– IF_NC_AND_NZ
IF_Z_OR_C	если Z устан. или C устан.	1110	IF_BE –и– IF_C_OR_Z
IF_Z_OR_NC	если Z устан. или C сброш.	1011	IF_NC_OR_Z
IF_NZ_OR_C	если Z сброш. или C устан.	1101	IF_C_OR_NZ
IF_NZ_OR_NC	если Z сброш. или C сброш	0111	IF_NC_OR_NZ

INA, INB

Регистр: Входные регистры 32-битных портов A и B.

```
DAT
  〈Метка〉 〈Условие〉 Инструкция  ОперандПрм  INA, 〈Воздействия〉
DAT
  〈Метка〉 〈Условие〉 Инструкция  ОперандПрм  INB, 〈Воздействия〉 (зарезервировано)
```

- ***Метка*** – метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистры CTRA и CTRB могут использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра DIRA либо DIRB из поля операнда-источника.

Описание

INA и INB – это два из шести регистров специального назначения (DIRA, DIRB, INA, INB, OUTA и OUTB), которые непосредственно влияют на линии B/B. Биты регистров INA и INB показывают текущие логические состояния на каждой из 32-х линий B/B соответственно портов A и B. Регистр INB зарезервирован для использования в

будущем; в ИМС Propeller P8X32A нет линий B/B порта B, поэтому дальнейшее обсуждение будет ограничено регистром **INA**.

INA представляет собой псевдо-регистр только для чтения; при использовании в качестве операнда-источника он читает текущее логическое состояние соответствующих лиинний B/B. Не следует использовать регистр **INA** как операнд-приемник; это лишь приведет к чтению и модификации теневого регистра, адрес которого занимает **INA**.

В регистре **INA**, сброшенное в ноль состояние бита указывает на то, что напряжение на соответствующей линии B/B близко к потенциалу земли, а установленное в единицу – что потенциал близок к напряжению питания VDD (3.3 Вольта). В следующем примере состояния линий B/B с P0 по P31 записывается в регистр с именем Temp.

```
mov     Temp, ina
```

Для дополнительной информации см. Регистры, стр. 446, и Язык Spin, секцию **INA**, **INB**, стр. 263. Имейте в виду, что в языке *Propeller*-асемблера, в отличие от Spin, доступ ко всем 32 битам регистра **INA** осуществляется одновременно, если только не используются инструкции **MUX**x.

JMP

Инструкция: Переход по адресу.

JMP ⟨#⟩ ***Address***

- ***Address*** (s-поле) – регистр или 9-битная константа, значение которой является адресом для перехода.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
010111 000i 1111	---------	ssssssss	Результат = 0	---	Не записан	4

Вход					Выход		
Приемник[1]	Источник	Z	C	Возд.	Приемник[2]	Z	C[3]
s----_----; -	s----_----; -	-	-	wr wz wc	31:9 неизм., 8:0 = PC+1	0	1

[1] **При типичном использовании JMP, Приемник обычно игнорируется, однако если указано воздействие WR, инструкция JMP становится инструкцией JMPRET, и s-поле Приемника (младшие 9 бит)перезаписывается адресом возврата (PC+1).**

[2] **Приемник не перезаписывается, если не задано воздействие WR.**

[3] Флаг C устанавливается в 1, кроме случая когда PC+1 равен 0, что очень маловероятно, поскольку при этом JMP будет выполняться с самого начала cog RAM ($1FF, регистр специальных функций VSCL).

Описание

JMP устанавливает счетчик команд (PC) в значение *Address*, что приводит к переходу на этот адрес в ОЗУ *Cog*. Команда JMP тесно связана с командами CALL, JMPRET и RET; на самом деле все они имеют одинаковый опкод, но с различными значениями r- и i-поля, а также изменяющимися под управлением ассемблера либо пользователя величинами d- и s-полей.

Условные переходы

Условные переходы, как и условное выполнение любых инструкций, выполняются путем задания условия в виде: IF_x, предваряющего инструкцию. Для дополнительной информации см. «Условия (IF_x)» на стр. 446.

Символ «Здесь» '$'

Символ «Здесь», $, представляет текущий адрес. Во время разработки и отладки, символ «Здесь» '$' зачастую используется в поле *Адреса* инструкции JMP (т.е.: JMP #$) с целью задать в этом месте выполнение бесконечного цикла. Он также может использоваться для задания относительного перехода на несколько инструкций назад либо вперед, например:

```
Toggle          mov     dira, Pin     'Set I/O pin to output
                xor     outa, Pin     'Toggle I/O pin's state
                jmp     #$-1          'Loop back endlessly
```

В последней инструкции, JMP #$-1, происходит переход назад на предпоследнюю инструкцию (т.е.: 'здесь' минус 1).

JMPRET

Инструкция: Переход на заданный адрес с дальнейшим "возвратом" на другой адрес.

JMPRET *RetInstAddr*, ⟨#⟩ *DestAddress*

Результат: PC + 1 записывается в s-поле регистра, указанного в d-поле.

- ***RetInstAddr*** (d-поле) – регистр, в котором сохраняется адрес возврата (PC + 1); часто это адрес соответствующей инструкции RET или JMP, выполняемой подпрограммой *DestAddress*.

- ***DestAddress*** (s-поле) – регистр или 9-битная константа, значение которого – адрес для перехода.

–ИНСТР– ZCRI –УСЛ–	–ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
010111 001i 1111	dddddddddd sssssssss	Результат = 0	---	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C[2]
s----_----; -	s----_----; -	-	-	wz wc	31:9 неизм., 8:0 = PC+1	0	1

[1] **s-поле приемника (младшие 9 бит) переписываются адресом возврата (PC+1) во время выполнения.**

[2] **Флаг C устанавливается в 1, кроме случая когда PC+1 равен 0, что очень маловероятно, поскольку при этом JMPRET будет выполняться с самого начала cog RAM ($1FF, регистр специальных функций VSCL).**

Описание

JMPRET (перейти и вепрнуться) обеспечивает механизм для “вызова” других подпрограмм и дальнейшего возврата на инструкцию, следующую после JMPRET. Для вызовов обычных подпрограмм, вместо JMPRET следует использовать инструкцию CALL, поскольку она выполняет действие, схожее с командами, близкими по названию в других процессорах. Инструкция JMPRET предоставляет дополнительное преимущество перед обычным “вызовом”, позволяющее выполнять несколько подпрограмм в стиле переключения задач.

В ИМС Propeller не используется аппаратный стек вызовов, поэтому адрес возврата операции типа вызова должен сохраняться в ином виде. Во время исполнения, инструкция JMPRET сохраняет адрес следующей инструкции (PC+1) в поле источника (s-field) регистра по адресу *RetInstAddr,* после чего осуществляет переход на *DestAddress.*

Если регистр по адресу *RetInstAddr* содержит инструкцию RET либо JMP, и она в дальнейшем будет запущена подпрограммой *DestAddress*, то такое поведение одинаково с инструкцией CALL: сохраняется адрес возврата, выполняется пдпрограмма *DestAddress*, и, в завершение, управление возвращается инструкции, следующей за JMPRET. Для дополннительной информации см. CALL на стр. 427.

Если же её использовать немного по-другому, инструкция JMPRET может помочь в реализации многозадачности в единственном потоке. Это достигается назначением набора регистров для хранения различных адресов перехода и возврата, и указания этих регистров в полях *RetInstAddr* и *DestAddress*. Например:

```
Initialize      mov     Task2, #SecondTask      'Initialize 1st Dest.

FirstTask       <start of first task>
```

```
                ...
                jmpret  Task1, Task2            'Give 2nd task cycles
                <more first task code>
                ...
                jmpret  Task1, Task2            'Give 2nd task cycles
                jmp       #FirstTask            'Loop first task

SecondTask      <start of second task>
                ...
                jmpret  Task2, Task1            'Give 1st task cycles
                <more second task code>
                ...
                jmpret  Task2, Task1            'Give 1st task cycles
                jmp     #SecondTask       'Loop second task

Task1           res     1                       'Declare task address
Task2           res     1                       'storage space
```

В приведенном примере имеется две подпрограммы: FirstTask и SecondTask, которые работают как отдельные задачи в рамках потока одного процессора. Действия, выполняемые каждой из задач, относительно не важны; они могут быть схожими либо различными. Регистры Task1 и Task2 – это двойные слова, которые объявлены в конце кода и предназначены для хранения адресов перехода и возврата, что способствует реализации переключения исполнения между двумя задачами.

Первая инструкция, mov Task2,#SecondTask, сохраняет адрес SecondTask в регистре Task2. Это настраивает регистры задач для первого события переключения.

После старта задачи FirstTask, она выполняет какие-то действия, обозначенные как "…" и достигает первой инструкции **JMPRET**, jmpret Task1,Task2. Здесь **JMPRET** вначале сохраняет адрес возврата (PC + 1, адрес кода <more first task code>) в s-поле регистра Task1, после чего переходит по адресу, записанному в Task2. Поскольку мы проинициализировали Task2 на точку входа SecondTask, то теперь будет запущена вторая задача.

Код SecondTask выполняет какие-то действия, обозначенные как "…" и достигает другой инструкции **JMPRET**, jmpret Task2,Task1. Заметьте, что это похоже на инструкцию **JMPRET** из FirstTask, за исключением того, что порядок Task1 и Task2 изменен на противоположный. Эта инструкция **JMPRET** сохраняет адрес возврата (PC + 1, адрес кода <more second task code>) в s-поле регистра Task2, после чего переходит по адресу, записанному в Task1. Поскольку Task1 содержит адрес кода <more first task

code>, который был записан предыдущей инструкцией JMPRET, то теперь выполнение переключится назад к первой задаче FirstTask, начиная со строки <more first task code>.

Выполнение продолжает переключаться вперед-назад между задачами FirstTask и SecondTask, где бы не встретилась инструкция JMPRET, возвращаясь точно в то место, откуда было осуществлено переключение в предыдущей задаче. Каждая инструкция JMPRET перезаписывает использованный ранее адрес назначения новым адресом возврата, после чего осуществляет переход на новый адрес назначения, то есть адрес возврата из прошлого раза.

Такой многозадачный подход может использоваться различными способами. К примеру, удаление одной из JMPRET из SecondTask приведет к тому, что FirstTask будет получать меньше процессорных тактов на единицу времени. Подобные методы могут использоваться для предоставления бОльшего количества процессорных тактов критическим по времени задачам, либо наиболее критическим частям задач. Можно также использовать большее количество регистров Task для переключения между тремя и более подпрограммами в рамках того же процессора. Время работы каждой задачи всегда фиксировано и определяется тем, где установлены инструкции JMPRET и к каким задачам они относятся.

Отметьте, что состояния флагов C и Z не изменились и не сохраняются между событиями переключения. Поэтому важно переключаться между задачами только когда состояния флагов уже не важны, либо есть уверенность, что они не изменятся перед возвратом выполнения.

Инструкция JMPRET функционально шире инструкции CALL; на самом деле JMPRET имеет тот же опкод, что и у CALL, но с i- и d-полями, задаваемыми разработчиком, а не ассемблером. Для дополнительной информации см. CALL на стр. 427.

Адрес взврата (PC + 1) записывается в поле-источник (s-поле) регистра *RetInstAddr*, если не указано воздействие NR. Естественно, указание NR для инструкции JMPRET не рекомендуется, поскольку оно превратит ее в инструкцию JMP или RET.

LOCKCLR

Инструкция: Сбросить бит защиты в FALSE и получить его предыдущее состояние.

LOCKCLR *ID*

Результат: Опционально, предыдущее состояние бита защиты записывается в флаг C.

- ***ID*** (d-поле) – регистр, содержащий *ID*-номер (0 – 7) бита защиты для сброса.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0001 1111	ddddddddd	------111	ID = 0	Предыд. сост. бита защиты	Не записан	7..22

Вход					Выход		
Приемник	Источник[1]	Z	C	Возд.	Приемник[2,3]	Z	C
$0000_0005; 5	%0_00000111; 7	–	–	wr wz wc	$0000_0005; 5	0	1[4]
$0000_0005; 5	%0_00000111; 7	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0000; 0	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	1[4]
$0000_0000; 0	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0008; 8[5]	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	0

[1] **Источник автоматически устанавливается транслятором в значение 7, чтобы указать, что это - hub-инструкция LOCKCLR.**

[2] **Приемник не перезаписывается, если не задано воздействие WR.**

[3] **Значение Приемника на выходе (его записанное значение) отображает ID-номер сброшенного бита защиты.**

[4] **Флаг C отображает предыдущее состояние бита защиты; в этих случаях бит защиты был предварительно установлен предыдущей инструкцией LOCKSET (не показано). В следующем примере флаг C сбрасывается, поскольку бит защиты был только что очищен в предыдущем примере.**

[5] **Используются только младшие три бита значения Приемника на входе, поэтому величина 8, к примеру, будет указывать на *cog* номер 0.**

Описание

LOCKCLR – одна из четырех инструкций (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемых для управления ресурсами, определяемыми пользователем и считающимися взаимоисключающими. LOCKCLR сбрасывает бит защиты, задаваемый регистром *ID,* в ноль(0) и возвращает предыдущее состояние этого бита в флаге C, если задано воздействие WC. Инструкция LOCKCLR выполняется аналогично команде LOCKCLR языка *Spin*; см. LOCKCLR на стр. 266.

При установленом воздействии WZ, флаг Z устанавливается в 1 в том случае, если номер ID очищенного бита защиты равен нолю. При заданном воздействии WC, флаг C становится равен предыдущему сстоянию бита защиты. Установка эффекта WR приводит к записи номера ID очищенного бита в *ID*.

LOCKCLR – Справочник по языку ассемблер

LOCKCLR – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

LOCKNEW

Инструкция: Проверить новый бит защиты и получить его *ID*-номер.

LOCKNEW *NewID*

Результат: *ID*-номер (0-7) нового бита защиты записывается в *NewID*.

- ***NewID*** (d-поле) – регистр, в котором записан *ID* нового бита защиты.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0011 1111	ddddddddd	------100	ID = 0	Нет своб.бита защиты	Записан	7..22

Вход					Выход		
Приемник	Источник[1]	Z	C	Возд.	Приемник	Z	C
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0001; 1	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0002; 2	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0003; 3	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0004; 4	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0005; 5	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0006; 6	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0007; 7	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0007; 7	0	1

[1] **Источник автоматически устанавливается транслятором в значение 4, чтобы указать, что это - hub-инструкция LOCKNEW.**

Описание

LOCKNEW – одна из четырех инструкций (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемых для управления ресурсами, определяемыми пользователем и считающимися взаимоисключающими. LOCKNEW получает свободный бит защиты от *Hub*, и предоставляет *ID*-номер этого бита. Инструкция LOCKNEW выполняется аналогично команде LOCKNEW языка *Spin*; см. LOCKNEW на стр. 268.

При указанном воздействии WZ, флаг Z устанавливается в 1, если возвращаемый *ID*-номер равен нолю (0). При установленном воздействии WC флаг C устанавливается в 1, если для использования не было доступно ни одного свободного бита защиты. *ID*-номер нового бита защиты записывается в регистр *NewID*, если не установлено воздействие NR.

LOCKNEW – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

LOCKRET

Инструкция: Освободить бит защиты `lock` для дальнейших запросов на новый.

LOCKRET ***ID***

- ***ID*** (d-поле) – регистр, содержащий *ID*-номер (0 – 7) возвращаемого в очередь бита защиты `lock`.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0001 1111 ddddddddd ------101	ID = 0	Нет своб.бита защиты	Не записан	7..22

Вход					Выход		
Приемник	Источник[1]	Z	C	Возд.	Приемник[2]	Z	C[3]
$0000_0000; 0	%0_00000101; 5	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0005; 5	%0_00000101; 5	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0008; 8[4]	%0_00000101; 5	–	–	wr wz wc	$0000_0000; 0	1	0

[1] **Источник автоматически устанавливается транслятором в значение 5, чтобы указать, что это - hub-инструкция LOCKRET.**

[2] **Приемник не перезаписывается, если не задано воздействие WR.**

[3] **Флаг C устанавливается в 1, если перед выполнением инструкции LOCKRET уже были заняты все биты защиты.**

[4] **Используются только младшие три бита значения Приемника на входе, поэтому величина 8, к примеру, будет указывать на *cog* номер 0.**

Описание

LOCKRET – одна из четырех инструкций (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемых для управления ресурсами, определяемыми пользователем и считающимися взаимоисключающими. LOCKRET возвращает бит защиты lock, по его *ID*, назад в *Hub* в очередь, после чего этот бит может использоваться другими процессорами. Инструкция LOCKRET выполняется аналогично команде LOCKRET языка *Spin*; см. LOCKRET на стр. 271.

При указанном воздействии WZ, флаг Z устанавливается в 1, если номер ID возвращаемого бита защиты равен 0. Заданное воздействие WC устанавливает флаг C в 1, если перед выполнением этой инструкции все биты уже были назначены. Если указано воздействие WR, то ID-номер возвращаемого бита защиты записывается в *ID*.

LOCKRET – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

LOCKSET

Инструкция: Установить бит защиты в TRUE и получить его предыдущее состояние.

LOCKSET *ID*

Результат: Опционально предыдущее состояние бита защиты lock записано в флаг C.

- ***ID*** (d-поле) – регистр, содержащий *ID*-номер (0 – 7) бита lock для установления.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
000011 0001 1111 ddddddddd ------110	ID = 0	Предыд. сост.бита защиты	Не записан	7..22

Вход					Выход		
Приемник	Источник[1]	Z	C	Возд.	Приемник[2,3]	Z	C
$0000_0005; 5	%0_00000110; 6	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0005; 5	%0_00000110; 6	–	–	wr wz wc	$0000_0005; 5	0	1[4]
$0000_0000; 0	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0000; 0	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	1[4]
$0000_0008; 8[5]	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	1[4]

[1] Источник автоматически устанавливается транслятором в значение 6, чтобы указать, что это - hub-инструкция LOCKSET.
[2] Приемник не перезаписывается, если не задано воздействие WR.
[3] Значение Приемника на выходе (его записанное значение) отображает ID-номер установвленного бита защиты.
[4] Флаг C отображает предыдущее состояние бита защиты; в этих случаях бит защиты уже был установлен в предыдущем примере.
[5] Используются только младшие три бита значения Приемника на входе, поэтому величина 8, к примеру, будет указывать на *cog* номер 0.

Описание

LOCKSET – одна из четырех инструкций (LOCKNEW, LOCKRET, LOCKSET, и LOCKCLR), используемых для управления ресурсами, определяемыми пользователем и считающимися взаимоисключающими. LOCKSET устанавливает бит защиты lock, заданный в регистре *ID* в единицу (1) и возвращает его предыдущее состояние в флаге C, если указано воздействие WC. Инструкция LOCKSET выполняется аналогично команде LOCKSET языка *Spin*; см. LOCKSET на стр. 272.

При указанном воздействии WZ, флаг Z устанавливается в 1, если номер ID устанавливаемого бита защиты равен 0. Заданное воздействие WC устанавливает флаг C равным предыдущему значению бита защиты. Если указано воздействие WR, то ID-номер устанавливаемого бита защиты записывается в *ID*.

LOCKSET – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

MAX

Инструкция: Ограничить по максимуму беззнаковую величину к другой беззнаковой величине.

MAX *Value1*, ⟨#⟩ *Value2*

Результат: Меньшее из беззнаковых *Value1* и *Value2* сохраняется в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину для сравнения с *Value2* и являющийся приемником, в который записывается меньшая из величин.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *Value1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
010011 001i 1111	ddddddddd	sssssssss	S = 0	Беззнаковое (D < S)	Записан	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

[1] **Источник и Приемник рассматриваются как величины без знака.**

Описание

MAX сравнивает две беззнаковые величины *Value1* и *Value2*, и сохраняет меньшую в регистр *Value1*, ограничивая таким образом *Value1* по максимуму до *Value2*.

При установленном воздействии WZ, флаг Z устанавливается в 1, если *Value2* равно 0. При указанном воздействии WC, флаг C устанавливается в 1, если беззнаковое *Value1* меньше беззнакового *Value2*. Меньшее из двух значений записывается в *Value1,* если не установлено воздействие NR.

MAXS

Инструкция: Ограничить по максимуму величину со знаком к другой величине со знаком.

MAXS *SValue1*, ⟨#⟩ *SValue2*

Результат: Меньшее из знаковых *SValue1* и *SValue2* сохраняется в *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину для сравнения с *SValue2* и являющийся приемником, в который записывается меньшая из величин.
- ***SValue2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *SValue1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
010001 001i 1111	ddddddddd	sssssssss	S = 0	Знаковое (D < S)	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$FFFF_FFFF; -1	–	–	wz wc	$FFFF_FFFF; -1	0	0
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$FFFF_FFFF; -1	$0000_0001; 1	–	–	wz wc	$FFFF_FFFF; -1	0	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

Описание

MAXS сравнивает две знаковые величины *SValue1* и *SValue2*, и сохраняет меньшую в регистр *SValue1*, ограничивая таким образом *SValue1* по максимуму до *SValue2*.

При установленном воздействии WZ, флаг Z устанавливается в 1, если *SValue2* равно 0. При указанном воздействии WC, флаг C устанавливается в 1, если знаковое *SValue1* меньше знакового *SValue2*. Меньшее из двух значений записывается в *Value1,* если не установлено воздействие NR.

MIN

Инструкция: Ограничить по минимуму беззнаковую величину к другой беззнаковой величине.

MIN *Value1*, ⟨#⟩ *Value2*

Результат: Большее из беззнаковых *Value1* и *Value2* сохраняется в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину для сравнения с *Value2* и являющийся приемником, в который записывается большая из величин.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *Value1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
010010 001i 1111	ddddddddd	sssssssss	S = 0	Беззнаковое (D < S)	Записан	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 2	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0001; 1	1	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1

[1] **Источник и Приемник рассматриваются как величины без знака.**

Описание

MIN сравнивает две беззнаковые величины *Value1* и *Value2*, и сохраняет большую в регистр *Value1*, ограничивая таким образом *Value1* по минимуму до *Value2*.

При установленном воздействии WZ, флаг Z устанавливается в 1, если *Value2* равно 0. При указанном воздействии WC, флаг C устанавливается в 1, если беззнаковое *Value1* меньше беззнакового *Value2*. Большее из двух значений записывается в *Value1,* если не установлено воздействие NR.

MINS

Инструкция: Ограничить по минимуму величину со знаком к другой величине со знаком.

MINS *SValue1*, ⟨#⟩ *SValue2*

Результат: Большее из знаковых *SValue1* и *SValue2* сохраняется в *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину для сравнения с *SValue2* и являющийся приемником, в который записывается большая из величин.
- ***SValue2*** (s-поле) регистр или 9-битная константа, величина которого сравнивается с *SValue1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
010000 001i 1111 ddddddddd sssssssss	S = 0	Знаковое (D < S)	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0002; 2	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0001; 1	1	0
$0000_0001; 1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1
$FFFF_FFFF; -1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1

Описание

MINS сравнивает две знаковые величины *SValue1* и *SValue2*, и сохраняет большую в регистр *SValue1*, ограничивая таким образом *SValue1* по минимуму до *SValue2*.

При установленном воздействии WZ, флаг Z устанавливается в 1, если *SValue2* равно 0. При указанном воздействии WC, флаг C устанавливается в 1, если знаковое *SValue1* меньше знакового *SValue2*. Большее из двух значений записывается в *Value1,* если не установлено воздействие NR.

MOV

Инструкция: Установить значение в регистре.

MOV *Приемник*, ⟨#⟩ *Value*

Результат: *Value* сохраняется в *Приемник*.

- ***Приемник*** (d-поле) – регистр, в котором сохраняется *Value*.

- ***Value*** (s-поле) регистр или 9-битная константа, величина которого сохраняется в *Приемник*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
101000 001i 1111	ddddddddd	sssssssss	Результат = 0	S[31]	Written	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0

Описание

MOV копирует, либо сохраняет значение *Value* в *Приемник*.

При установленном воздействии WZ флаг Z устанавливается в 1, если значение *Value* равно нолю. При указанном воздействии WC, флаг C устанавливается в значение MSB величины *Value*. Результат записывается в *Приемник*, если не указано воздействие NR.

MOVD

Инструкция: Установить в заданное значение поле приемника в регистре.

MOVD *Приемник*, ⟨#⟩ *Value*

Результат: *Value* сохраняется в d-поле (биты 17..9) регистра *Приемник*.

- ***Приемник*** (d-поле) – регистр, d-поле (биты 17..9) которого устанавливаются в значение *Value*.
- ***Value*** (s-поле) регистр или 9-битная константа, величина которого сохраняется в d-поле *Приемник*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
010101 001i 1111	ddddddddd	sssssssss	Результат = 0	---	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	-	-	wz wc	$0003_FE00; 261632	0	1

$FFFF_FFFF; -1	$0000_01FF; 511	-	-	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$0000_0000; 0	-	-	wz wc	$FFFC_01FF; -261633	0	0

Описание

MOVD копирует 9-битное значение *Value* в d-поле регистра *Приемник* (поле приемника), биты 17..9. Остальные биты регистра *Приемник* не изменяются. Эта инструкция удобна для установки значений определенных регистров, таких как CTRA и VCFG, а также для обновления полей приемника инструкций в самомодифицируемом коде.

В самомодифицируемом коде следите, чтобы еще как минимум одна инструкция находилась между инструкциями MOVD и той, которую MOVD модифицирует. Это даст процессору время для записи результата выполнения MOVD до того, как он прочтет и запустит ее на выполнение; в противном случае, будет прочитана и выполнена еще немодифицированная инструкция.

При установленном воздействии WZ флаг Z устанавливается в 1, если *Value* равно нолю. Результат записывается в *Приемник*, если не указано воздействие NR.

MOVI

Инструкция: Установить поля инструкции и воздействий регистра в заданные значения.

MOVI *Приемник*, ⟨#⟩ *Value*

Результат: *Value* сохраняется в i- поле и поле воздействий регистра *Приемник* (биты 31..23).

- ***Приемник*** (d-поле) – регистр, в поля инструкции и воздействий которого (биты 31..23) записывается значение *Value*.
- ***Value*** (s-поле) – регистр или 9-битная константа, величина которого сохраняется в полях инструкции и воздействий регистра *Приемник*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
010110 001i 1111	ddddddddd	sssssssss	Результат = 0	---	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	-	-	wz wc	$FF80_0000; -8388608	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	-	-	wz wc	$FFFF_FFFF; -1	0	0

$FFFF_FFFF; -1	$0000_0000; 0	-	-	wz wc	$007F_FFFF; 8388607	0	0

Описание

MOVI копирует 9-битное значение *Value* в поля инструкции и воздействий регистра *Приемник* (биты 31..23). Остальные биты регистра *Приемник* не изменяются. Эта инструкция удобна для установки значений определенных регистров, таких как CTRA и VCFG, а также для обновления полей приемника инструкций в самомодифицирующемся коде.

В самомодифицируемом коде следите, чтобы еще как минимум одна инструкция находилась между инструкциями MOVI и той, которую MOVI модифицирует. Это даст процессору время для записи результата выполнения MOVI до того, как он прочтет и запустит ее на выполнение; в противном случае, будет прочитана и выполнена еще немодифицированная инструкция.

При установленном воздействии WZ флаг Z устанавливается в 1, если *Value* равно нолю. Результат записывается в регистр *Приемник,* если не указано воздействие NR.

MOVS

Инструкция: Установить поле источника в регистре в заданное значение.

MOVS *Приемник*, ⟨#⟩ *Value*

Результат: *Value* сохраняется в s-поле (биты 8..0) регистра *Приемник*.

- ***Приемник*** (d-поле) – регистр, поле источника в котором (биты 8..0) устанавливается равным значению *Value*.
- ***Value*** (s-поле) регистр или 9-битная константа, величина которого сохраняется в поле источника в регистре *Приемник*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
010100 001i 1111 ddddddddd sssssssss	Результат = 0	---	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	-	-	wz wc	$0000_01FF; 511	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	-	-	wz wc	$FFFF_FFFF; -1	0	0

$FFFF_FFFF; -1	$0000_0000; 0	-	-	wz wc	$FFFF_FE00; -512	0	0

Описание

MOVS копирует 9-битное значение *Value* в поле источника (s-поле, биты 8..0) регистра *Приемник*. Остальные биты регистра *Приемник* остаются без изменения. Эта инструкция удобна для установки значений определенных регистров, таких как CTRA и VCFG, а также для обновления полей приемника инструкций в самомодифицирующемся коде.

В самомодифицируемом коде следите, чтобы еще как минимум одна инструкция находилась между инструкциями MOVS и той, которую MOVS модифицирует. Это даст процессору время для записи результата выполнения MOVS до того, как он прочтет и запустит ее на выполнение; в противном случае, будет прочитана и выполнена еще немодифицированная инструкция.

При установленном воздействии WZ флаг Z устанавливается в 1, если *Value* равно нолю. Результат записывается в регистр *Приемник,* если не указано воздействие NR.

MUXC

Инструкция: Установить отдельные биты величины в состояние флага C.

MUXC ***Приемник***, ⟨#⟩ ***Mask***

Результат: Биты *Приемник*, указанные в *Mask*, устанавливаются в значение C.

- ***Приемник*** (d-поле) – регистр, биты которого, заданные в *Mask*, устанавливаются в состояние флага C.
- ***Mask*** (s-поле) регистр или 9-битная константа, величина которого содержит установленные в 1 биты для каждого бита в *Приемник*, устанавливаемого в состояние флага C.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
011100 001i 1111	ddddddddd	sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	1

$0000_0000; 0	$0000_0003; 3	-	0	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	-	1	wz wc	$0000_0003; 3	0	0
$AA55_2200; -1437261312	$1234_5678; 305419896	-	0	wz wc	$A841_2000; -1472126976	0	0
$AA55_2200; -1437261312	$1234_5678; 305419896	-	1	wz wc	$BA75_7678; -1166707080	0	1
$AA55_2200; -1437261312	$FFFF_FFFF; -1	-	0	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1437261312	$FFFF_FFFF; -1	-	1	wz wc	$FFFF_FFFF; -1	0	0

Описание

MUXC устанавливает каждый из битов регистра *Приемник*, соответствующий взведенному в 1 биту в *Mask*, в состояние флага C. Все биты *Приемник*, не указанные единицами в маске *Mask*, остаются без изменения. Эта инструкция удобна для установки либо сброса групп или отдельных битов в существующем значении.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если в результате значение *Приемник* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если в результирующем значении *Приемник* содержится нечетное количество взведенных в 1 битов. Результат записывается в *Приемник*, если не указано воздействие NR.

MUXNC

Инструкция: Установить отдельные биты величины в состояние !C.

MUXNC ***Приемник***, ⟨#⟩ ***Mask***

Результат: Биты *Приемник*, указанные в *Mask*, устанавливаются в значение !C.

- ***Приемник*** (d-поле) – регистр, биты которого, заданные в *Mask*, устанавливаются в состояние, противоположное флагу C.
- ***Mask*** (s-поле) регистр или 9-битная константа, величина которого содержит установленные в 1 биты для каждого бита в *Приемник*, устанавливаемого в состояние, противоположное флагу C.

–ИНСТР– ZCRI	–УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
011101 001i	1111	ddddddddd	sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0001; 1	-	0	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	-	1	wz wc	$0000_0000; 0	1	0

$0000_0000; 0	$0000_0003; 3	-	0	wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0003; 3	-	1	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1437261312	$1234_5678; 305419896	-	0	wz wc	$BA75_7678; -1166707080	0	1
$AA55_2200; -1437261312	$1234_5678; 305419896	-	1	wz wc	$A841_2000; -1472126976	0	0
$AA55_2200; -1437261312	$FFFF_FFFF; -1	-	0	wz wc	$FFFF_FFFF; -1	0	0
$AA55_2200; -1437261312	$FFFF_FFFF; -1	-	1	wz wc	$0000_0000; 0	1	0

Описание

MUXNC устанавливает каждый из битов величины регистра *Приемник*, соответствующий взведенному (1) биту в *Mask*, в состояние !C. Все биты *Приемник*, не указанные единицами в маске *Mask*, остаются без изменения. Эта инструкция удобна для установки либо сброса групп или отдельных битов в существующем значении.

При установленном воздействии WZ флаг Z устанавливается в 1, если в результате значение *Приемник* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если в результирующем значении *Приемник* содержится нечетное количество взведенных в 1 битов. Результат записывается в *Приемник*, если не указано воздействие NR.

MUXNZ

Инструкция: Установить отдельные биты величины в состояние !Z.

MUXNZ *Приемник*, ⟨#⟩ *Mask*

Результат: Биты *Приемник*, указанные в *Mask*, устанавливаются в значение !Z.

- ***Приемник*** (d-поле) – регистр, биты которого, заданные в *Mask*, устанавливаются в состояние, противоположное флагу Z.
- ***Mask*** (s-поле) регистр или 9-битная константа, величина которого содержит установленные в 1 биты для каждого бита в *Приемник*, устанавливаемого в состояние, противоположное флагу Z.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
011111 001i 1111	ddddddddd	sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0001; 1	0	-	wz wc	$0000_0001; 1	0	1

$0000_0000; 0	$0000_0001; 1	1	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	0	-	wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0003; 3	1	-	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1437261312	$1234_5678; 305419896	0	-	wz wc	$BA75_7678; -1166707080	0	1
$AA55_2200; -1437261312	$1234_5678; 305419896	1	-	wz wc	$A841_2000; -1472126976	0	0
$AA55_2200; -1437261312	$FFFF_FFFF; -1	0	-	wz wc	$FFFF_FFFF; -1	0	0
$AA55_2200; -1437261312	$FFFF_FFFF; -1	1	-	wz wc	$0000_0000; 0	1	0

Описание

MUXNZ устанавливает каждый из битов величины регистра *Приемник*, соответствующий взведенному в 1 биту в *Mask*, в состояние !Z. Все биты *Приемник*, не указанные единицами в маске *Mask*, остаются без изменения. Эта инструкция удобна для установки либо сброса групп или отдельных битов в существующем значении.

При установленном воздействии WZ флаг Z устанавливается в 1, если финальное значение *Приемник* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если в результирующем значении *Приемник* содержится нечетное количество взведенных в 1 битов. Результат записывается в *Приемник*, если не указано воздействие NR.

MUXZ

Инструкция: Установить отдельные биты величины в состояние флага Z.

MUXZ *Приемник*, ⟨#⟩ *Mask*

Результат: Биты *Приемник*, указанные в *Mask*, устанавливаются в значение Z.

- ***Приемник*** (d-поле) – регистр, биты которого, заданные в *Mask*, устанавливаются в состояние флага Z.
- ***Mask*** (s-поле) регистр или 9-битная константа, величина которого содержит установленные в 1 биты для каждого бита в *Приемник*, устанавливаемого в состояние флага Z.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
011110 001i 1111	ddddddddd	sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0001; 1	0	-	wz wc	$0000_0000; 0	1	0

$0000_0000; 0	$0000_0001; 1	1	-	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0003; 3	0	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	1	-	wz wc	$0000_0003; 3	0	0
$AA55_2200; -1437261312	$1234_5678; 305419896	0	-	wz wc	$A841_2000; -1472126976	0	0
$AA55_2200; -1437261312	$1234_5678; 305419896	1	-	wz wc	$BA75_7678; -1166707080	0	1
$AA55_2200; -1437261312	$FFFF_FFFF; -1	0	-	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1437261312	$FFFF_FFFF; -1	1	-	wz wc	$FFFF_FFFF; -1	0	0

Описание

MUXZ устанавливает каждый из битов величины регистра *Приемник*, соответствующий взведенному в 1 биту в *Mask*, в состояние флага Z. Все биты *Приемник*, не указанные единицами в маске *Mask*, остаются без изменения. Эта инструкция удобна для установки либо сброса групп или отдельных битов в существующем значении.

При установленном воздействии WZ флаг Z устанавливается в 1, если в результате значение *Приемник* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если в результирующем значении *Приемник* содержится нечетное количество взведенных в 1 битов. Результат записывается в *Приемник*, если не указано воздействие NR.

NEG

Инструкция: Получить отрицательное значение числа.

NEG *NValue*, ⟨#⟩ *SValue*

Результат: –*SValue* сохраняется в *NValue*.

- ***NValue*** (d-поле) – регистр, в котором сохраняется отрицательное от *SValue*.
- ***SValue*** (s-поле) – регистр или 9-битная константа, отрицательное значение которого будет сохранено в *NValue*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
101001 001i 1111 ddddddddd sssssssss	Результат = 0	S[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2147483647	-	-	wz wc	$8000_0001; -2147483647	0	0

$----_----; -	$8000_0000; -2147483648	-	-	wz wc	$8000_0000; -2147483648[1]	0	1
$----_----; -	$8000_0001; -2147483647	-	-	wz wc	$7FFF_FFFF; 2147483647	0	1

[1] **Наименьшее отрицательное число (-2147483648) не имеет соответствующего положительного в 32-битном допкоде.**

Описание

NEG записывает отрицательное значение величины *SValue* в регистр *NValue*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *SValue* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если *SValue* отрицательное, и сбрасывается в 0, если *SValue* – положительное. Результат записывается в *NValue*, если не указано воздействие **NR**.

NEGC

Инструкция: Получить величину, либо обратную ей, в зависимости от флага C.

NEGC *RValue*, ⟨#⟩ *Value*

Результат: *Value* либо *–Value* сохраняется в *RValue*.

- ***RValue*** (d-поле) – регистр, в котором сохраняется *Value* либо *–Value*.
- ***Value*** (s-поле) – регистр или 9-битная константа, величина (при C = 0) или обратная величина (при C = 1) которого записывается в *RValue*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
101100 001i 1111 ddddddddd sssssssss	Результат = 0	S[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$FFFF_FFFF; -1	-	0	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$FFFF_FFFF; -1	-	1	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	-	x	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	0	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0001; 1	-	1	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2147483647	-	0	wz wc	$7FFF_FFFF; 2147483647	0	0
$----_----; -	$7FFF_FFFF; 2147483647	-	1	wz wc	$8000_0001; -2147483647	0	0
$----_----; -	$8000_0000; -2147483648	-	x	wz wc	$8000_0000; -2147483648[1]	0	1
$----_----; -	$8000_0001; -2147483647	-	0	wz wc	$8000_0001; -2147483647	0	1
$----_----; -	$8000_0001; -2147483647	-	1	wz wc	$7FFF_FFFF; 2147483647	0	1

[1] **Наименьшее отрицательное число (-2147483648) не имеет соответствующего положительного в 32-битном допкоде.**

Описание

NEGC сохраняет *Value* (при C = 0) или *–Value* (при C = 1) в *RValue*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *Value* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если *Value* отрицательное, и сбрасывается в 0, если *Value* – положительное. Результат записывается в *RValue*, если не указано воздействие **NR**.

NEGNC

Инструкция: Получить величину, либо обратную ей, в зависимости от !C.

NEGNC *RValue*, ⟨#⟩ *Value*

Результат: *–Value* либо *Value* сохраняется в *RValue*.

- ***RValue*** (d-поле) – регистр, в котором сохраняется *–Value* либо *Value*.
- ***Value*** (s-поле) – регистр или 9-битная константа, обратная величина (при C = 0) или сама величина (при C = 1) которого записывается в *RValue*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
101101 001i	1111	ddddddddd sssssssss	Результат = 0	S[31]	Written	4

					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$FFFF_FFFF; -1	-	0	wz wc	$0000_0001; 1	0	1
$----_----; -	$FFFF_FFFF; -1	-	1	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	-	x	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	0	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	0
$----_----; -	$7FFF_FFFF; 2147483647	-	0	wz wc	$8000_0001; -2147483647	0	0
$----_----; -	$7FFF_FFFF; 2147483647	-	1	wz wc	$7FFF_FFFF; 2147483647	0	0
$----_----; -	$8000_0000; -2147483648	-	x	wz wc	$8000_0000; -2147483648[1]	0	1
$----_----; -	$8000_0001; -2147483647	-	0	wz wc	$7FFF_FFFF; 2147483647	0	1
$----_----; -	$8000_0001; -2147483647	-	1	wz wc	$8000_0001; -2147483647	0	1

[1] Наименьшее отрицательное число (-2147483648) не имеет соответствующего положительного в 32-битном допкоде.

Описание

NEGNC сохраняет *–Value* (при C = 0) либо *Value* (при C = 1) в регистре *RValue*.

При установленном воздействии WZ флаг Z устанавливается в 1, если *Value* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если *Value* отрицательное, и сбрасывается в 0, если *Value* – положительное. Результат записывается в *RValue*, если не указано воздействие NR.

NEGNZ

Инструкция: Получить величину, либо обратную ей, в зависимости от !Z.

NEGNZ *RValue*, ⟨#⟩ *Value*

Результат: –*Value* либо *Value* сохраняется в *RValue*.

- ***RValue*** (d-поле) – регистр, в котором сохраняется –*Value* либо *Value*.
- ***Value*** (s-поле) – регистр или 9-битная константа, обратная величина (при Z = 0) или сама величина (при Z = 1) которого записывается в *RValue*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
101111 001i 1111 ddddddddd sssssssss	Результат = 0	S[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$FFFF_FFFF; -1	0	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$FFFF_FFFF; -1	1	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	x	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	0	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0001; 1	1	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$7FFF_FFFF; 2147483647	0	-	wz wc	$8000_0001; -2147483647	0	0
$----_----; -	$7FFF_FFFF; 2147483647	1	-	wz wc	$7FFF_FFFF; 2147483647	0	0
$----_----; -	$8000_0000; -2147483648	x	-	wz wc	$8000_0000; -2147483648[1]	0	1
$----_----; -	$8000_0001; -2147483647	0	-	wz wc	$7FFF_FFFF; 2147483647	0	1
$----_----; -	$8000_0001; -2147483647	1	-	wz wc	$8000_0001; -2147483647	0	1

[1] **Наименьшее отрицательное число (-2147483648) не имеет соответствующего положительного в 32-битном допкоде.**

Описание

NEGNZ сохраняет –*Value* (при Z = 0) либо *Value* (при Z = 1) в *RValue*.

При установленном воздействии WZ флаг Z устанавливается в 1, если *Value* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если *Value* отрицательное, и сбрасывается в 0, если *Value* – положительное. Результат записывается в *RValue*, если не указано воздействие NR.

NEGZ

Инструкция: Получить величину, либо обратную ей, в зависимости от флага Z.

NEGZ *RValue*, ⟨#⟩ *Value*

Результат: *Value* либо *–Value* сохраняется в *RValue*.

- ***RValue*** (d-поле) – регистр, в котором сохраняется *Value* либо *–Value*.
- ***Value*** (s-поле) – регистр или 9-битная константа, величина (при Z = 0) или обратная величина (при Z = 1) которого записывается в *RValue*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
101110 001i 1111	ddddddddd	sssssssss	Результат = 0	S[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$----_----; -	$FFFF_FFFF; -1	0	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$FFFF_FFFF; -1	1	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	x	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	0	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0001; 1	1	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2147483647	0	-	wz wc	$7FFF_FFFF; 2147483647	0	0
$----_----; -	$7FFF_FFFF; 2147483647	1	-	wz wc	$8000_0001; -2147483647	0	0
$----_----; -	$8000_0000; -2147483648	x	-	wz wc	$8000_0000; -2147483648[1]	0	1
$----_----; -	$8000_0001; -2147483647	0	-	wz wc	$8000_0001; -2147483647	0	1
$----_----; -	$8000_0001; -2147483647	1	-	wz wc	$7FFF_FFFF; 2147483647	0	1

[1] **Наименьшее отрицательное число (-2147483648) не имеет соответствующего положительного в 32-битном допкоде.**

Описание

NEGZ сохраняет *Value* (при Z = 0) либо *–Value* (при Z = 1) в *RValue*.

При установленном воздействии **WZ**, флаг Z устанавливается в 1, если *Value* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если *Value* отрицательное, и сбрасывается в 0, если *Value* – положительное. Результат записывается в *RValue*, если не указано воздействие **NR**.

NOP

Инструкция: Нет операции, простой в течение четырех тактов.

NOP

–ИНСТР– ZCRI	–УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
------ ----	0000	---------	---------	---	---	---	4

<ТАБЛИЦА ИСТИННОСТИ НЕ ПРИВОДИТСЯ, ПОСКОЛЬКУ ИНСТРУКЦИЯ NOP НЕ ВЫПОЛНЯЕТ НИКАКИХ ДЕЙСТВИЙ >

Описание

NOP не выполняет операций, однако требует 4 тактов. Инструкция NOP в поле **«–УСЛ–»** содержит все ноли, то есть условие NEVER; таким образом, каждая инструкция, содержащая условие NEVER является инструкцией NOP.

Из-за этого инструкция NOP никогда не предваряется условием *Condition*, таким как IF_Z или IF_C_AND_Z, поскольку она никогда не может быть условно выполнена.

NR

Воздействие: Запрещает ассемблерной инструкции записывать результат.

⟨***Метка***⟩ ⟨***Условие***⟩ ***Инструкция Операнды*** NR

Результат: Регистр приемника результата остается неизмененным.

- ***Метка*** – опциональная метка. См. «Общие элементы синтаксиса» на стр. 408.
- ***Условие*** – опциональное условие. См. «Общие элементы синтаксиса» на стр. 408.
- ***Инструкция*** – необходимая ассемблерная инструкция.
- ***Операнды*** – это ноль, один либо два операнда, в зависимости от инструкции.

Описание

NR (Нет результата) является одним из четырех опциональных воздействий (WC, WZ, WR, и NR), которые влияют на поведение ассемблерных инструкций. NR заставляет выполняемую ассемблерную инструкцию оставить величину в регистре приемника без изменений.

Например, инструкция SHL (Сдвиг влево) сдвигает величину соответствующего регистра влево на заданное количество бит, записывает результат обратно в этот регистр, и, опционально, отображает статус флагами C и Z. Но если все, что Вам на самом деле

нужно – это значение бита C после выполнения инструкции, просто укажите воздействия **WC** и **NR**:

```
shl     value, #1     WC, NR     'Put value's MSB in C
```

В приведенном примере во флаг C записывается состояние старшего бита (бита 31) величины `value` без изменения самого значения `value`.

Для дополнительной информации см. «Воздействия» на стр. 450.

Операторы

Код на языке *Propeller*-ассемблер может содержать выражения из констант, в которых могут использоваться любые операторы, допустимые для применения в выражениях-константах. В Табл. 5-4 сведены все операторы, допустимые для использования в коде на языке *Propeller*-ассемблер (для выражений-констант). См. секцию Справочник по Операторам языка *Spin* для детального описания их функций; номера страниц для каждого оператора приведены в Табл. 5-4.

Табл. 5-4: Математич./логич. операторы в выражениях с константами		
Оператор	**Унарный**	**Описание, номер страницы**
+		Сложение, 298
+	✓	Положительное (+X); унарное от Сложение, 298
-		Вычитание, 299
-	✓	Отрицание (-X); унарное от Вычитание, 299
*		Умножить и вернуть младшие 32 бита (знаковое), 301
**		Умножить и вернуть старшие 32 бита (знаковое), 302
/		Деление (знаковое), 302
//		Остаток от деления Mod (знаковое), 303
#>		Ограничение по минимуму (знаковое), 303
<#		Ограничение по максимуму (знаковое), 304
^^	✓	Квадратный корень, 304
||	✓	Абсолютное значение, 305
~>		Арифметический сдвиг вправо, 307
|<	✓	Побитовое: Дешифровать, 309
>|	✓	Побитовое : Шифровать, 310
<<		Побитовое : Сдвиг влево, 310
>>		Побитовое : Сдвиг вправо, 311
<-		Побитовое : Циклический сдвиг влево, 312
->		Побитовое : Циклический сдвиг вправо, 312
><		Побитовое : Реверс, 313
&		Побитовое : И (AND), 314
|		Побитовое : ИЛИ (OR), 315
^		Побитовое : ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR), 316
!	✓	Побитовое : НЕ (NOT), 317
AND		Логическое: И (AND) (не-0 представляет как -1), 317
OR		Логическое: ИЛИ (OR) (не-0 представляет как -1), 318
NOT	✓	Логическое: НЕ (NOT) (не-0 представляет как -1), 319
==		Логическое: Равенство, 320
<>		Логическое: Не равно, 321
<		Логическое: Меньше (знаковое), 321
>		Логическое: Больше (знаковое), 322
=<		Логическое: Меньше или равно (знаковое), 322
=>		Логическое: Больше или равно (знаковое), 323
@	✓	Адрес идентификатора, 324

OR

Инструкция: Побитовое ИЛИ двух величин.

OR *Value1*, ⟨#⟩ *Value2*

Результат: *Value1* ИЛИ *Value2* сохраняется в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину для побитного ИЛИ с величиной *Value2* и являющийся приемником для записи результата.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого побитно складывается по ИЛИ с величиной *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
011010 001i 1111 ddddddddd sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1
$0000_000A; 10	$0000_0005; 5	–	–	wz wc	$0000_000F; 15	0	0

Описание

OR (Побитовое ИЛИ) выполняет Побитовое ИЛИ величины *Value2* с величиной *Value1*.

При установленном воздействии WZ флаг Z устанавливается в 1, если *Value1* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если *Value1* отрицательное, и сбрасывается (0), если *Value1* – положительное. Результат записывается в *Value1*, если не указано воздействие NR.

ORG

Директива: Установить указатель адреса ассемблера при компиляции.

ORG ⟨*Адрес*⟩

- ***Адрес*** – опционально адрес в ОЗУ *Cog* (0-495) начиная с которого располагать последующий код. Если *Адрес* не указан, используется значение 0.

Описание

Директива **ORG** (origin) устанавливает новое значение указателя ассемблера программы *Propeller Tool*, для его использования при ссылках в рамках ассемблерного блока. Эта директива обычно используется в виде `ORG 0`, или просто `ORG`, в начале любого нового ассемблерного блока, предназначенного для запуска в отдельном процессоре.

ORG влияет только на ссылки между идентификаторами, она не влияет на положение самого кода в процессоре. При запуске ассемблерного кода на выполнение в новом процессоре командой **COGNEW** или **COGINIT**, он всегда загружается в ОЗУ с адреса 0.

Даже учитывая то, что ассемблерная программа всегда загружается подобным образом, компилятор/ассемблер все равно не знает, где её начало, поскольку разработчик определяет сам, какую часть кода и с какого адреса запускать.

Для разрешения этой неоднозначности, ассемблер использует опорную точку (величину указалеля ассемблера) для вычисления абсолютных значений адресов идентификаторов. Такие абсолютные адреса записываются в инструкции в поля адреса приемника (d-поле) либо источника (s-поле) в месте ссылки на идентификатор.

```
DAT
                org     0               'Start at Cog RAM 0
Toggle          mov     dira, Pin       'Set I/O direction to output
:Loop           xor     outa, Pin       'Toggle output pin state
                jmp     #:Loop          'Loop endlessly

Pin             long    $0000_0010      'Use I/O pin 4 ($10 or %1_0000)
```

В этом примере директива **ORG** устанавливает указатель ассемблера в ноль, поэтому последующий код подключается с учетом этой опорной точки. Исходя из этого, идентиффикатор `Toggle`, с точки зрения ассемблера, логически расположен по адресу 0 в ОЗУ Cog, идентификатор `:Loop` – по адресу 1, а `Pin` – по адресу 3. Ассемблер заменяет все обращения к этим идентификаторам назначенными им адресами.

При запуске кода `Toggle`, – например при помощи команды `COGNEW(@Toggle, 0)`, – он будет корректно выполняться, начиная с адреса 0 в ОЗУ Cog, поскольку адреса всех идентификаторов были расчитаны исходя из этой стартовой точки. Но если бы директива **ORG** была представлена в виде `ORG 1`, и был запущен код `Toggle`, он бы не выполнился корректно, поскольку адреса идентификаторов были бы расчитаны от ошибочной опорной точки (1 вместо 0).

Директива **ORG** может встречаться в Propeller-объекте несколько раз, располагаясь каждый раз непосредственно перед участком выполнимого ассемблерного кода.

Однако использование ORG со значениями, отличными от ноля не является частой практикой, хотя и может оказаться полезным при создании перемещаемых во время исполнения участков ассемблерного кода.

OUTA, OUTB

Регистр: Выходной регистр 32-битных портов A и B.

```
DAT
   〈Метка〉 〈Условие〉 Инструкция   OUTA, ОперандИст, 〈Воздействия〉
DAT
   〈Метка〉 〈Условие〉 Инструкция   ОперандПрм   OUTA, 〈Воздействия〉

DAT
   〈Метка〉 〈Условие〉 Инструкция   OUTB, ОперандИст, 〈Воздействия〉(зарезервирован)
DAT
   〈Метка〉 〈Условие〉 Инструкция   ОперандПрм   OUTB, 〈Воздействия〉 (зарезервирован)
```

Результат: Обновление выходного регистра (опционально).

- ***Метка*** – метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистры OUTA и OUTB могут использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандИст*** – выражение, используемое инструкцией для выполнения действия, и при необходимости – записи в регисры OUTA или OUTB в ***ОперандПрм***.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра OUTA либо OUTB из поля операнда-источника.

Описание

OUTA и OUTB – это два из шести регистров специальных функций (DIRA, DIRB, INA, INB, OUTA и OUTB), которые непосредственно воздействуют на линии B/B. Биты регистров OUTA и OUTB отображают состояния выходов каждой из 32 линий B/B портов A и B соответственно. Идентификатор OUTB зарезервирован для использования в

будущем, ИМС Propeller P8X32A не содержит линий порта B, поэтому дальнейшее обсуждение будет ограничено регистром OUTA.

OUTA является регистром для чтения/записи, который можно указывать в качестве операнда-приемника либо -источника в соответствующих полях инструкции. Если линия B/B установлена на вывод, то ноль в соответствующем бите OUTA дает на выходе уровень земли, а единица – уровень напряжения питания VDD (3.3 B). B следующем примере линии B/B с P0 по P3 устанавливаются для вывода всех единиц.

```
mov     dira, #$0F
mov     outa, #$0F
```

Для дальнейшей инфоромации см. Регистры, стр. 499, и Язык Spin, секцию OUTA, OUTB, стр. 326. Имейте в виду, что в языке *Propeller*-ассемблер, в отличие от языка Spin, доступ ко всем 32 битам регистра OUTA осуществляется одновременно, если только не используются инструкции MUXx.

PAR

Регистр: Регистр параметров загрузки процессора.

DAT
〈***Метка***〉 〈***Условие***〉 ***Инструкция*** ***ОперандПрм*** PAR, 〈***Воздействия***〉

- ***Метка*** – метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистр PAR является регистром только для чтения, поэтому может использоваться только в поле операнда-источника.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра PAR.

Описание

Регистр PAR содержит значение адреса, передаваемое в поле *Параметр* команды COGINIT или COGNEW языка *Spin*, либо в старшие биты поля *Назначение* ассемблерной команды COGINIT. При старте процессора, его ассемблерная программа может

использовать содержимое регистра PAR для выделения и работы с общей памятью, разделяемой между Cog-ом и вызвавшим его кодом.

Важно помнить, что величина, передаваемая через PAR, должна быть адресом *long*-а, поэтому умещаются только 14-битные значения (биты со 2 по 15); нижние два бита очищаются в ноль, чтобы обеспечить выравнивание по сетке адресов *long*-ов. В то же время могут передаваться и величины, не являющиеся адресами *long*-ов, но они должны быть не более 14 бит размером, и сдвинуты влево вызывающим кодом и затем, соответственно назад, вправо – программой в стартовавшем процессоре.

PAR является псевдо-регистром только для чтения; когда он используеся в поле операнда-источника инструкции, происходит чтение величины, переданной в процессор при его запуске. Не следует использовать PAR как операнд-приемник, это приведет лишь к чтению и модификации теневого регистра с адресом, занимаемым регистром PAR.

В приведенном далее коде значение регистра PAR сохраняется в регистре Addr для дальнейшего использования. Для дополнительной информации см. Регистры, стр. 499, и Язык Spin, секцию PAR, стр. 330.

```
DAT
                org 0                   'Reset assembly
pointer
AsmCode         mov     Addr, par       'Get shared address
                '<more code here>       'Perform operation

Addr    res     1
```

PHSA, PHSB

Регистр: Регистры фазы для счетчиков A и B.

```
DAT
  ⟨Метка⟩ ⟨Условие⟩ Инструкция  PHSA, ОперандИст, ⟨Воздействия⟩
DAT
  ⟨Метка⟩ ⟨Условие⟩ Инструкция  ОперандПрм  PHSA, ⟨Воздействия⟩

DAT
  ⟨Метка⟩ ⟨Условие⟩ Инструкция  PHSB, ОперандИст, ⟨Воздействия⟩
DAT
  ⟨Метка⟩ ⟨Условие⟩ Инструкция  ОперандПрм  PHSB, ⟨Воздействия⟩
```

Результат: Обновление регистров фаз (опционально).

- ***Метка*** – метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистры PHSA и PHSB могут использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандИст*** – выражение, используемое инструкцией для выполнения действия, и при необходимости – записи в регисры PHSA или PHSB в ***ОперандПрм***.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра PHSA либо PHSB из поля операнда-источника.

Описание

PHSA и PHSB – это два из шести регистров (CTRA, CTRB, FRQA, FRQB, PHSA и PHSB), которые непосредственно влияют на работу модулей счетчиков процессора. В каждом *cog*-е имеется два одинаковых модулей счетчиков (A и B), которые могут выполнять многие повторяющиеся задачи. Регистры PHSA и PHSB содержат накопленные значения величин регистров FRQA и FRQB соответственно, согласно заданному режиму работы и входных воздействий каждого из счетчиков. Для дополнительной информации см. Язык Spin, секцию CTRA, CTRB на стр. 239.

PHSA и PHSB – это псевдо-регистры для чтения/записи. При использовании в поле операнда-источника, они позволяют прочесть значение аккумулятора соответствующего счетчика. В поле операнда-приемника они читают значение теневого регистра, адрес которого совпадает с занимаемыми ими, но изменения затрагивают как теневой регистр, так и соответствующий регистр аккумулятора.

В следующем далее примере производится запись величины из регистра PHSA в регистр Result. Для дополнительной информации см. Регистры, стр. 499, и Язык Spin, секцию PHSA, PHSB на стр. 332.

```
mov     Result, phsa     'Get current phase value
```

RCL

Инструкция: Сдвинуть величину с флагом C влево на заданное количество бит.

RCL *Value*, ⟨#⟩ *Bits*

Результат: Величина *Value* имеет *Bits* копий флага C, сдвинутого в ней влево.

- ***Value*** (d-поле) – регистр, который сдвигается влево с флагом C.
- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которого представляет число битов, на которое сдвигается влево величина *Value* с флагом C.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
001101 001i 1111	ddddddddd	sssssssss	Результат = 0	D[31]	Записан	4

Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$8000_0000; -2147483648	$0000_0000; 0	-	x	wz wc	$8000_0000; -2147483648	0	1
$8000_0000; -2147483648	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	1
$8000_0000; -2147483648	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	1
$2108_4048; 554188872	$0000_0002; 2	-	0	wz wc	$8421_0120; -2078211808	0	0
$2108_4048; 554188872	$0000_0002; 2	-	1	wz wc	$8421_0123; -2078211805	0	0
$8765_4321; -2023406815	$0000_0004; 4	-	0	wz wc	$7654_3210; 1985229328	0	1
$8765_4321; -2023406815	$0000_0004; 4	-	1	wz wc	$7654_321F; 1985229343	0	1

Описание

RCL (Rotate Carry Left) выполняет сдвиг влево величины *Value*, на *Bits* позиций, используя исходное значение флага C для каждого изменяемого LSB.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если результирующее значение *Value* равно нолю. При указанном воздействии **WC**, в конце операции флаг C устанавливается равным значению исходного 31-го бита величины *Value*. Результат записывается в *Value1*, если не указано воздействие **NR**.

RCR

Инструкция: Сдвинуть величину с флагом C вправо на заданное количество бит.

RCR Value, ⟨#⟩ Bits

Результат: Величина *Value* имеет *Bits* копий флага C, сдвинутого в ней вправо.

- ***Value*** (d-поле) – регистр, который сдвигается вправо с флагом C.
- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которого представляет число битов, на которое сдвигается вправо величина *Value* с флагом C.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
001100 001i 1111	ddddddddd	sssssssss	Результат = 0	D[0]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0000; 0	–	x	wz wc	$0000_0001; 1	0	1
$0000_0001; 1	$0000_0001; 1	–	0	wz wc	$0000_0000; 0	1	1
$0000_0001; 1	$0000_0001; 1	–	1	wz wc	$8000_0000; -2147483648	0	1
$18C2_1084; 415371396	$0000_0002; 2	–	0	wz wc	$0630_8421; 103842849	0	0
$18C2_1084; 415371396	$0000_0002; 2	–	1	wz wc	$C630_8421; -969898975	0	0
$8765_4321; -2023406815	$0000_0004; 4	–	0	wz wc	$0876_5432; 141972530	0	1
$8765_4321; -2023406815	$0000_0004; 4	–	1	wz wc	$F876_5432; -126462926	0	1

Описание

RCR (Rotate Carry Right) выполняет сдвиг вправо величины *Value*, на *Bits* позиций, используя исходное значение флага C для каждого изменяемого MSB.

При установленном воздействии **WZ,** флаг Z в 1, если результирующее значение *Value* равно нолю. При указанном воздействии WC, в конце операции флаг C устанавливается равным значению исходного 0-го бита величины *Value*. Результат записывается в *Value*, если не указано воздействие NR.

RDBYTE

Инструкция: Прочитать байт из Основной Памяти.

RDBYTE *Value*, ⟨#⟩ *Address*

Результат: Прочитанный и дополненный нолями байт сохраняется в *Value*.

- ***Value*** (d-поле) – регистр, в котором сохраняется дополненный нолями байт.
- ***Address*** (s-поле) – регистр или 9-битная константа, значение которой представляет адрес в Основной Памяти, по которому будет производиться чтение.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000000 001i 1111	ddddddddd	sssssssss	Результат = 0	---	Записан	7..22

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z[2]	C
$----_----; -	$----_----; -	–	–	wz wc	31:8 = 0, 7:0 = значение байта	0	0

[1] **Величина в регистре Приемника на выходе представляет собой байт, прочитанный из основной памяти, дополненный нолями, которая генерируется в любом случае, поскольку указание эффекта NR преобразовало бы инструкцию RDBYTE в инструкцию WRBYTE.**

[2] **Флаг Z сбрасывается в 0, кроме случая, когда значение регистра Приемника на выходе равно 0.**

Описание

RDBYTE синхронизируется с *Hub*, читает байт Основной Памяти по адресу *Address*, дополняет его слева нолями и сохраняет в регистре *Value*.

При установленном воздействии WZ флаг Z устанавливается в 1, если прочитанное из Основной Памяти значение равно нолю. Воздействие NR не может использоваться совместно с RDBYTE, поскольку это изменит ее на инструкцию WRBYTE.

RDBYTE – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

RDLONG

Инструкция: Прочитать двойное слово (*long*) из Основной Памяти.

RDLONG *Value*, ⟨#⟩ *Address*

Результат: *Long* сохраняется в *Value*.

- ***Value*** (d-поле) – регистр, в котором сохраняется прочитанный *long*.
- ***Address*** (s-поле) – регистр или 9-битная константа, значение которой представляет адрес в Основной Памяти, по которому будет производиться чтение.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000010 001i 1111	ddddddddd	sssssssss	Результат = 0	---	Записан	7..22

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	Long-величина	0	0

[1] Значение в регистре Приемника на выходе генерируется всегда, поскольку указание воздействия NR преобразовало бы инструкцию RDLONG в инструкцию WRLONG.

[2] Флаг Z сбрасывается в 0, кроме случая, когда значение регистра Приемника на выходе равно 0.

Описание

RDLONG синхронизируется с *Hub*, читает *long* Основной Памяти по адресу *Address* и сохраняет его в регистре *Value*. Параметр *Address* может указывать на любой байт в рамках необходимого *long*-а; младшие два бита адреса будут очищены в ноль, выполняя выравнивание адреса на границу *long*-а.

При установленном воздействии WZ, флаг Z устанавливается в 1, если прочитанное из Основной Памяти значение равно нолю. Воздействие NR не может использоваться совместно с RDLONG, поскольку это изменит ее на инструкцию WRLONG.

RDLONG – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

RDWORD

Инструкция: Прочитать слово из Основной Памяти.

RDWORD *Value*, ⟨#⟩ *Address*

Результат: Прочитанное дополненное нолями слово сохраняется в *Value*.

- ***Value*** (d-поле) – регистр, в котором сохраняется дополненное нолями слово.
- ***Address*** (s-поле) – регистр или 9-битная константа, значение которой представляет адрес в Основной Памяти, по которому будет производиться чтение.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000001 001i 1111	ddddddddd	sssssssss	Результат = 0	---	Записан	7..22

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	31:16 = 0, 15:0 = word-величина	0	0

[1] **Значение в регистре Приемника на выходе представляет собой прочитанное из основной памяти и дополненное нолями слово, и оно генерируется всегда, поскольку указание воздействия NR преобразовало бы инструкцию RDWORD в WRWORD.**

[2] **Флаг Z сбрасывается в 0, кроме случая, когда значение регистра Приемника на выходе равно 0.**

Описание

RDWORD синхронизируется с *Hub*, читает слово из Основной Памяти по адресу *Address*, дополняет его слева нолями и сохраняет в регистре *Value*. Параметр *Address* может указывать на любой байт в рамках необходимого *word*-а; младшие два бита адреса будут очищены в ноль, выполняя выравнивание адреса на границу *word*-а.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если прочитанное из Основной Памяти значение равно нолю. Воздействие NR не может использоваться совместно с RDWORD, поскольку это изменит ее на инструкцию WRWORD .

RDWORD – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

Регистры

Каждый из процессоров содержит 16 регистров специальных функций (РСФ), которые служат для доступа к линиям В/В, встроенным счетчикам и видео генератору, а так же к передаваемому при запуске *Cog*-а параметру. Все эти регистры описаны в Справочнике по Языку *Spin*, и бОльшая часть информации применима как для языка *Propeller Spin*, так и для *Propeller*-Ассемблера. В следующей таблице сведены все 16 РСФ; в ней указано, где можно найти подробную информацию, а также какие моменты, если таковые имеются, не относятся к языку *Propeller*-Ассемблер.

Доступ к каждому из этих регистров возможен, как и для остальных регистров, с использованием полей *Приемника* или *Источника* инструкции, за исключением регистров, обозначенных примечанием 1 либо 2. Такие специальные регистры могут лишь быть прочитаны посредством поля *Источник*, они не перезаписываемы (1); либо они не могут использоваться в поле *Приемника* для операций чтение-модификация-запись (2).

Табл. 5-5: Регистры	
Регистр(ы)	**Описание**
DIRA, DIRB [3]	Регистры Направления для 32-битных портов A и B. См. секцию Описание DIRA, DIRB на стр. 249. Опциональный параметр "[*Pin(s)*]" не применяется в ассемблере Propeller; все биты всего регистра читаются/записываются одновременно, если не используются инструкции MUXx.
INA [1], INB [1,3]	Входные Регистры 32-битных портов A и B (только чтение). См. секцию Описание INA, INB on стр. 264. Опциональный параметр "[*Pin(s)*]" не применяется в ассемблере Propeller; все биты регистра читаются/записываются за один раз.
OUTA, OUTB [3]	Выходные Регистры 32-битных портов A и B. См. секцию Описание OUTA, OUTB на стр. 326. Опциональный параметр "[*Pin(s)*]" не применяется в ассемблере Propeller; все биты всего регистра читаются/записываются за один раз, если не используются инструкции MUXx.
CNT [1]	32-битный регистр Системного Счетчика. (Только чтение). См. секцию Описание CNT на стр. 215.
CTRA, CTRB	Регистры управления Счетчиков A и B. См. CTRA, CTRB на стр. 239 и стр. 447.
FRQA, FRQB	Регистры Частоты Счетчиков A и B. См. FRQA, FRQB на стр. 256 и стр. 451.
PHSA [2], PHSB [2]	Регистры ФАПЧ Счетчиков A и B. См. PHSA, PHSB на стр. 332 и стр. 492.
VCFG	Регистр Конфигурации Видео. См. VCFG на стр. 368 и стр. 526.
VSCL	Регистр Масштаба Видео. См. VSCL на стр. 371 и стр. 527.
PAR [1]	Регистр параметра Загрузки Процессора (только для чтения). См. PAR на стр. 330.

Прим. 1: Для языка *Propeller*-ассемблер, доступ возможен только как к регистру-источнику (т.е. mov *dest*, *source*). См. Язык Ассемблера, секции: PAR, стр. 491, CNT, стр. 440, и INA, INB, стр. 457.

Прим. 2: Для языка *Propeller*-ассемблер, только чтение как регистра-источника, (т.е. mov *dest, source*); операции чтение-модификация-запись как с регистром-приемником не возможны. См. Язык Ассемблера, секцию PHSA, PHSB на стр. 492.

Прим. 3: Зарезервировано для использования в будущем.

RES

Директива: Зарезервировать следующий *long*(-и) для идентификатора.

⟨***Symbol***⟩ RES ⟨***Count***⟩

- ***Symbol*** – опциональное имя для резервируемой области в ОЗУ *Cog*.
- ***Count*** – опциональное количество *long*-ов для резервирования под *Symbol*. Если не указано, `RES` резервирует один *long*.

Описание

Директива `RES` (reserve) резервирует одно или более двойных слов в ОЗУ *Cog* путем увеличения указателя ассемблера этапа компиляции на величину *Count*. Обычно это используется для резервирования памяти под ассемблерный идентификатор, который не нужно инициализировать в определенное значение. Например:

```
DAT
                ORG     0
AsmCode         mov     Time, cnt         'Get system counter
                add     Time, Delay       'Add delay
:Loop           waitcnt Time, Delay       'Wait for time window
                nop                       'Do something useful
                jmp     #:Loop            'Loop endlessly

Delay   long    6_000_000                 'Time window size
Time    RES     1                         'Time window workspace
```

Последняя строка приведенного выше примера `AsmCode`, резервирует один *long* ОЗУ процессора под идентификатор `Time` без задания его начального значения. Идентификатор `Time` используется в коде `AsmCode` как переменная *long,* предназначенная для ожидания начала окна времени в 6 миллионов тактов. При запуске программы `AsmCode` в процессоре, она загружается в ОЗУ *Cog*-а как показано ниже.

Symbol	Address	Instruction/Data
AsmCode	0	mov Time, cnt
	1	add Time, Delay
:Loop	2	waitcnt Time, Delay

	3	nop
	4	jmp #:Loop
Delay	5	6_000_000
Time	6	?

RES попросту увеличивает указатель ассемлера этапа компиляции, что влияет на дальнейшее распределение адресов (в ОЗУ *Cog*); он не занимает место в объекте (в Основном ОЗУ). Это отличие важно, оно воздействует на код объекта, инициализируемые идентификаторы и влияет на работу в процессе выполнения.

- Поскольку инкрементируется лишь указатель ассемблера этапа компиляции, то вычисляемый адрес всех идентификаторов, следующих за выражением RES также изменяется .
- Место в объекте/приложении (Основной Памяти) не занимается. Это является преимуществом при задании буферов данных, которые должны существовать лишь в ОЗУ *Cog*, но не в Основном ОЗУ.

Внимание: Используйте RES только ПОСЛЕ всех инструкций и данных

Важно помнить, что использовать RES в логически верной ассемблерной программе можно лишь после завершающих инструкций и данных. Расположение RES в более ранних областях может привести к нежелательным эффектам, описаным ниже.

Помните, что ассемблерные инструкции и данные располагаются в образе памяти приложения в точно таком же порядке, что и в исходном коде, независимо от указателя ассемблера. Так необходимо потому, что запущеный ассемблерный код должен загружаться по порядку, начиная с метки необходимой подпрограммы. Однако, поскольку RES не генерирует никаких данных либо инструкций), она не производит абсолютно никакого воздействия на образ памяти приложения, RES лишь устанавливает величину указателя ассемблера.

По природе директивы RES, любые данные либо код, оказавшийся после выражения RES, будет помещен непосредственно после последнего не-RES экземпляра, в том же логическом пространстве, что и сами экземпляры RES. Рассмотрим следующий пример, в котором приведен код предыдущего примера с измененным порядком следования объявлениний Time и Delay.

```
DAT
                ORG     0
AsmCode         mov     Time, cnt       'Get system counter
                add     Time, Delay     'Add delay
:Loop           waitcnt Time, Delay     'Wait for time window
```

```
                nop                             'Do something useful
                jmp     #:Loop                  'Loop endlessly

Time    RES     1                               'Time window workspace
Delay   long    6_000_000                       'Time window size
```

Этот пример будет загружен в процессор следующим образом:

Symbol	Address	Instruction/Data
AsmCode	0	mov Time, cnt
	1	add Time, Delay
:Loop	2	waitcnt Time, Delay
	3	nop
	4	jmp #:Loop
Time	5	6_000_000
Delay	6	?

Заметили, что порядок следования Time и Delay изменен по отношению к предыдущему примеру, в то время, как порядок данных – нет? Вот что произошло:

- Сначала ассемблер расположил все в образе памяти объекта точно так же, как и в прошлом примере, дойдя до и включая инструкцию **JMP**.
- Ассемблер достиг идентификатора Time, который объявлен с директивой **RES**, поэтому он приравнял идентификатор адресу 5 (текущее значение указателя ассемблера), после чего увеличил указатель на 1. На этом шаге в образе памяти приложения не было размещено никаких данных.
- Ассемблер достиг идентификатора Delay, который оъявлен как данные размером **LONG**, поэтому он приравнял Delay адресу 6 (текущее значение указателя ассемблера), увеличил указатель на 1, после чего разместил данные, 6000000, в следующую доступную ячейку образа памяти сразу после инструкции **JMP**, с которой ассоциирован идентификатор Time.

В результате, при запуске в процессоре этого кода, идентификатор Time занимает ту же самую область памяти ОЗУ *Cog*, что и первоначальное значение Delay, а Delay находится в соседней ячейке, которая содержит непроинициализированные данные. Такой код не будет работать так, как задумывалось.

По этой причине, как видно из приведенного примера, лучше всего располагать операторы RES после самой последней инструкции и объявленных данных, от которых зависит выполнение программы.

RET

Инструкция: Возврат на ранее записанный адрес.

RET

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
010111 0001 1111	---------	---------	Результат = 0	---	Не записан	4

Вход					Выход		
Приемник[1]	Источник	Z	C	Возд.	Приемник[2]	Z	C[3]
s----_----; -	s----_----; -	-	-	wr wz wc	31:9 неизм., 8:0 = PC+1	0	1

[1] При типичном использовании RET, Приемник обычно игнорируется, однако если указано воздействие WR, инструкция RET становится инструкцией CALL, и s-поле Приемника (младшие 9 бит)перезаписывается адресом возврата (PC+1).

[2] Приемник не перезаписывается, если не задано воздействие WR.

[3] Флаг C устанавливается в 1, кроме случая когда PC+1 равен 0, что очень маловероятно, поскольку при этом RET будет выполняться с самого начала cog RAM ($1F, регистр специальных функций VSCL).

Описание

RET осуществляет возврат выполнения на предварительно записанный адрес путемустановки программного счетчика (PC) в значение этого адреса. Инструкция RET должна использоваться вместе с меткой в формате “label_ret” и инструкцией CALL, указывающей на заданную подпрограмму с RET – “label”. См. CALL на стр. 427, для более детальной информации.

RET является подмножеством инструкции JMP, но с установленным i- полем и незаданным s-полем. Она также тесно связана с командами CALL и JMPRET; на самом деле, все они имеют одинаковый опкод, но с различными значениями в r- и i-полях, а также изменяющимися под управлением ассемблера и разработчика значениями d- и s-полей.

REV

Инструкция: Реверсировать и дополнить нолями LSB-биты величины.

REV *Value*, ⟨#⟩ *Bits*

Результат: У *Value* младшие (*32- Bits)* его LSB реверсируются, а старшие очищаются.

- ***Value*** (d-поле) – регистр, содержащий величину, биты которой реверсируются.

- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которой вычитается из 32, (32 - *Bits*) – количество LSB битов величины *Value* для реверсирования. Старшие *Bits* MSB величины *Value* очищаются в 0.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
001111 001i 1111 ddddddddd sssssssss	Результат = 0	D[0]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$8421_DECA; -2078155062	$0000_001F; 31	–	–	wz wc	$0000_0000; 0	1	0
$8421_DECA; -2078155062	$0000_001C; 28	–	–	wz wc	$0000_0005; 5	0	0
$8421_DECA; -2078155062	$0000_0018; 24	–	–	wz wc	$0000_0053; 83	0	0
$8421_DECA; -2078155062	$0000_0010; 16	–	–	wz wc	$0000_537B; 21371	0	0
$8421_DECA; -2078155062	$0000_0000; 0	–	–	wz wc	$537B_8421; 1400603681	0	0
$4321_8765; 1126270821	$0000_001C; 28	–	–	wz wc	$0000_000A; 10	0	1
$4321_8765; 1126270821	$0000_0018; 24	–	–	wz wc	$0000_00A6; 166	0	1
$4321_8765; 1126270821	$0000_0010; 16	–	–	wz wc	$0000_A6E1; 42721	0	1
$4321_8765; 1126270821	$0000_0000; 0	–	–	wz wc	$A6E1_84C2; -1495169854	0	1

Описание

REV (Reverse) реверсирует младшие (32 - *Bits*) LSB-битов величины *Value* и очищает старшие *Bits* MSB-битов этой величины.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если результирующее значение *Value* равно нолю. При указанном WC, флаг C устанавливается равным биту 0 исходного значения *Value*. Результат записывается в *Value*, если не указано NR.

ROL

Инструкция: Сдвиг величины влево на заданное количество битов.

ROL *Value*, ⟨#⟩ *Bits*

Результат: *Value* сдвигается влево на количество битов *Bits*.

- ***Value*** (d-поле) – регистр для сдвига влево.
- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которого представляет количество битов для сдвига влево.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
001001 001i 1111 ddddddddd sssssssss	Результат = 0	D[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$8765_4321; -2023406815	$0000_0004; 4	–	–	wz wc	$7654_3218; 1985229336	0	1
$7654_3218; 1985229336	$0000_000C; 12	–	–	wz wc	$4321_8765; 1126270821	0	0
$4321_8765; 1126270821	$0000_0010; 16	–	–	wz wc	$8765_4321; -2023406815	0	0

Описание

ROL (Rotate Left) сдвигает величину *Value* влево *Bits* раз. Биты MSB, выдвигаемые из величины *Value*, задвигаются в ее биты LSB.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если результирующее значение *Value* равно нолю. При указанном **WC**, флаг C устанавливается равным биту 31 исходного значения *Value*. Результат записывается в *Value*, если не указано **NR**.

ROR

Инструкция: Сдвиг величины вправо на заданное количество битов.

ROR ***Value***, ⟨#⟩ ***Bits***

Результат: *Value* сдвигается вправо на количество битов *Bits*.

- ***Value*** (d-поле) – регистр для сдвига вправо.
- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которого представляет количество битов для сдвига вправо.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
001000 001i 1111 ddddddddd sssssssss	Результат = 0	D[0]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$1234_5678; 305419896	$0000_0004; 4	–	–	wz wc	$8123_4567; -2128394905	0	0
$8123_4567; -2128394905	$0000_000C; 12	–	–	wz wc	$5678_1234; 1450709556	0	1
$5678_1234; 1450709556	$0000_0010; 16	–	–	wz wc	$1234_5678; 305419896	0	0

Описание

ROR (Rotate Right) сдвигает величину *Value* вправо *Bits* раз. Биты LSB, выдвигаемые из величины *Value*, задвигаются в ее биты MSB.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если результирующее значение *Value* равно нолю. При указанном WC, флаг C устанавливается равным биту 0 исходного значения *Value*. Результат записывается в *Value*, если не указано воздействие NR.

SAR

Инструкция: Арифметический сдвиг значения вправо на заданное количество битов.

SAR *Value*, ⟨#⟩ *Bits*

Результат: *Value* сдвигается вправо на число битов *Bits*.

- ***Value*** (d-поле) – регистр для арифметического сдвига вправо.
- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которого представляет количество битов для арифметического сдвига вправо.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
001110 001i	1111 ddddddddd	sssssssss	Результат = 0	D[0]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$FFFF_FF9C; -100	$0000_0001; 1	–	–	wz wc	$FFFF_FFCE; -50	0	0
$FFFF_FF9C; -100	$0000_0002; 2	–	–	wz wc	$FFFF_FFE7; -25	0	0
$FFFF_FF9C; -100	$0000_0003; 3	–	–	wz wc	$FFFF_FFF3; -13	0	0
$FFFF_FFF3; -13	$0000_0001; 1	–	–	wz wc	$FFFF_FFF9; -7	0	1
$FFFF_FFF9; -7	$0000_0001; 1	–	–	wz wc	$FFFF_FFFC; -4	0	1
$FFFF_FFFC; -4	$0000_0001; 1	–	–	wz wc	$FFFF_FFFE; -2	0	0
$0000_0006; 6	$0000_0001; 1	–	–	wz wc	$0000_0003; 3	0	0
$0000_0006; 6	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0006; 6	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	0

Описание

SAR (Shift Arithmetic Right) сдвигает величину *Value* вправо на число битов *Bits*, дополняя по мере сдвига битом MSB. Таким образом в величине со знаком знак

сохраняется, и SAR представляет собой быстрый вариант деления на величину, равную степени двух, для целых величин со знаком.

При установленном воздействии WZ флаг Z устанавливается в 1, если результирующее *Value* равно нолю. При указанном WC, флаг C устанавливается равным биту 0 исходного значения *Value*. Результат записывается в *Value*, если не указано воздействие NR.

SHL

Инструкция: Сдвиг величины влево на заданное количество битов.

SHL *Value*, ⟨#⟩ *Bits*

Результат: *Value* сдвигается влево на количество битов *Bits*.

- ***Value*** (d-поле) – регистр для сдвига влево.
- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которого представляет количество битов для сдвига влево.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
001011 001i 1111	ddddddddd	sssssssss	Результат = 0	D[31]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$8765_4321; -2023406815	$0000_0004; 4	–	–	wz wc	$7654_3210; 1985229328	0	1
$7654_3210; 1985229328	$0000_000C; 12	–	–	wz wc	$4321_0000; 1126236160	0	0
$4321_0000; 1126236160	$0000_0010; 16	–	–	wz wc	$0000_0000; 0	1	0

Описание

SHL (Shift Left) сдвигает *Value* влево на *Bits* битов, а новые LSB устанавливает в 0.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если результирующее *Value* равно нолю. При указанном **WC**, флаг C устанавливается равным биту 31 исходного значения *Value*. Результат записывается в *Value*, если не указано **NR**.

SHR

Инструкция: Сдвиг величины вправо на заданное количество битов.

SHR *Value*, ⟨#⟩ *Bits*

Результат: *Value* сдвигается вправо на количество битов *Bits*.

- ***Value*** (d-поле) – регистр для сдвига вправо.
- ***Bits*** (s-поле) – регистр или 5-битная константа, значение которого представляет количество битов для сдвига вправо.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
001010 001i 1111 ddddddddd sssssssss	Результат = 0	D[0]	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$1234_5678; 305419896	$0000_0004; 4	–	–	wz wc	$0123_4567; 19088743	0	0
$0123_4567; 19088743	$0000_000C; 12	–	–	wz wc	$0000_1234; 4660	0	1
$0000_1234; 4660	$0000_0010; 16	–	–	wz wc	$0000_0000; 0	1	0

Описание

SHR (Shift Right) сдвигает *Value* вправо на *Bits* битов, а новые MSB устанавливает в 0.

При установленном воздействии WZ, флаг Z устанавливается в 1, если результирующее значение *Value* равно нолю. При указанном WC, флаг C устанавливается равным биту 0 исходного значения *Value*. Результат записывается в *Value*, если не указано NR.

SUB

Инструкция: Вычитание двух беззнаковых величин.

SUB *Value1*, ⟨#⟩ *Value2*

Результат: Разность беззнакового *Value1* и беззнакового *Value2* сохраняется в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину, уменьшаемую на *Value2*, и являющийся приемником для записи результата.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого вычитается из *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
100001 001i 1111 ddddddddd sssssssss	D - S = 0	Беззнаков. заем	Записан	4

Вход					Выход		
Приемник[1]	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0002; 2	–	–	wz wc	$0000_0000; 0	1	0
$0000_0002; 2	$0000_0003; 3	–	–	wz wc	$FFFF_FFFF; 4294967295	0	1

[1] **Источник и Приемник рассматриваются как величины без знака.**

Описание

SUB вычитает беззнаковую величину *Value2* из беззнаковой величины *Value1* и сохраняет результат в регистре *Value1*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *Value1* − *Value2* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если в результате вычитания возникает беззнаковый заем (32-битное переполнение). Результат записывается в *Value1*, если не указано NR.

Для выполнения вычитания двух беззнаковых величин размером в несколько *long*-ов, используйте SUB, сопровождаемую командой SUBX. Для дополнительной информации, см. SUBX на стр. 516.

SUBABS

Инструкция: Вычесть абсолютное значение из другой величины.

SUBABS *Value*, ⟨#⟩ *SValue*

Результат: Разность величины *Value* и абсолютного значения величины со знаком *SValue* сохраняется в *Value*.

- ***Value*** (d-поле) – регистр, содержащий величину, уменьшаемую на абсолютное значение величины *SValue*, и являющийся приемником для записи результата.
- ***SValue*** (s-поле) – регистр или 9-битная константа, абсолютное значение которой вычитается из *Value*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
100011 001i 1111	ddddddddd	sssssssss	D - |S| = 0	Беззнаков. заем[1]	Записан	4

1: При отрицательном S, результат в C является инверсным от беззнакового переноса (для D + S).

Вход					Выход		
Приемник[1]	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0003; 3	$FFFF_FFFC; -4	–	–	wz wc	$FFFF_FFFF; 4294967295	0	0
$0000_0003; 3	$FFFF_FFFD; -3	–	–	wz wc	$0000_0000; 0	1	1
$0000_0003; 3	$FFFF_FFFE; -2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0003; 3	$FFFF_FFFF; -1	–	–	wz wc	$0000_0002; 2	0	1
$0000_0003; 3	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wz wc	$FFFF_FFFF; 4294967295	0	1

[1] **Приемник рассматривается как величина без знака.**

Описание

SUBABS вычитает абсолютное значение *SValue* из *Value* и сохраняет результат в регистре *Value*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *Value* − |*SValue*| равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если в результате вычитания возник беззнаковый заем (32-битное переполнение). Результат записывается в *Value*, если не указано **NR**.

SUBS

Инструкция: Вычитание двух знаковых величин.

SUBS *SValue1*, ⟨#⟩ *SValue2*

Результат: Разность знакового *SValue1* и знакового *SValue2* сохраняется в *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину, уменьшаемую на *SValue2*, и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) регистр или 9-битная константа, величина которого вычитается из *SValue1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
110101 001i 1111 ddddddddd sssssssss	D - S = 0	Знаков.переполнение	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFE; -2	–	–	wz wc	$0000_0001; 1	0	0
$8000_0001; -2147483647	$0000_0001; 1	–	–	wz wc	$8000_0000; -2147483648	0	0
$8000_0001; -2147483647	$0000_0002; 2	–	–	wz wc	$7FFF_FFFF; 2147483647	0	1
$7FFF_FFFE; 2147483646	$FFFF_FFFF; -1	–	–	wz wc	$7FFF_FFFF; 2147483647	0	0
$7FFF_FFFE; 2147483646	$FFFF_FFFE; -2	–	–	wz wc	$8000_0000; -2147483648	0	1

Описание

SUBS вычитает величину со знаком *SValue2* из величины со знаком *SValue1* и сохраняет результат в регистре *SValue1*.

При установленном воздействии WZ, флаг Z устанавливается в 1, если *SValue1* − *SValue2* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если при вычитании возникло знаковое переполнение. Результат записывается в *Svalue1*, если не указано воздействие NR.

Для выполнения вычитания *multi-long* чисел со знаком, используйте команды SUB, возможно, SUBX, и, в конце, − SUBSX. Для информации см. SUBSX на стр. 514.

SUBSX

Инструкция: Вычесть величину со знаком плюс C, из другой величины со знаком.

SUBSX ***SValue1***, ⟨#⟩ ***SValue2***

Результат: Разность знакового *SValue1* и знакового *SValue2* плюс C сохраняется в *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину, уменьшаемую на *SValue2* плюс C, и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) регистр или 9-битная константа, величина которого плюс C вычитается из *SValue1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
110111 001i 1111 dddddddd ssssssss	Z & (D–(S+C) = 0)	Знаков.переполнение	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0001; 1	0	0	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$0000_0001; 1	1	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	x	1	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	0	wz wc	$0000_0000; 0	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	x	1	wz wc	$FFFF_FFFF; -1	0	0
$8000_0001; -2147483647	$0000_0001; 1	x	0	wz wc	$8000_0000; -2147483648	0	0
$8000_0001; -2147483647	$0000_0001; 1	x	1	wz wc	$7FFF_FFFF; 2147483647	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	x	0	wz wc	$8000_0000; -2147483648	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	x	1	wz wc	$7FFF_FFFF; 2147483647	0	0

Описание

SUBSX (Subtract Signed, Extended) вычитает знаковую величину *SValue2* плюс C из *SValue1*и сохраняет результат в регистре *SValue1*. Инструкция SUBSX используется для выполнения операций вычитания над числами со знаком, размером в несколько двойных слов (*multi-long*), например, 64-битное вычитание.

При выполнении вычитания *multi-long* чисел со знаком, превая инструкция является беззнаковой (напр.: SUB), любые промежуточные инструкции являются беззнаковыми, расширенными (напр.: SUBX), а последняя – знаковая, расширенная (напр.: SUBSX). Не

забывайте использовать воздействие **WC**, и, опционально, **WZ**, для предшествующих **SUB** и **SUBX** инструкций.

К примеру, вычитание двух 64-битных величин со знаком выглядит следующим образом:

```
  sub     XLow,  YLow   wc wz   'Subtract low longs; save C and Z
  subsx   XHigh, YHigh          'Subtract high longs
```

после выполнения приведенного кода, 64-битный результат находится в 32-битных регистрах *XHigh*:*XLow*. Если сначала *XHigh*:*XLow* было равно $0000_0000:0000_0001 (т.е. 1), а *YHigh*:*YLow* было $0000_0000:0000_0002 (т.е. 2), то результат в *XHigh*:*XLow* будет $FFFF_FFFF:FFFF_FFFF (т.е. -1). Это объясняется далее.

```
                          Hexadecimal              Decimal
                        (high)     (low)
    (XHigh:XLow)      $0000_0000:0000_0001                  1
  - (YHigh:YLow)    - $0000_0000:0000_0002      -           2
                    ----------------------      --------------
                    = $FFFF_FFFF:FFFF_FFFF      =          -1
```

Вычитание 96-битных величин со знаком будет выглядеть похоже, с дополнительной инструкцией **SUBX**, вставленной между инструкциями **SUB** и **SUBSX**:

```
  sub     XLow,  YLow   wc wz 'Subtract low longs; save C and Z
  subx    XMid,  YMid   wc wz 'Subtract middle longs; save C and Z
  subsx   XHigh, YHigh        'Subtract high longs
```

Конечно, может быть необходимо указывать воздействия **WC** и **WZ** для последней инструкции, **SUBSX**, с тем, чтобы проследить возможные обнуление либо переполнение результата. Отметьте, что во время этой многошаговой операции флаг Z всегда отражает, равен ли результат нолю, а флаг C отображает беззнаковые заемы вплоть до последней инструкции, для которой он отображает знаковое переполнение.

Для инструкции **SUBSX**, при установленном воздействии **WZ,** флаг Z устанавливается в 1, если Z был установлен ранее и *SValue1* − (*SValue2* + C) равно нолю (используйте **WC** и **WZ** в предыдущих инструкциях **SUB** и **SUBX**). При указанном воздействии **WC**, флаг C устанавливается в 1, если в результате вычитания возникло знаковое переполнение. Результат записывается в *SValue1*, если не указано воздействие **NR**.

SUBX

Инструкция: Вычесть беззнаковую величину плюс C из другой беззнаковой.

SUBX *Value1*, ⟨#⟩ *Value2*

Результат: Разность беззнаковой *Value1*, и беззнаковой *Value2* плюс флаг сохраняется в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину, уменьшаемую на *Value2* плюс C from, и являющийся приемником для записи результата.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого плюс C вычитается из *Value1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
110011 001i 1111	ddddddddd	sssssssss	Z & (D–(S+C) = 0)	Беззнаков. заем	Записан	4

Вход					Выход		
Приемник[1]	Источник[1]	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0001; 1	0	0	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$0000_0001; 1	1	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	×	1	wz wc	$FFFF_FFFF; 4294967295	0	1

[1] **Источник и Приемник рассматриваются как величины без знака.**

Описание

SUBX (Subtract Extended) вычитает беззнаковую величину *Value2* плюс C из беззнаковой *Value1* и сохраняет результат в регистре *Value1*. Инструкция SUBX используется для выполнения операций вычитания над числами размером в несколько двойных слов (*multi-long* числами), например, 64-битное вычитание.

При выполнении операций с *multi-long* числами, первая инструкция всегда беззнаковая (напр.: SUB), любые промежуточные – беззнаковые расширенные (напр.: SUBX), а последняя – беззнаковая расширенная (SUBX) либо знаковая расширенная (SUBSX), в зависимости от природы самих *multi-long* величин. В этой секции мы будем рассматривать беззнаковое вычитание; примеры вычитания *multi-long* величин со знаком см. в секции SUBSX, на стр. 514. Не забывайте указывать воздействия WC, и, опионально, WZдля предшествующих инструкций SUB и SUBX.

К примеру, беззнаковое вычитание двух 64-битных величин может выглядеть следующим образом:

```
  sub     XLow,  YLow   wc wz    'Subtract low longs; save C and Z
  subx    XHigh, YHigh          'Subtract high longs
```

После выполнения этого примера, 64-битный результат находится в 32-битных регистрах *XHigh*:*XLow*. Если в начале *XHigh*:*XLow* была равна $0000_0001:0000_0000 (т.е. 4294967296), а *High*:*YLow* была $0000_0000:0000_0001 (т.е. 1), то результат в *XHigh*:*XLow* будет равен $0000_0000:FFFF_FFFF (т.е. 4294967295). Это объясняется ниже.

```
                          Hexadecimal              Decimal
                            (high)     (low)
   (XHigh:XLow)          $0000_0001:0000_0000       4,294,967,296
 - (YHigh:YLow)        - $0000_0000:0000_0001     -             1
                       ---------------------      --------------
                       = $0000_0000:FFFF_FFFF     = 4294967295
```

Конечно, может стать необходимо указать воздействия **WC** и **WZ**для последней инструкции, **SUBX**, с тем, чтобы проследить обнуление результата либо событие беззнакового заема.

Для **SUBX,** при установленном воздействии **WZ,** флаг Z устанавливается в 1, если Z был установлен ранее, и *Value1* − (*Value2* + C) равно нолю (используйте **WC** и **WZ** в предыдущих инструкциях **SUB** и **SUBX**). При указанном воздействии **WC**, флаг C устанавливается в 1, если в результате вычитания возникло беззнаковое переполнение. Результат записывается в *Value1*, если не указано воздействие **NR**.

SUMC

Инструкция: Сложить величину со знаком с другой величиной, знак которой инвертируется в зависимости от флага C.

SUMC *SValue1*, ⟨#⟩ *SValue2*

Результат: Сумма знаковой *SValue1* и ±*SValue2* сохраняется в регистре *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину для суммирования с *–SValue2* либо с *SValue2*, и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) – регистр или 9-битная константа, величина которого изменяет знак согласно C и складывается с *SValue1*.

–ИНСТР– ZCRI	–УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
100100 001i	1111	ddddddddd	sssssssss	D ± S = 0	Знаков. Переполнен.	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0001; 1	–	0	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	1	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFF; -1	–	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	0	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	1	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0001; 1	–	0	wz wc	$0000_0000; 0	1	0
$8000_0000; -2147483648	$0000_0001; 1	–	0	wz wc	$8000_0001; -2147483647	0	0
$8000_0000; -2147483648	$0000_0001; 1	–	1	wz wc	$7FFF_FFFF; 2147483647	0	1
$8000_0000; -2147483648	$FFFF_FFFF; -1	–	0	wz wc	$7FFF_FFFF; 2147483647	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	–	0	wz wc	$7FFF_FFFE; 2147483646	0	0
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	–	1	wz wc	$8000_0000; -2147483648	0	1
$7FFF_FFFF; 2147483647	$0000_0001; 1	–	0	wz wc	$8000_0000; -2147483648	0	1

Описание

SUMC (Sum with C-affected sign) складывает знаковую величину *SValue1* с *–SValue2* (при C = 1), либо с *SValue2* (при C = 0), и сохраняет результат в регистре *SValue1*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *SValue1* ± *SValue2* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если в результате суммирования возникло знаковое переполнение. Результат записывается в *SValue1*, если не указано воздействие NR.

SUMNC

Инструкция: Сложить величину со знаком с другой величиной, знак которой инвертируется в зависимости от !C.

SUMNC *SValue1*, ⟨#⟩ *SValue2*

Результат: Сумма знаковой *SValue1* и ±*SValue2* сохраняется в регистре *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину для суммирования с –*SValue2* либо с *SValue2*, и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) – регистр или 9-битная константа, величина которого изменяет знак согласно !C и складывается с *SValue1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
100101 001i 1111 ddddddddd sssssssss	D ± S = 0	Знаков. Переполнен.	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0001; 1	–	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	–	1	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$FFFF_FFFF; -1	–	1	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	1	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$0000_0001; 1	–	1	wz wc	$0000_0000; 0	1	0
$8000_0000; -2147483648	$0000_0001; 1	–	0	wz wc	$7FFF_FFFF; 2147483647	0	1
$8000_0000; -2147483648	$0000_0001; 1	–	1	wz wc	$8000_0001; -2147483647	0	0
$8000_0000; -2147483648	$FFFF_FFFF; -1	–	1	wz wc	$7FFF_FFFF; 2147483647	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	–	0	wz wc	$8000_0000; -2147483648	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	–	1	wz wc	$7FFF_FFFE; 2147483646	0	0
$7FFF_FFFF; 2147483647	$0000_0001; 1	–	1	wz wc	$8000_0000; -2147483648	0	1

Описание

SUMNC (Sum with !C-affected sign) складывает знаковую величину *SValue1* с *SValue2* (при C = 1), либо с –*SValue2* (при C = 0), и сохраняет результат в регистре *SValue1*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *SValue1* ± *SValue2* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если в результате суммирования возникло знаковое переполнение. Результат записывается в *SValue1*, если не указано воздействие **NR**.

SUMNZ

Инструкция: Сложить величину со знаком с другой величиной, знак которой инвертируется в зависимости от !Z.

SUMNZ *SValue1*, ⟨#⟩ *SValue2*

Результат: Сумма знаковой *SValue1* и ±*SValue2* сохраняется в регистре *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину для суммирования с *–SValue2* либо с *SValue2*, и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) – регистр или 9-битная константа, величина которого изменяет знак согласно !Z и складывается с *SValue1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
100111 001i 1111	ddddddddd	sssssssss	D ± S = 0	Знаков. Переполнен.	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0001; 1	0	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	1	–	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$FFFF_FFFF; -1	1	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	–	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$0000_0001; 1	1	–	wz wc	$0000_0000; 0	1	0
$8000_0000; -2147483648	$0000_0001; 1	0	–	wz wc	$7FFF_FFFF; 2147483647	0	1
$8000_0000; -2147483648	$0000_0001; 1	1	–	wz wc	$8000_0001; -2147483647	0	0
$8000_0000; -2147483648	$FFFF_FFFF; -1	1	–	wz wc	$7FFF_FFFF; 2147483647	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	0	–	wz wc	$8000_0000; -2147483648	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	1	–	wz wc	$7FFF_FFFE; 2147483646	0	0
$7FFF_FFFF; 2147483647	$0000_0001; 1	1	–	wz wc	$8000_0000; -2147483648	0	1

Описание

SUMNZ (Sum with !Z-affected sign) складывает знаковую величину *SValue1* с *SValue2* (при Z = 1), либо с *–SValue2* (при Z = 0), и сохраняет результат в регистре *SValue1*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *SValue1* ± *SValue2* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если в результате суммирования возникло знаковое переполнение. Результат записывается в *SValue1*, если не указано воздействие **NR**.

SUMZ

Инструкция: Сложить величину со знаком с другой величиной, знак которой инвертируется в зависимости от флага Z.

SUMZ *SValue1*, ⟨#⟩ *SValue2*

Результат: Сумма знаковой *SValue1* и ±*SValue2* сохраняется в регистре *SValue1*.

- ***SValue1*** (d-поле) – регистр, содержащий величину для суммирования с *–SValue2* либо с *SValue2*, и являющийся приемником для записи результата.
- ***SValue2*** (s-поле) – регистр или 9-битная константа, величина которого изменяет знак согласно Z и складывается с *SValue1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
100110 001i 1111 ddddddddd sssssssss	D ± S = 0	Знаков. Переполнен.	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0001; 1	$0000_0001; 1	0	–	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	1	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFF; -1	0	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	–	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0001; 1	0	–	wz wc	$0000_0000; 0	1	0
$8000_0000; -2147483648	$0000_0001; 1	0	–	wz wc	$8000_0001; -2147483647	0	0
$8000_0000; -2147483648	$0000_0001; 1	1	–	wz wc	$7FFF_FFFF; 2147483647	0	1
$8000_0000; -2147483648	$FFFF_FFFF; -1	0	–	wz wc	$7FFF_FFFF; 2147483647	0	1
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	0	–	wz wc	$7FFF_FFFE; 2147483646	0	0
$7FFF_FFFF; 2147483647	$FFFF_FFFF; -1	1	–	wz wc	$8000_0000; -2147483648	0	1
$7FFF_FFFF; 2147483647	$0000_0001; 1	0	–	wz wc	$8000_0000; -2147483648	0	1

Описание

SUMZ (Sum with Z-affected sign) складывает знаковую величину *SValue1* с *–SValue2* (при Z = 1), либо с *SValue2* (при Z = 0), и сохраняет результат в регистре *SValue1*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *SValue1* ± *SValue2* равно нолю. При указанном воздействии WC, флаг C устанавливается в 1, если в результате суммирования возникло знаковое переполнение. Результат записывается в *SValue1*, если не указано воздействие NR.

Символы

Символы, приведенные ниже в Табл.5-6, служат для выполнения одной или нескольких функций в коде на языке *Propeller*-ассемблер. Для получения информации по символам языка Spin, см. Символы, на стр. 360. В Табл.5-6 дано краткое описание назначения каждого из символов, с сылками на подробные описания с примерами.

Табл. 5-6: Символы

Символ	Назначение
%	Указатель двоичного: используется для указания, что это значение приведено в двоичном формате (основание 2). См. «Представление величин» на стр. 183.
%%	Указатель четверичного: используется для указания, что это значение приведено в четверичном формате (основание 4). См. «Представление величин» на стр. 183.
$	1) Указатель шестнадцатеричного: используется для указания, что это значение приведено в шестнадцатеричном формате (основание 16). См. «Представление величин» на стр. 183 2) Ассемблерный указатель «Здесь»: используется для указания на текущий адрес в ассемблерных инструкциях. См. JMP на стр. 458.
''	Указатель строки: используется в начале и конце строки текстовых символов. См. «Блоки Данных» на стр. 243.
_	1) Разделитель: используется как разделитель групп чисел в константах (где запятая ‘,’ или точка ‘.’ используются как обычный разделитель групп чисел). См. «Представление величин» на стр. 183. 2) Подчеркивание: используется как часть идентификатора. См. «Правила Идентификаторов» на стр. 183.
#	Ассемблерный признак константы: используется для обозначения того, что выражение либо идентификатор является константой, а не адресом регистра. См. «Где инструкция берет свои данные», на стр. 396
:	Индикатор ассемблерной локальной метки: вводится непосредственно перед локальной меткой. См. «Глобальные и Локальные метки» на стр. 398.
,	Разделитель списка: используется для разделения элементов в списках. См. DAT, Объявление Данных (Синтаксис 1) на стр. 243.
'	Указатель комментария кода: используется для ввода внутристроковых комментариев кода (некомпилируемый текст) в целях просмотра текста. См. «Упражнение 3: Output.spin» на стр. 118.

' '	Указатель комментария документации: используется для ввода внутристроковых комментариев документации (некомпилируемый текст) для просмотра документации. См. «Упражнение 3: Output.spin» на стр. 118.
{ }	Указатель однострочковых (In-line)/многострочковых (multi-line) комментариев кода: используется для ввода многострочковых комментариев кода (некомпилируемый текст) для просмотра кода.
{{ }}	Указатель однострочковых (In-line)/многострочковых (multi-line) комментариев: используется для ввода многострочковых комментариев документации (некомпилируемый текст) для просмотра документации. См. «Упражнение 3: Output.spin» на стр. 118.

TEST

Инструкция: Побитовое И двух величин с влиянием только на флаги.

TEST *Value1*, ⟨#⟩ *Value2*

Результат: Опционально, признак ноля и четность записываются в флаги Z и C.

- ***Value1*** (d-поле) – регистр, содержащий величину для побитного И с *Value2*.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого умножается по И с *Value1*.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
011000 000i 1111 ddddddddd sssssssss	D = 0	Четность результата	Не записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
$0000_000A; 10	$0000_0005; 5	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_0007; 7	–	–	wr wz wc	$0000_0002; 2	0	1
$0000_000A; 10	$0000_000F; 15	–	–	wr wz wc	$0000_000A; 10	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR. ОТМЕТЬТЕ: инструкция TEST с воздействием WR становится инструкцией AND.**

Описание

TEST эквивалентна инструкции AND, за исключением того, что она не записывает результат в регистр *Value1*; она выполняет Побитовое И величин *Value1* и *Value2*, и опционально сохраняет признак ноля и четность в флаги Z и C.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *Value1* И *Value2* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если результат содержит нечетное количество установленных в 1 битов.

TESTN

Инструкция: Побитовое И величины с инверсным значением другой величины и влиянием только на флаги.

TESTN *Value1*, ⟨#⟩ *Value2*

Результат: Опционально, признак ноля и четность записываются в флаги Z и C.

- ***Value1*** (d-поле) – регистр, содержащий величину для побитного И с *Value2*.
- ***Value2*** (s-поле) регистр или 9-битная константа, инверсная величина которого (побитовое НЕ) умножается по И с *Value1*

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
011001 000i 1111 ddddddddd sssssssss	D = 0	Четность результата	Не записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
$F731_125A; -147778982	$FFFF_FFFA; -6	–	–	wr wz wc	$0000_0000; 0	1	0
$F731_125A; -147778982	$FFFF_FFF8; -8	–	–	wr wz wc	$0000_0002; 2	0	1
$F731_125A; -147778982	$FFFF_FFF0; -16	–	–	wr wz wc	$0000_000A; 10	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR. ОТМЕТЬТЕ: инструкция TESTN с воздействием WR становится инструкцией ANDN.**

Описание

TESTN эквивалентна **ANDN** за исключением того, что она не записывает результат в регистр *Value1*; она выполняет побитовое И НЕ величины *Value2* с *Value1* и опционально сохраняет признак ноля и четность результата во флаги Z и C.

При установленном воздействии **WZ**, флаг Z устанавливается в 1, если *Value1* И НЕ *Value2* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если результат содержит нечетное количество установленных в 1 битов.

TJNZ

Инструкция: Проверить величину и перейти по адресу, если не ноль.

TJNZ ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-поле) – регистр для проверки.
- ***Address*** (s-поле) – регистр или 9-битная константа, значение которой представляет собой адрес перехода, когда *Value* содержит ненулевое значение.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
111010 000i 1111 ddddddddd sssssssss	D = 0	0	Не записан	4 or 8

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
$0000_0000; 0	$----_----; -	-	-	wr wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$----_----; -	-	-	wr wz wc	$0000_0001; 1	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR.**

Описание

TJNZ проверяет регистр *Value* и переходит по *Address*, если он содержит ненулевое значение.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если регистр *Value* содержит ноль.

TJNZ требует различное количество тактов, в зависимости от того, необходимо ли выполнять переход. При переходе она выполняется 4 такта, если перехода нет – выполнение занимает 8 тактов. Поскольку циклы, использующие TJNZ должны быть быстрыми, эта инструкция таким образом оптимизирована по скорости.

TJZ

Инструкция: Проверить величину и перейти по адресу, если ноль.

TJZ ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-поле) – регистр для проверки.
- ***Address*** (s-поле) – регистр или 9-битная константа, значение которой представляет собой адрес перехода, когда *Value* содержит нулевое значение.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
111011 000i 1111 ddddddddd sssssssss	D = 0	0	Не записан	4 or 8

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
s0000_0000; 0	s----_----; -	-	-	wr wz wc	s0000_0000; 0	1	0
s0000_0001; 1	s----_----; -	-	-	wr wz wc	s0000_0001; 1	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR.**

Описание

TJZ проверяет регистр *Value* и переходит по *Address*, если он содержит ноль.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если регистр *Value* содержит ноль.

TJZ требует различное количество тактов, в зависимости от того, необходимо ли выполнять переход. При переходе она выполняется 4 такта, если перехода нет – выполнение занимает 8 тактов.

VCFG

Регистр: Регистр настройки видеогенератора.

```
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   VCFG, ОперандИст ⟨Воздействия⟩
DAT
   ⟨Метка⟩ ⟨Условие⟩ Инструкция   ОперандПрм, VCFG ⟨Воздействия⟩
```

Результат: Обновление регистра конфигурации видеогенератора (опционально).

- ***Метка*** – метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистр VCFG может использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандИст*** – выражение, используемое инструкцией для выполнения действия, и при необходимости – записи в регистр VCFG в ***ОперандПрм***.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра VCFG из поля операнда-источника.

Описание

VCFG – это один из двух регистров (VCFG и VSCL), которые влияют на работу Видео-Генератора процессора. Каждый *cog* имеет модуль видеогенератора, который позволяет передавать сигнал видеоизображения на постоянной скорости. Регистр VCFG содержит настройки видеогенератора.

В следующем примере приведен код, настраивающий видеогенератор на передачу композитного видеосигнала в режиме 1 с четырьмя цветами, узкополосный сигнал цвета, на группе выводов 1, нижних 4-х линиях (то есть линиях P11:8).

```
                mov     vcfg,  VidCfg

VidCfg    long  %0_10_1_0_1_000_00000000000_001_0_00001111
```

Для дополнительной информации см. Регистры, стр. 499, и Spin, секция VCFG, стр. 368.

VSCL

Регистр: Регистр настройки масштаба видео.

DAT
〈***Метка***〉 〈***Условие***〉 ***Инструкция*** VSCL, ***ОперандИст*** 〈***Воздействия***〉
DAT
〈***Метка***〉 〈***Условие***〉 ***Инструкция*** ***ОперандПрм,*** VSCL 〈***Воздействия***〉

Результат: Обновление регистра настройки масштаба видео (опционально).

- ***Метка*** – метка выражения. См. «Общие элементы синтаксиса», стр.408.
- ***Условие*** – условие выполнения. См. «Общие элементы синтаксиса», стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция. Регистр VSCL может использоваться в поле и операнда-источника, и опреанда-приемника.
- ***ОперандИст*** – выражение, используемое инструкцией для выполнения действия, и при необходимости – записи в регистр VSCL в ***ОперандПрм***.
- ***ОперандПрм*** – выражение, представляющее регистр, над которым производится действие и в который может быть произведена запись с использованием значения регистра VSCL из поля операнда-источника.

Описание

VSCL – это один из двух регистров (VCFG и VSCL), которые влияют на работу Видео-Генератора процессора. Каждый *cog* имеет модуль видеогенератора, который позволяет передавать сигнал видеоизображения на постоянной скорости. Регистр VSCL устанавливает скорость, на которой происходит генерация видеоданных.

В следующем примере приведен код, настраивающий регистр масштаба видео на 160 импульсов пикселей и 2560 импульсов кадров (для кадра в 16-пикселей на 2-битный цвет). Безусловно, реальная скорость, на которой будет происходить выдача импульсов, зависит от частоты PLL, в комбинации с этим масштабным фактором.

```
                mov     vcfg,  VscCfg

VscCfg    long  %000000000000_10100000_101000000000
```

Для дополнительной информации см. Регистры, стр. 499, и Spin, секция VSCL, стр. 371.

WAITCNT

Инструкция: Временно приостановить выполнение программы в процессоре.

WAITCNT *Target*, ⟨#⟩ *Delta*

Результат: *Target* + *Delta* сохраняется в *Target*.

- ***Target*** (d-поле) –регистр с необходимой величиной для сравнения с Системным Счетчиком (CNT). Когда Системный Счетчик достигает величины *Target*, к ней прибавляется *Delta,* и выполнение продолжается со следующей инструкции.
- ***Delta*** (s-поле) – регистр или 9-битная константа, значение которого прибавляется к величине *Target* при подготовке к следующей инструкции WAITCNT. Тем самым создается окно синхронизированной задержки.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
111110 001i 1111 ddddddddd sssssssss	Результат = 0	Беззнаков. перенос	Записан	5+

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; 4294967295	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

Описание

WAITCNT, “Wait for System Counter” – это одна из четырех инструкций ожидания (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки выполнения программы в процессоре до достижения условия. Инструкция WAITCNT приостанавливает процессор до тех пор, пока глобальный Системный Счетчик не станет равным значению в регистре *Target*, затем она добавляет *Delta* к *Target* и выполнение продолжается со следующей инструкции. Инструкция WAITCNT выполняется аналогично команде для Синхронизированных Задержек WAITCNT языка *Spin*; см. WAITCNT на стр. 373.

При установленном WZ, флаг Z устанавливается в 1, если сумма *Target* и *Delta* равна 0. При указанном WC, флаг C устанавливается в 1, если сумма *Target* и *Delta* приводит к 32-битному переполнению. Результат записывается в *Target*, если не указано NR.

WAITPEQ

Инструкция: Приостановить процессор, до получения заданной комбинации на линиях B/B.

WAITPEQ ***State***, ⟨#⟩ ***Mask***

- ***State*** (d-поле) – регистр, содержащий заданные состояния, сравниваемые с INx, умноженным по И с *Mask*.
- ***Mask*** (s-поле) – регистр или 9-битная константа, значение которой побитно умножается по И с INx перед сравнением со *State*.

–ИНСТР– ZCRI	–УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
111100 000i	1111	ddddddddd	sssssssss	---	---	Не записан	5+

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
s0000_0000; 0	s0000_0000; 0	–	–	wr wz wc	s0000_0000; 0	1	0
s0000_0000; 0	s0000_0001; 1	–	–	wr wz wc	s0000_0001; 1	0	0
s0000_0001; 1	s0000_0001; 1	–	–	wr wz wc	s0000_0002; 2	0	0
s0000_0000; 0	s0000_0002; 2	–	–	wr wz wc	s0000_0002; 2	0	0
s0000_0002; 2	s0000_0002; 2	–	–	wr wz wc	s0000_0004; 4	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR.**

Описание

WAITPEQ, "Wait for Pin(s) to Equal" – это одна из четырех инструкций ожидания (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки выполнения программы в процессоре до достижения условия. Инструкция WAITPEQ приостанавливает процессор до тех пор, пока результат (INx И *Mask*) не станет равным величине в регистре *State*. INx – это либо INA, либо INB – в зависимости от состояния флага C при выполнении: INA при C = 0, INB при C = 1 (P8X32A – исключение, он всегда проверяет INA).

Инструкция WAITPEQ выполняется аналогично команде WAITPEQ языка *Spin*, стр. 377.

WAITPNE

Инструкция: Приостановить процессор, пока на линиях B/B присутствует заданная комбинация.

WAITPNE *State*, ⟨#⟩ *Mask*

- ***State*** (d-поле) – регистр, содержащий заданные состояния, сравниваемые с INx, умноженным по И с *Mask*.
- ***Mask*** (s-поле) – регистр или 9-битная константа, значение которой побитно умножается по И с INx перед сравнением со *State*.

–ИНСТР– ZCRI	–УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов

111101 000i 1111 ddddddddd sssssssss	---	---	Не записан	5+

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wr wz wc	$0000_0000; 0	1	1
$0000_0000; 0	$0000_0001; 1	–	–	wr wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wr wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0002; 2	–	–	wr wz wc	$0000_0003; 3	0	0
$0000_0002; 2	$0000_0002; 2	–	–	wr wz wc	$0000_0002; 2	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR.**

Описание

WAITPNE, "Wait for Pin(s) to Not Equal," – это одна из четырех инструкций ожидания (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки выполнения программы в процессоре до достижения условия. Инструкция WAITPNE приостанавливает процессор до тех пор, пока результат (INx И *Mask*) не станет отличным от величины в регистре *State*. INx – это либо INA, либо INB – в зависимости от состояния флага C при выполнении: INA при C = 0, INB при C = 1 (P8X32A – исключение, он всегда проверяет INA).

Инструкция WAITPNE выполняется аналогично команде WAITPNE языка *Spin*, стр. 377.

WAITVID

Инструкция: Приостановить процессор, пока Генератор Видео не будет готов принять данные пикселей.

WAITVID *Colors*, ⟨#⟩ *Pixels*

- ***Colors*** (d-поле) – регистр с четырехбайтными значениями цвета, каждый описывающий четыре возможных цвета набора пикселей в *Pixels*.
- ***Pixels*** (s-поле) – регистр или 9-битная константа, значение которого – это следующий набор 16-пикселов по 2-бита (или 32-пиксела на 1-бит) для вывода.

–ИНСТР– ZCRI –УСЛ– –ПРМ– –ИСТ–	Z Результат	C Результат	Результат	Тактов
111111 000i 1111 ddddddddd sssssssss	D + S = 0	Беззн. переполнение	Не записан	5+

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z	C

$0000_0002; 2	$FFFF_FFFD; -3	-	-	wr wz wc	$FFFF_FFFF; -1	0	0
$0000_0002; 2	$FFFF_FFFE; -2	-	-	wr wz wc	$0000_0000; 0	1	1
$0000_0002; 2	$FFFF_FFFF; -1	-	-	wr wz wc	$0000_0001; 1	0	1
$0000_0002; 2	$0000_0000; 0	-	-	wr wz wc	$0000_0002; 2	0	0

[1] **Приемник не перезаписывается, если не задано воздействие WR.**

Описание

WAITVID, "Wait for Video Generator," – это одна из четырех инструкций ожидания (WAITCNT, WAITPEQ, WAITPNE, и WAITVID), используемых для приостановки выполнения программы в процессоре до достижения условия. Инструкция WAITVID приостанавливает процессор до тех пор, пока аппаратный Генератор Видео не будет готов к приему следующих данных (*Colors* и *Pixels*), после чего процессор продолжит выполнение со следующей инструкции. Инструкция WAITVID выполняется аналогично команде WAITVID языка *Spin*; см. WAITVID на стр. 380.

При заданном **WZ,** флаг Z устанавливается в 1, если *Colors* и *Pixels* равны.

Перед выполнением инструкции WAITVID убедитесь, что модуль видеогенератора и счетчик A процессора работают, иначе она будет выполняться бесконечно долго. См. VCFG на стр. 526 и VSCL на стр. 527, а также CTRA, CTRB на стр. 447.

WC

Воздействие: Принуждает ассемблерную инструкцию изменять флаг C.

⟨Метка⟩ ⟨Условие⟩ Инструкция Операнды **WC**

Результат: Флаг C обновляется статусом выполнения *Инструкции*.

- ***Метка*** – опциональная метка. См. «Общие элементы синтаксиса», стр.408.
- ***Condition*** – опциональное условие. См. «Общие элементы синтаксиса» на стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция.
- ***Операнды*** – это ноль, один либо два операнда, в зависимости от инструкции.

Описание

WC (Write C flag) – это одно из четырех опциональных воздействий (NR, WR, WZ и WC), которые влияют на поведение ассемблерных инструкций. Воздействие WC принуждает выполняемую ассемблерную инструкцию изменять флаг C в соответствии с результатом ее выполнения.

Например, инструкция CMP (Compare Unsigned) сравнивает две величины (источник и приемник) но автоматически не записывает результаты во флаги C и Z. Вы можете определить, является ли величина приемника меньше величины источника, используя инструкцию CMP с воздействием WC:

```
cmp     value1, value2   WC      'C = 1 if value1 < value2
```

Приведенная в примере инструкция CMP сравнивает value1 с value2 и устанавливает флаг C в 1, если value1 меньше, чем value2.

См. «Воздействия», стр. 450, для дополнительной информации.

WR

Воздействие: Принуждает ассемблерную инструкцию записать результат.

⟨Метка⟩ ⟨Условие⟩ Инструкция Операнды WR

Результат: В регистр приемника *Инструкции* записывается величина результата.

- ***Метка*** – опциональная метка. См. «Общие элементы синтаксиса», стр.408.
- ***Condition*** – опциональное условие. См. «Общие элементы синтаксиса» на стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция.
- ***Операнды*** – это ноль, один либо два операнда, в зависимости от инструкции.

Описание

WR (Write Result) – это одно из четырех опциональных воздействий (NR, WR, WZ и WC), которые влияют на поведение ассемблерных инструкций. Воздействие WR принуждает исполняемую инструкцию записать свой результат в регистр назначения.

Например, инструкция COGINIT (Cog Initialize), по умолчанию не записывает результат в регистр приемника. Это не проблема, когда нам неоходимо запустить процессор с определенным ID-номером, но когда мы хотим запустить следующий доступный cog (т.е.: бит 3 в регистре приемника установлен в 1), то может понадобиться узнать, какой именно процессор стартовал, его ID. Мы можем получить ID стартовавшего процессора из инструкции COGINIT, используя воздействие WR:

```
coginit   launch_value   WR      'Launch new cog, get ID back
```

Предположив, что launch_value указывает на регистр, содержащий единицу в бите 3, инструкция COGINIT запускает следующий доступный *cog* и записывает его ID назад в регистр launch_value.

См. «Воздействия», стр.450, для дополнительной информации.

WRBYTE

Инструкция: Записать байт в Основную Память.

WRBYTE ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-поле) – регистр, содержащий 8-битное значение для записи.
- ***Address*** (s-поле) регистр или 9-битная константа, величина которого является адресом для записи.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000000 000i 1111	ddddddddd	sssssssss	---	---	Не записан	7..22

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	WZ WC	n/a	1	0

[1] Приемник на выходе не существует, поскольку указание воздействия WR превратило бы инструкцию WRBYTE в RDBYTE.

[2] Флаг Z всегда устанавливается в 1, за исключением случая, когда адрес основной памяти (биты 13:0) находится на границе двойных слов.

Описание

WRBYTE синхронизируется с *Hub* и записывает младший байт регистра *Value* в Основную Память по адресу *Address*.

Воздействие WR не должно использоваться совмесно с инструкцией WRBYTE, поскольку оно заменит ее на инструкцию RDBYTE.

WRBYTE – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

WRLONG

Инструкция: Записать двойное слово (*long*) в Основную Память.

WRLONG ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-поле) – регистр, содержащий 32-битное значение для записи.
- ***Address*** (s-поле) регистр или 9-битная константа, величина которого является адресом для записи.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000010 000i	1111	ddddddddd sssssssss	---	---	Не записан	7..22

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	n/a	0	0

[1] Приемник на выходе не существует, поскольку указание воздействия WR превратило бы инструкцию WRLONG в RDLONG.
[2] Флаг Z всегда сброшен в 0, поскольку адрес основной памяти (биты 13:2) всегда находится на границе двойных слов

Описание

WRLONG синхронизируется с *Hub* и записывает *long* из регистра *Value* в Основную Память по адресу *Address*.

Воздействие WR не должно использоваться совместно с инструкцией WRLONG, поскольку оно заменит ее на инструкцию RDRLONG.

WRLONG – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

WRWORD

Инструкция: Записать двойное слово (*word*) в Основную Память.

WRWORD ***Value***, ⟨#⟩ ***Address***

- ***Value*** (d-поле) – регистр, содержащий 16-битное значение для записи.
- ***Address*** (s-поле) регистр или 9-битная константа, величина которого является адресом для записи.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
000001 000i 1111	ddddddddd	sssssssss	---	---	Не записан	7..22

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	WZ WC	n/a	1	0

[1] Приемник на выходе не существует, поскольку указание воздействия WR превратило бы инструкцию WRWORD в RDWORD.

[2] Флаг Z всегда устанавливается в 1, за исключением случая, когда адрес основной памяти (биты 13:1) находится на границе двойных слов.

Описание

WRWORD синхронизируется с *Hub* и записывает *word* из регистра *Value* в Основную Память по адресу *Address*.

Воздействие WR не должно использоваться совмесно с инструкцией WRWORD, поскольку оно заменит ее на инструкцию RDWORD.

WRWORD – это *Hub*-инструкция. *Hub*-инструкциям необходимо от 7 до 22 тактов для выполнения, в зависимости от взаимного расположения окна доступа процессора к *Hub* и моментом выполнения инструкции во времени. См. Переключатель (*Hub*) на стр. 29 для более детальной информации.

WZ

Воздействие: Принуждает ассемблерную инструкцию изменять флаг Z.

⟨***Метка***⟩ ⟨***Условие***⟩ ***Инструкция Операнды*** WZ

Результат: Флаг Z обновляется статусом выполнения *Инструкции*.

- ***Метка*** – опциональная метка. См. «Общие элементы синтаксиса», стр.408.
- ***Condition*** – опциональное условие. См. «Общие элементы синтаксиса» на стр.408.
- ***Инструкция*** – необходимая ассемблерная инструкция.
- ***Операнды*** – это ноль, один либо два операнда, в зависимости от инструкции.

Описание

WR (Write Z flag) – это одно из четырех опциональных воздействий (NR, WR, WZ и WC), которые влияют на поведение ассемблерных инструкций. Воздействие WZ принуждает исполняемую ассемблерную инструкцию изменять флаг Z в соответствии с результатом ее выполнения.

Например, инструкция CMP (Compare Unsigned) сравнивает две величины (приемника и источника), но автоматически не записывает результаты во флаги C и Z. Вы можете определить, равна ли величина приемника величине источника, используя инструкцию CMP совместно с воздействием WZ:

```
cmp     value1, value2   WZ      'Z = 1 if value1 = value2
```

В приведенном примере инструкция CMP сравнивает величину value1 с value2 и устанавливает флаг Z в 1, если value1 равно value2.

См. «Воздействия», стр.450, для дополнительной информации.

XOR

Инструкция: Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ двух величин.

XOR *Value1*, ⟨#⟩ *Value2*

Результат: *Value1* ИСКЛЮЧАЮЩЕЕ ИЛИ *Value2* сохраняется в *Value1*.

- ***Value1*** (d-поле) – регистр, содержащий величину для выполнения ИСКЛЮЧАЮЩЕЕ ИЛИ с *Value2* и являющийся приемником для записи результата.
- ***Value2*** (s-поле) регистр или 9-битная константа, величина которого складывается по ИСКЛЮЧАЮЩЕЕ ИЛИ с величиной *Value1*.

–ИНСТР– ZCRI –УСЛ–	–ПРМ–	–ИСТ–	Z Результат	C Результат	Результат	Тактов
011011 001i 1111	ddddddddd	sssssssss	Результат = 0	Четность результата	Записан	4

Вход					Выход		
Приемник	Источник	Z	C	Возд.	Приемник	Z	C

$0000_000A; 10	$0000_0005; 5	-	-	wz wc	$0000_000F; 15	0	0
$0000_000A; 10	$0000_0007; 7	-	-	wz wc	$0000_000D; 13	0	1
$0000_000A; 10	$0000_000A; 10	-	-	wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_000D; 13	-	-	wz wc	$0000_0007; 7	0	1
$0000_000A; 10	$0000_000F; 15	-	-	wz wc	$0000_0005; 5	0	0

Описание

XOR (Побитовое exclusive OR) выполняет Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ величины из регистра *Value2* с величиной из *Value1*.

При установленном воздействии **WZ,** флаг Z устанавливается в 1, если *Value1* **XOR** *Value2* равно нолю. При указанном воздействии **WC**, флаг C устанавливается в 1, если результат содержит нечетное количество установленных в 1 битов. Результат записывается в регистр *Value1*, если не указано воздействие **NR**.

Приложение A: Список служебных слов

Эти слова являются зарезервированными как в языке *Spin*, так и в ассемблере Propeller.

Табл. A-0-1: Список служебных слов Propeller

_CLKFREQ[s]	CONSTANT[s]	IF_NC_AND_NZ[a]	MIN[a]	PLL4X[s]	SUBSX[a]
_CLKMODE[s]	CTRA[d]	IF_NC_AND_Z[a]	MINS[a]	PLL8X[s]	SUBX[a]
_FREE[s]	CTRB[d]	IF_NC_OR_NZ[a]	MOV[a]	PLL16X[s]	SUMC[a]
_STACK[s]	DAT[s]	IF_NC_OR_Z[a]	MOVD[a]	POSX[d]	SUMNC[a]
_XINFREQ[s]	DIRA[d]	IF_NE[a]	MOVI[a]	PRI[s]	SUMNZ[a]
ABORT[s]	DIRB[d#]	IF_NEVER[a]	MOVS[a]	PUB[s]	SUMZ[a]
ABS[a]	DJNZ[a]	IF_NZ[a]	MUL[a#]	QUIT[s]	TEST[a]
ABSNEG[a]	ELSE[s]	IF_NZ_AND_C[a]	MULS[a#]	RCFAST[s]	TESTN[a]
ADD[a]	ELSEIF[s]	IF_NZ_AND_NC[a]	MUXC[a]	RCL[a]	TJNZ[a]
ADDABS[a]	ELSEIFNOT[s]	IF_NZ_OR_C[a]	MUXNC[a]	RCR[a]	TJZ[a]
ADDS[a]	ENC[a]	IF_NZ_OR_NC[a]	MUXNZ[a]	RCSLOW[s]	TO[s]
ADDSX[a]	FALSE[d]	IF_Z[a]	MUXZ[a]	RDBYTE[a]	TRUE[d]
ADDX[a]	FILE[s]	IF_Z_AND_C[a]	NEG[a]	RDLONG[a]	TRUNC[s]
AND[d]	FIT[a]	IF_Z_AND_NC[a]	NEGC[a]	RDWORD[a]	UNTIL[s]
ANDN[a]	FLOAT[s]	IF_Z_EQ_C[a]	NEGNC[a]	REBOOT[s]	VAR[s]
BYTE[s]	FROM[s]	IF_Z_NE_C[a]	NEGNZ[a]	REPEAT[s]	VCFG[d]
BYTEFILL[s]	FRQA[d]	IF_Z_OR_C[a]	NEGX[d]	RES[a]	VSCL[d]
BYTEMOVE[s]	FRQB[d]	IF_Z_OR_NC[a]	NEGZ[a]	s	WAITCNT[d]
CALL[a]	HUBOP[a]	INA[d]	NEXT[s]	RET[a]	WAITPEQ[d]
CASE[s]	IF[s]	INB[d#]	NOP[a]	RETURN[s]	WAITPNE[d]
CHIPVER[s]	IFNOT[s]	JMP[a]	NOT[s]	REV[a]	WAITVID[d]
CLKFREQ[s]	IF_A[a]	JMPRET[a]	NR[a]	ROL[a]	WC[a]
CLKMODE[s]	IF_AE[a]	LOCKCLR[d]	OBJ[s]	ROR[a]	WHILE[s]
CLKSET[d]	IF_ALWAYS[a]	LOCKNEW[d]	ONES[a#]	ROUND[s]	WORD[s]
CMP[a]	IF_B[a]	LOCKRET[d]	OR[d]	SAR[a]	WORDFILL[s]
CMPS[a]	IF_BE[a]	LOCKSET[d]	ORG[a]	SHL[a]	WORDMOVE[s]
CMPSUB[a]	IF_C[a]	LONG[s]	OTHER[s]	SHR[a]	WR[a]
CMPSX[a]	IF_C_AND_NZ[a]	LONGFILL[s]	OUTA[d]	SPR[s]	WRBYTE[a]
CMPX[a]	IF_C_AND_Z[a]	LONGMOVE[s]	OUTB[d#]	STEP[s]	WRLONG[a]
CNT[d]	IF_C_EQ_Z[a]	LOOKDOWN[s]	PAR[d]	STRCOMP[s]	WRWORD[a]
COGID[d]	IF_C_NE_Z[a]	LOOKDOWNZ[s]	PHSA[d]	STRING[s]	WZ[a]
COGINIT[d]	IF_C_OR_NZ[a]	LOOKUP[s]	PHSB[d]	STRSIZE[s]	XINPUT[s]
COGNEW[s]	IF_C_OR_Z[a]	LOOKUPZ[s]	PI[d]	SUB[a]	XOR[a]
COGSTOP[d]	IF_E[a]	MAX[a]	PLL1X[s]	SUBABS[a]	XTAL1[s]
CON[s]	IF_NC[a]	MAXS[a]	PLL2X[s]	SUBS[a]	XTAL2[s]
					XTAL3[s]

a = элемент ассемблера; s = элемент Spin; d = доступен в обоих языках, # - зарезервирован

Приложение B: Математические примеры и таблицы функций

Умножение, деление и вычисление квадратного корня.

Операции умножения, деления, а также вычисления квадратного корня могут быть осуществлены с использованием инструкций сложения, вычитания и сдвига. Ниже приведен пример подпрограммы беззнакового умножения, которая выполняет умножение двух 16-битных величин и возвращает 32-битный результат:

```
'
' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
multiply        shl     x,#16           'get multiplicand into x[31..16]
                mov     t,#16           'ready for 16 multiplier bits
                shr     y,#1      wc    'get initial multiplier bit into c
:loop  if_c     add     y,x       wc    'if c set, add multiplicand to product
                rcr     y,#1      wc    'put next multiplier in c, shift prod.
                djnz    t,#:loop        'loop until done
multiply_ret    ret                     'return with product in y[31..0]
```

Время выполнения приведенной подпрограммы могло быть сокращено примерно на треть, если бы использовался развернутый цикл, без **DJNZ,** с повторяющимися инструкциями **ADD** / **RCR**.

Операция деления похожа на умножение, но наоборот. В то же время она более сложная, поскольку сравнение должно производиться до того, как выполнится вычитание. Для этого имеется специальная инструкция, **CMPSUB D,S**, которая проверяет, может ли выполниться операция вычитания, не приводя к заему. Если такая ситуация не возникнет, то выполняется вычитание и в результате C будет равен 1. Если же возникает заем (переполнение снизу), то D не изменяется, а C становится равным 0.

Ниже приведен пример подпрограммы беззнакового деления, которая выполняет деление одной 32-битной величины на другую, 16-битную, и в результате возвращает 16-битные величины результата и остатка:

```
'
```

```
' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]
'
divide          shl     y,#15           'get divisor into y[30..15]
                mov     t,#16           'ready for 16 quotient bits
:loop           cmpsub  x,y       wc    'y =< x? Subtract it, quotient bit in
c
                rcl     x,#1            'rotate c into quotient, shift
dividend
                djnz    t,#:loop        'loop until done
divide_ret      ret                     'quotient in x[15..0],
                                        'remainder in x[31..16]
```

Так же, как и в случае подпрограммы умножения, эта подпрограмма может быть представлена в виде последовательности из 16 пар инструкций **CMPSUB** + **RCL**, чтобы избавиться от **DJNZ** и сократить время выполнения примерно на треть. В таком случае можно достичь высокой скорости выполнения за счет большего размера кода.

Ниже приведена подпрограмма вычисления квадратного корня, использующая инструкцию **CMPSUB**:

```
'
' Compute square-root of y[31..0] into x[15..0]
'
root            mov     a,#0            'reset accumulator
                mov     x,#0            'reset root
                mov     t,#16           'ready for 16 root bits
:loop           shl     y,#1    wc      'rotate top two bits of y to
accumulator
                rcl     a,#1
                shl     y,#1    wc
                rcl     a,#1
                shl     x,#2            'determine next bit of root
                or      x,#1
                cmpsub  a,x     wc
                shr     x,#2
                rcl     x,#1
                djnz    t,#:loop        'loop until done
root_ret        ret                     'square root in x[15..0]
```

Многие сложные математические функции могут быть реализованы с использованием инструкций сложения, вычитания и сдвига. Здесь приведены специфические примеры, однако эти алгоритмы в различных вариациях могут использоваться для оптимального решения конкретных задач.

Таблицы Log и Anti-Log ($C000-DFFF)

Таблицы log и anti-log полезны для преобразования величин между нормальной и экспоненциальной формами их представления.

Когда числа представлены в экспоненциальной форме, простые математические операции приобретают более сложные качества. Например ‘сложение’ и ‘вычитание’ становятся ‘умножением’ и ‘делением’, ‘сдвиг влево’ становится ‘квадратом’, а ‘сдвиг вправо’ – ‘квадратным корнем’, ‘деление на 3’ дает ‘кубический корень’. При обратном преобразовании из экспоненты в число, получим соответствующий результат. Этот процесс хоть и несовершенный, но довольно быстрый.

Для приложений, в которых должно выполняться большое количество умножений и делений без наличия большого количества сложений и вычитаний, экспоненциальное представление может существенно ускорить процесс вычислений. Кроме того, экспоненциальное представление полезно для сжатия чисел в меньшее количество бит – жертвование разрешением на более высоких амплитудах. Во многих приложениях, таких как синтез аудио, природа сигналов – логарифмическая как по частоте, так и по амплитуде. Обработка таких данных в экспоненциальной форме вполне естественна и эффективна, что приводит к ‘линейной’ простоте при реальной логарифмичности.

В приведенных ниже примерах кода из каждой таблицы используются непосредственно сами точки. Более высокое разрешение можно было бы получить, используя линейную интерполяцию между точками, поскольку нелинейность между соседними точками таблицы очень мала. Однако ценой этому будет больший объем кода и более медленное выполнение.

Таблица Log ($C000-$CFFF)

Таблица логарифмов содержит данные, используемые для преобразования беззнаковых чисел в экспоненты по основанию 2.

Таблица логарифмов состоит из 2048 беззнаковых слов, представляющих дробные части экспоненты чисел по основанию 2. Для использования этой таблицы Вы сначала должны определить целую часть экспоненты преобразуемого числа. Это попросту позиция лидирующего бита. Для числа $60000000 она равна 30 ($1E). Эта целая часть всегда умещается в 5 бит. Отделите эти 5 бит в результате так, чтобы они занимали позиции битов 20..16. В нашем случае с числом $60000000, предварительный результат будет равен $001E0000. Далее следует выровнять сверху и отделить первые 11 бит, следующих за лидирующим битом в позициях 11..1. В нашем примере это будет $0800. Прибавив $C000 к базе таблицы логарифмов мы получаем адрес дробной части экспоненты. Читая слово по адресу $C800, мы получаем величину $95C0. Сложение этой части с предварительным результатом дает результат $001E95C0 – это и есть

число $60000000 в экспоненциальной форме. Заметьте, что биты 20..16 образуют целую часть экспоненты, в то время как биты 15..0 образуют ее дробную часть, в которой бит 15 представляет ½, бит 14 – это ¼, и так далее, до бита 0. С полученной экспонентой теперь можно выполнять сложение, вычитание и сдвиг. Всегда убеждайтесь в том, что Ваши математические операции никогда не приводят к уменьшению экспоненты ниже ноля или увеличению выше бита 20. Иначе экспоненциальное значение может не перевестись назад в обычную форму корректно.

Ниже приведена подпрограмма, преобразующая беззнаковое число в его экспоненциальную форму представления по основанию 2, используя таблицу логарифмов:

```
' Convert number to exponent
'
' on entry: num holds 32-bit unsigned value
' on exit:  exp holds 21-bit exponent with 5 integer bits and 16 fractional bits
'
numexp          mov     exp,#0              'clear exponent

                test    num,num4        wz  'get integer portion of exponent
                muxnz   exp,exp4            'while top-justifying number
    if_z        shl     num,#16
                test    num,num3        wz
                muxnz   exp,exp3
    if_z        shl     num,#8
                test    num,num2        wz
                muxnz   exp,exp2
    if_z        shl     num,#4
                test    num,num1        wz
                muxnz   exp,exp1
    if_z        shl     num,#2
                test    num,num0        wz
                muxnz   exp,exp0
    if_z        shl     num,#1

                shr     num,#30-11          'justify sub-leading bits as word
offset
                and     num,table_mask      'isolate table offset bits
                add     num,table_log       'add log table address
                rdword  num,num             'read fractional portion of exponent
                or      exp,num             'combine fractional & integer portions

numexp_ret      ret                         '91..106 clocks
                                            '(variance due to HUB sync on RDWORD)

num4            long    $FFFF0000
```

```
num3            long    $FF000000
num2            long    $F0000000
num1            long    $C0000000
num0            long    $80000000
exp4            long    $00100000
exp3            long    $00080000
exp2            long    $00040000
exp1            long    $00020000
exp0            long    $00010000
table_mask      long    $0FFE               'table offset mask
table_log       long    $C000               'log table base

num             long    0                   'input
exp             long    0                   'output
```

Таблица Anti-Log ($D000-$DFFF)

Таблица антилогарифмов содержит данные, используемые при преобразовании экспонент по основанию 2 в беззнаковые числа.

Таблица антилогарифмов состоит из 2048 беззнаковых слов, каждое из которых представляет младшие 16 бит от 17-битной мантиссы (17-тый бит изначально не определен и должен устанавливаться отдельно). Для использования этой таблицы, сдвиньте верхние 11 бит дробной части экспоненты (биты 15..5) в биты 11..1 и изолируйте. Прибавьте $D000 к базе таблицы антилогарифмов. Прочтите слово по полученному адресу в результат – это мантисса. Далее сдвиньте мантиссу влево в биты 30..15 и установите бит 31 – оставшийся 17-й бит мантиссы. Последний шаг – это сдвиг результата вправо на 31 минус целую часть экспоненты, в биты 20..16. Теперь экспонента преобразована в беззнаковое число.

Ниже приведена подпрограмма, преобразующая экспоненту по основанию 2 в беззнаковое число, используя таблицу антилогарифмов:

```
' Convert exponent to number
'
' on entry: exp holds 21-bit exponent with 5 integer bits and 16 fraction bits
' on exit:  num holds 32-bit unsigned value
'
expnum          mov     num,exp             'get exponent into number
                shr     num,#15-11          'justify exponent fraction as word
offset
                and     num,table_mask      'isolate table offset bits
                or      num,table_antilog   'add anti-log table address
                rdword  num,num             'read mantissa word into number
                shl     num,#15             'shift mantissa into bits 30..15
                or      num,num0            'set top bit (17th bit of mantissa)
```

```
                shr     exp,#20-4           'shift exponent integer into bits 4..0
                xor     exp,#$1F            'inverse bits to get shift count
                shr     num,exp             'shift number into final position

expnum_ret      ret                         '47..62 clocks
                                            '(variance is due to HUB sync on
RDWORD)

num0            long    $80000000           '17th bit of the mantissa
table_mask      long    $0FFE               'table offset mask
table_antilog   long    $C000               'anti-log table base

exp             long    0                   'input
num             long    0                   'output
```

Таблица SIN ($E000-$F001)

Таблица синусов предоставляет 2049 беззнаковых 16-битных отсчетов синуса, нарастающего от 0° до 90°, включительно (разрешение 0.0439°).

Небольшая ассемблерная подпрограмма может зеркально отобразить и перевернуть отсчеты таблицы синусов, чтобы получить полный период функции синуса/косинуса с угловым разрешением в 13 бит и точностью каждого отсчета 17 бит:

```
' Get sine/cosine
'
'      quadrant:  1             2             3             4
'         angle:  $0000..$07FF  $0800..$0FFF  $1000..$17FF  $1800..$1FFF
'   table index:  $0000..$07FF  $0800..$0001  $0000..$07FF  $0800..$0001
'        mirror:  +offset       -offset       +offset       -offset
'          flip:  +sample       +sample       -sample       -sample
'
' on entry: sin[12..0] holds angle (0° to just under 360°)
' on exit:  sin holds signed value ranging from $0000FFFF ('1') to
' $FFFF0001 ('-1')
'
getcos          add     sin,sin_90          'for cosine, add 90°
getsin          test    sin,sin_90      wc  'get quadrant 2|4 into c
                test    sin_sin_180     wz  'get quadrant 3|4 into nz
                negc    sin,sin             'if quadrant 2|4, negate offset
                or      sin,sin_table       'or in sin table address >> 1
                shl     sin,#1              'shift left to get final word address
                rdword  sin,sin             'read word sample from $E000 to $F000
                negnz   sin,sin                 'if quadrant 3|4, negate sample
getsin_ret
getcos_ret      ret                         '39..54 clocks
```

```
                                                '(variance due to HUB sync on RDWORD)


sin_90          long    $0800
sin_180         long    $1000
sin_table       long    $E000 >> 1              'sine table base shifted right

sin             long    0
```

Как и с таблицами логарифмов и антилогарифмов, для увеличения точности можно использовать линейную интерполяцию между соседними отсчетами таблицы синусов.

Индекс

_

_CLKFREQ, 153, 154
_CLKFREQ (spin), 206–7
_CLKMODE, 153
_CLKMODE (spin), 209–12
_FREE (spin), 255
_STACK (spin), 355
_XINFREQ, 153, 154
_XINFREQ (spin), 392–93

A

Abort
  Ловушка Trap, 189
ABORT (spin), 187–91
ABS (asm), 416
ABSNEG (asm), 417
ADD (asm), 417–18
ADDABS (asm), 418–19
Addressing, optimized, 337, 366
ADDS (asm), 420
ADDSX (asm), 421–22, 421–22
ADDX (asm), 423–24
AND (asm), 425
AND (spin), 259
ANDN (asm), 426
Assembly language
  JMPRET, 459–62, 459–62, 459–62, 459–62, 459–62, 459–62, 459–62, 459–62
  Local label indicator, :, 522
  OR, 488
  TEST, 523–24

B

BOEn (пин), 19
Brown Out Enable (пин), 19
Byte
  Тип памяти, 20, 192
  Чтение/запись, 495, 534
BYTE (spin), 192–97
BYTEFILL (spin), 198
BYTEMOVE (spin), 199

C

CALL (asm), 427–29
CASE (spin), 200–202
Case statement separator, :, 522
CHIPVER (spin), 203
CLKFREQ (spin), 158, 204–5
CLKMODE (spin), 208
CLKSELx (таблица), 35
CLKSET (asm), 429–30
CLKSET (spin), 158, 213–14
CMP (asm), 431
CMPSUB (asm), 433–34
CMPX (asm), 438–40
CNT, 28, 113, 353
CNT (asm), 499–500
CNT (spin), 215–16, 215–16
Cog
  ID, 217, 441
  Карта ОЗУ (рисунок), 28
  ОЗУ, 27
  Определение, 26, 115
COGID (asm), 441–42
COGID (spin), 217
COGINIT (asm), 442–44
COGINIT (spin), 218–19
COGNEW (spin), 221–26
COGSTOP (asm), 444–45
COGSTOP (spin), 227
CON (spin), 228–34
CONSTANT (spin), 235–36
Crystal Input (пин), 19
Crystal Output (пин), 19
CTRA, CTRB, 28, 353
CTRA, CTRB (asm), 499–500
CTRA, CTRB (spin), 239–42, 239–42, 239–42, 239–42

D

Delay
  Fixed (figure), 195, 277, 385
DIP, 18

DIRA, DIRB, 28, 353
DIRA, DIRB (asm), 499–500
DIRB, DIRB (spin), 249–51, 249–51, 249–51, 249–51
DJNZ (asm), 447–49

E

ELSE (spin), 259
ELSEIF (spin), 260
ELSEIFNOT (spin), 262

F

FALSE, 237
Figures
  Fixed Delay Timing, 195, 277, 385
FILE (spin), 252
FIT (asm), 453
Fixed Delay Timing (figure), 195, 277, 385
FLOAT (spin), 253–54
FROM (spin), 340
FRQA, FRQB, 28, 353
FRQA, FRQB (asm), 499–500
FRQA, FRQB (spin), 256

H

Hexadecimal indicator, $, 361
Hub (Концентратор), 25, 29
HUBOP (asm), 454
Hub-инструкции, количество тактов, 415

I

ID процессора, 217, 441
IF (spin), 257–62
IFNOT (spin), 263
INA, INB, 28, 353
INA, INB (asm), 499–500
INA, INB (spin), 263–65

J

JMP (asm), 458–59
JMPRET (asm), 459–62, 459–62, 459–62, 459–62, 459–62, 459–62, 459–62

L

List delimiter (,), 522
Local optimized addressing, 337, 366
LOCKCLR (asm), 463–64, 463–64
LOCKCLR (spin), 266–67
LOCKNEW (asm), 465
LOCKNEW (spin), 268–70
LOCKRET (asm), 466
LOCKRET (spin), 271
LOCKSET (asm), 467–68
LOCKSET (spin), 272–73
Long
  Local optimized addressing, 337, 366
  Reading/writing, 484, 532, 533, 536
  Тип памяти, 20, 274, 382
  Чтение/запись, 496, 535
  Чтение/Запись, 276, 384
LONG (spin), 274–76
LONGFILL (spin), 281
LONGMOVE (spin), 282
LOOKDOWN, LOOKDOWNZ (spin), 283–84
LOOKUP, LOOKUPZ (spin), 285–86
LQFP, 18
LSB, 308

M

MAX (asm), 468
MAXS (asm), 469
MINS (asm), 470–71
MOV (asm), 471–72
MOVD (asm), 472–73
MOVI (asm), 473–74
MOVS (asm), 474–75
MSB, 308
MUXC (asm), 475–76
MUXNC (asm), 476–77
MUXNZ (asm), 477–78
MUXZ (asm), 478–79

N

NEG (asm), 479–80
NEGC (asm), 480–81
NEGNC (asm), 481–82
NEGNZ (asm), 482

NEGX, 237, 238
NEGZ (asm), 483
NEXT (spin), 287
NOP (asm), 483–84
NR (asm), 450–51

O

OBJ (spin), 288–90
OBJ Блок, 145
Object assignment, :, 522
Optimized addressing, 337, 366
OR (asm), 488
OR (spin), 259
ORG (asm), 488–93
OSCENA (таблица), 34
OSCMx (таблица), 35
OTHER (spin), 201
OUTA, OUTB, 28, 353
OUTA, OUTB (asm), 499–500
OUTA, OUTB (spin), 326–29

P

PAR, 28, 353
PAR (asm), 499–500
PAR (spin), 330–31, 330–31, 330–31, 330–31
PHSA, PHSB, 28, 353
PHSA, PHSB (asm), 499–500
PHSA, PHSB (spin), 332
PI, 237, 238
PLL16X, 209, 237, 238
PLL1X, 209, 237, 238
PLL2X, 209, 237, 238
PLL4X, 209, 237, 238
PLL8X, 209, 237, 238
PLLENA (таблица), 34
POSX, 237, 238
PRI (spin), 333
Propeller Tool
  Организация экрана, 45
  Перемещение нескольких объектов, 50
  Перечень директорий, 48
  Поле фильтра, 48
  Пункты меню, 55–60
  Режим просмотра Весь исходный текст, 50
  Режим просмотра Документация, 50
  Режим просмотра Общий, 50
  Режим просмотра Сжатый, 50
  Режимы просмотра, 50
Propeller Ассемблер. *См. Язык Ассемблера*

Q

QFN, 18
QUIT (spin), 338

R

RC генератор, 33
RCFAST, 35, 209, 237, 238
RCL (asm), 494
RCR (asm), 494–95
RCSLOW, 35, 209, 237, 238
RDBYTE (asm), 495–96
RDLONG (asm), 496–97, 496–97, 496–97, 496–97, 496–97
RDWORD (asm), 498
Reading/writing
  Longs of main memory, 484, 532, 533, 536
  Words of main memory, 447, 448, 451, 457, 490, 491, 492, 526, 527
RES (asm), 501
Reset (пин), 19
RESET (таблица), 34
RESn (пин), 19
RESULT (spin), 347–48
RET (asm), 504
RETURN (spin), 349–50
Return value separator, :, 522
REV (asm), 504–5
ROL (asm), 505–6
ROR (asm), 506–7
ROUND (spin), 351–52

S

SAR (asm), 507–8, 507–8
SHL (asm), 509
SHR (asm), 509–10
Spin language
  CNT, 215–16
  CTRA, CTRB, 239–42, 239–42, 239–42
  DIRB, DIRB, 249–51, 249–51, 249–51

FRQA, FRQB, 256
Symbols, 360–62
VSCL, 371–72
SPR (spin), 353–54
STEP (spin), 340, 344
STRCOMP (spin), 356–57
STRING (spin), 164, 358
STRSIZE (spin), 359
SUB (asm), 510–11
SUBABS (asm), 512
SUBS (asm), 513
SUBSX (asm), 514–15
SUBX (asm), 516–17
SUMC (asm), 517–18
SUMNC (asm), 519
SUMNZ (asm), 520
SUMZ (asm), 521
Symbols
$ (Hexadecimal indicator), 361
, (list delimiter), 522
: (multipurpose), 522

T

TEST (asm), 523–24, 523–24
TJNZ (asm), 525
TJZ (asm), 526
TO (spin), 340
TRUE, 237
TRUNC, 170
TRUNC (spin), 363

U

UNTIL (spin), 341, 345

V

VAR (spin), 364–65
VCFG, 28, 353
VCFG (asm), 499–500
VCFG (spin), 368–70, 368–70
Version number, 203
VSCL, 28, 353
VSCL (asm), 499–500
VSCL (spin), 371–72, 371–72

W

WAITCNT (asm), 528–29
WAITCNT (spin), 373–76
WAITPEQ (asm), 529–30
WAITPEQ (spin), 377–78, 377–78
WAITPNE (asm), 530–31
WAITPNE (spin), 379
WAITVID (asm), 531–32
WAITVID (spin), 380–81
WC (asm), 450–51
WHILE (spin), 341, 345
Word
Reading/writing, 447, 448, 451, 457, 490, 491, 492, 526, 527
Тип памяти, 20
Чтение/запись, 498, 535
WORD (spin), 382–88, 382–88, 382–88
WORDFILL (spin), 390
WORDMOVE (spin), 391
WR (asm), 450–51
WRBYTE (asm), 534
WRLONG (asm), 534–35, 534–35, 534–35, 534–35, 534–35
WRWORD (asm), 535–36
WZ (asm), 450–51

X

XI (pin), 19
XINPUT, 35, 209, 237, 238
XO (пин), 19
XOR (asm), 537–38
XTAL1, 35, 209, 237, 238
XTAL2, 35, 209, 237, 238
XTAL3, 35, 209, 237, 238

A

Абсолютное значение ‘||’, 305
Адрес ‘@’, 324
Адрес идентификатора ‘@’, 324
Адрес Объекта Плюс Идентификатора ‘@@’, 325
Адрес Плюс Идентификатор ‘@@’, 325
Арифметический Сдвиг Вправо ‘~>’, ‘~>=’, 307
Архитектура, 24

Атрибуты операторов, 291

Б

Базы, численные, 183
Библиотечная папка, 64
Библиотечные Объекты, 161
Бинарные операторы (asm), 406
Бинарные операции (spin), 181
Биты защиты, 35, 268, 466
Биты защиты, правила, 269
Блок
  CON, 117, 228
  DAT, 118, 243
  OBJ, 117, 145, 288
  PRI, 118, 333
  private-метода, 118, 333
  PUB, 118, 334
  public-метода, 118, 334
  VAR, 117, 364
  глобальных констант, 117
  глобальных переменных, 117
  данных, 118, 243
  констант, 117, 228
  объекта, 288
  переменных, 117, 364
  ссылок на объекты, 117
Блок схема (рисунок), 24
Больше или Равно, логическое '=>', '=>=', 323
Больше, логическое '>', '>=', 322

В

Величина возврата, 335
Верхний Объектный Файл, 104, 132, 133
Верхний объектный файл, установка (рисунок), 134
Вещественные числа, 167
Взаимодействие Cog и Hub (рисунок), 29, 30
Взаимодействие Cog-Hub, 25
Вид в Таблице символов
  карта ПЗУ, 70
  символьный, 70
  стандартный, 70
Вид объекта, 131
Вид Объекта (рисунок), 64
Видео конфигурация, 28
Видео масштаб, 28
Включение питания, 22
Внешние соединения, 21, 107
Внешние соединения (рисунок), 21
Внутренний RC-генератор (техн.хар), 20
Воздействия (asm), 402, 450–51
Воздействия, Ассемблер (таблица), 450
Временная диаграмма
  для синхронизированной задержки (рисунок), 376
  для фиксированной задержки (рисунок), 375
Время жизни объекта, 151
Время, вычисление, 376
Встроенный браузер (рисунок), 47
Выдаваемый/потребляемый ток (техн.хар), 20
Выделение и перемещение блока, 83–84
Выделение и перемещение блока (рисунок), 84
Вызов метода, 124
Выключение, 23
Выполнение в отдельном Cog (рисунок), 115
Выравнивание
  Величины, 244
  Текст, 80
Выход из метода, 337
Вычисление времени, 376
Вычитание '-', '-=', 299

Г

Гарантия, 2
Генератор
  Временные соотношения, 155
  Диапазон частот, 35
  Константы установки режимов (таблица), 209, 210
  Режим работы, 208
  Установки, 153
  ФАПЧ, 26, 33

Д

Два процессора, выполняющих приложение (рисунок), 127
Декремент, пре- или пост- '- -', 299
Деление '/', '/=', 302
Демонстрационная плата Propeller Demo Board, 107

Дешифровать, побитовое ‘|<’, 309
Диалог связи с ИМС Propeller, 110
Директивы (asm), 400
Директивы (spin), 179
Доступ к Основной Памяти (asm), 402

Е

Емкость XIN/XOUT, 35

З

Загрузка, 22
Загрузка (рисунок), 106
Задержка
  синхронизированная, 374
  синхронизированная (рисунок), 376
  фиксированная, 215, 373
  фиксированная (рисунок), 375
Запуск нового процессора, 218, 221, 443
Знакогенератор, 37
Значения выходов P31 - P0, 28

И

И, логическое ‘AND’, ‘AND=’, 317
И, побитовое ‘&’, ‘&=’, 314
Идентификаторы:, 244, 288
Иерархия Объекта (рисунок), 104
ИЛИ, логическое ‘OR’, ‘OR=’, 318
ИЛИ, побитовое ‘|’, ‘|=’, 315
ИМС Propeller, 21
  Cogs (процессоры), 26
  Архитектура, 24
  Блок схема (рисунок), 24
  Включение питания, 22
  Внешние соединения, 21, 107
  Выключение, 23
  Гарантия, 2
  Загрузка, 22
  Исполнение приложения, 22
  Описание выводов, 19
  Разделяемые ресурсы, 26
  Расположение выводов, 18
  Технические характеристики, 20
  Типы корпусов, 18
  ЭППЗУ, 21
  Ядра (Cogs), 26
Индикаторы Блок-Групп, 89
Индикаторы Блок-Групп (рисунок), 89
Инкремент, пре- или пост- ‘+ +’, 300
Инструкции Ассемблера Propeller (таблица), 413–15
Интерпретатор Spin, 40, 116
Интерфейс с объектом, 130
Информации об объекте, 149
Информация компиляции, 171
Информация о компиляции (рисунок), 171
Информация об объекте (рисунок), 67, 69, 150
ИСКЛЮЧАЮЩЕЕ-ИЛИ, побитовое ‘^’, ‘^=’, 316
Исполнение приложения, 22

К

Карта Основной Памяти (рисунок), 36
Квадратный Корень ‘^^’, 304
Кнопка
  Загрузить ОЗУ, 69
  Загрузить ЭППЗУ, 69
  Открыть файл, 69
  Сохранить бинарный файл, 69
  Сохранить файл ЭППЗУ, 69
Кодировка
  ANSI, 44
  Unicode, 44
Коллизии доступа, 268
Комбинирование условий, 259
Комментарии, 118
  документации, 119
  исходного кода, 44, 119
  нескольких строк документации, {{ }}, 119
  нескольких строк кода, { }, 119
  одиночной строки документации, ' ', 119
  одиночной строки кода, ', 119
Конечные Циклы, 122
Константы, 116
Константы (предопределенные), 237–38
Контекстно-зависимая информация компиляции, 171
Конфигурация (asm), 400
Конфигурация (spin), 176

Л

Линии B/B, 30
Линии B/B (техн.хар), 20
Локальные переменные, 336

М

Максимум, ограничение ‘<#’, ‘<#=’, 304
Математич./логич. операторы в выражениях с константами (таблица), 487
Математические/логические операторы (таблица), 292
Меньше или Равно, логическое ‘=<’, ‘=<=’, 322
Меньше, логическое ‘<’, ‘<=’, 321
Меню
  Архивировать (Archive), 55
  Вставить (Paste), 57
  Выбрать верхний объектный файл… (Select Top Object File…), 55
  Выделить все (Select All), 57
  Вырезать (Cut), 57
  выход (Exit), 56
  Загрузить ОЗУ (Load RAM), 58, 59
  Загрузить ЭППППЗУ (Load EEPROM), 58, 59
  Закрыть (Close), 55
  Закрыть все (Close All), 55
  Заменить (Replace), 57
  Компилировать верхний (Compile Top), 58, 132
  Компилировать текущий (Compile Current), 58, 132
  Копировать (Copy), 57
  Найти следующий (Find Next), 57
  Найти/Заменить… (Find/Replace…), 57
  Настройки… (Preferences…), 57
  Новый (New), 55
  Обновить статус (Update Status), 58, 59
  Откат (Undo), 56
  Открыть (Open...), 55
  Открыть из… (Open From…), 55
  Перейти на отметку (Go To Bookmark), 57
  Печать… (Print…), 56
  Повтор (Redo), 56
  Показать/Скрыть браузер (Hide/Show Explorer), 56
  Просмотр Печати… (Print Preview…), 56
  Просмотр таблицы символов… (View Character Chart…), 59
  Смотреть информацию… (View Info…), 58
  Сохранить (Save), 55
  Сохранить в… (Save To…), 55
  Сохранить все (Save All), 55
  Сохранить как... (Save As…), 55
  Увеличить текст (Text Bigger), 57
  Уменьшить текст (Text Smaller), 57
Меню компиляции (рисунок), 133
Метод
  Init, 136
  Start, 136
  Stop, 136
Минимум, ограничение ‘#>’, ‘#>=’, 303

Н

Наиболее значимый бит (MSB), 308
Наименее значимый бит (LSB), 308
Найти/Заменить (рисунок), 61
Начальная загрузка, 111
Не Равно, логическое ‘<>’, ‘<>=’, 321
НЕ, логическое ‘NOT’, 319
НЕ, побитовое ‘!’, 317
Номера строк, 74, 78
Номера строк (рисунок), 78

О

Область видимости констант, 234
Область видимости объектных идентификаторов, 290
Область видимости переменных, 366
Область стека, 126
Общие элементы синтаксиса (asm), 408
Объект
  информация, 149
  ссылка, 104
  структура, 174
Объект Propeller (рисунок), 103
Объекты, 102
Объекты Propeller, 102
Объекты и приложения, 103
Объекты и Процессоры, 136
Объявление Данных, 194, 244, 276, 362, 384, 387
Ограничение по максимуму ‘<#’, ‘<#=’, 304

Ограничение по минимуму ‘#>’, ‘#>=’, 141, 303
Одновременное выполнение, 125
Ожидание переходов (фронтов импульсов), 378
ОЗУ
  Cog (техн.хар), 20
  Основное, 37
ОЗУ Cog (техн.хар), 20
Окно доступа к концентратору, 29
Операторы
  - - (Декремент, пре- или пост-), 299
  - (Отрицание), 299
  ! (Побитовое НЕ), 317
  #>, #>= (Ограничение по минимуму), 303
  &, &= (Побитовое И), 314
  **, **= (Умножение, Вернуть старшее), 302
  *, *= (Умножение, Вернуть младшее), 301
  -, -= (Вычитание), 299
  /, /= (Деление), 302
  //, //= (Остаток от деления Mod), 303
  := (Присвоение констант), 297
  ? (Случайное), 308
  @ (Адрес идентификатора), 324
  @@ (Адрес Объекта Плюс Идентификатора), 325
  ^, ^= (Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ), 316
  ^^ (Квадратный Корень), 304
  |, |= (Побитовое ИЛИ), 315
  || (Абсолютное значение), 305
  |< (Побитовое Дешифровать), 309
  ~ (Распространение Знака 7 или Пост-Очистка), 305
  ~~ (Распространение Знака 15 или Пост-Установка), 306
  ~>, ~>= (Арифметический Сдвиг Вправо), 307
  + (Положительное), 298
  + + (Инкремент, пре- или пост-), 300
  +, += (Сложение), 298
  <#, <#= (Ограничение по максимуму), 304
  <, <= (Логическое Меньше), 321
  <-, <-= (Побитовое Циклический Сдвиг Влево), 312
  <<, <<= (Побитовое Сдвиг влево), 310
  <>, <>= (Логическое Не Равно), 321
  = (Присвоение констант), 296
  =<, =<= (Логическое Меньше или Равно), 322
  ==, === (Логическое Равенство), 320
  =>, =>= (Логическое Больше или Равно), 323
  >, >= (Логическое Больше), 322
  ->, ->= (Побитовое Циклический Сдвиг Вправо), 312
  >| (Побитовое Шифровать), 310
  ><, ><= (Побитовое Реверс), 313
  >>, >>= (Побитовое Сдвиг вправо), 311
  AND, AND= (Логическое И), 317
  NOT (Логическое НЕ), 319
  OR, OR= (Логическое ИЛИ), 318
  Логическое Больше - ‘>’, ‘>=’, 322
  Логическое Больше или Равно - ‘=>’, ‘=>=’, 323
  Логическое И - ‘AND’, ‘AND=’, 317
  Логическое ИЛИ - ‘OR’, ‘OR=’, 318
  Логическое Меньше - ‘<’, ‘<=’, 321
  Логическое Меньше или Равно - ‘=<’, ‘=<=’, 322
  Логическое НЕ - ‘NOT’, 319
  Логическое Не Равно - ‘<>’, ‘<>=’, 321
  Логическое Равенство - ‘==’, ‘===’, 320
  Побитовое Дешифровать ‘|<’, 309
  Побитовое И ‘&’, ‘&=’, 314
  Побитовое И, таблица истинности (таблица), 314
  Побитовое ИЛИ ‘|’, ‘|=’, 315
  Побитовое ИЛИ, таблица истинности (таблица), 315
  Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ ‘^’, ‘^=’, 316
  Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ, таблица истинности (таблица), 316
  Побитовое НЕ (NOT) ‘!’, 113, 317
  Побитовое НЕ ‘!’, 317
  Побитовое НЕ, таблица истинности (таблица), 317
  Побитовое Реверс ‘><’, ‘><=’, 313
  Побитовое Сдвиг влево ‘<<’, ‘<<=’, 310
  Побитовое Сдвиг вправо ‘>>’, ‘>>=’, 311
  Побитовое Циклический Сдвиг Влево ‘<-’, ‘<-=’, 312
  Побитовое Циклический Сдвиг Вправо ‘->’, ‘->=’, 312
  Побитовое шифровать ‘>|’, 310
Операторы (asm), 486–87
Операторы (spin), 291–325
Описание выводов, 19
Определения синтаксиса (asm), 408

Определить аппаратуру… (Identify Hardware…), 59
Организация экрана, Propeller Tool (рисунок), 46
Основная память, 36
Основное ОЗУ, 37
Основное ОЗУ (техн.хар), 20
Основное ОЗУ/ПЗУ (техн.хар), 20
Основное ПЗУ, 37
Основное ПЗУ (техн.хар), 20
Остановка процессора, 227, 444
Остаток от деления Mod '//', '//=', 303
Отметки, 77
Отметки (рисунок), 77
Отрицание '-', 299
Отступы, 341
  несколько строк, 86
  одиночные строки, 85
Отступы и Выступы, 84–88
Очистка, пост '~', 305

П

Память
  Заполнение, 198, 281, 390
  Копирование, 199, 282
Папка Propeller Library, 48
Параллельное выполнение, 125, 127
Параметр загрузки, 28
Параметр загрузки, регистр, 330
Парные скобки, 139
Перемещение нескольких объектов, 50
Перечень директорий, 48
Перечень по категориям
  Язык Propeller Spin, 176
  Язык Propeller Ассемблер, 400
Перечень элементов Propeller ассемблер по категориям, 400
Перечисления, 231, 361
Перменная, обасть видимсти, 366
ПЗУ
  Вид символов карта ПЗУ, 70
  Основное, 37
Подключение для программирования, 21
Подключение объекта, 104
Поле PLLDIV (таблица), 240
Поле фильтра, 48
Положительное '+', 298
Полосатые Папки, 65
Последовательное выполнение, 125
Пост- очистка '~', 140
Пост-Декремент'- -', 299
Пост-Инкремент'+ +', 300
Пост-очистка '~', 305
Пост-установка '~~', 306
Потребление тока (техн.хар), 20
Правила Идентификаторов, 183
Правила линий B/B, 31, 250
Правила Синтаксиса (spin), 185
Пре-Декремент'- -', 141, 299
Представление величин, 183
Пре-Инкремент '+ +', 300
Приложение
  исходный режим генератора, 209
  Определение, 22, 104, 151
  Стартовая точка, 118
Приложение Propeller
  Определение, 22, 104, 151
Приложения
  Определение, 103
Приложения и объекты, 103
Пример приложения (рисунок), 131
Пример распределения данных в памяти (таблица), 244
Примеры использования линий B/B (таблица), 32
Приоритета уровень, 291, 294
Присвоение
  констант '=', 296
  переменных':=', 297
Присвоение констант '=', 296
Присвоение переменных':=', 297
Программный сброс, 34
Просмотр библиотеки Propeller (рисунок), 161
Просмотр информации об объекте, 150
Просмотр нескольких объектов (рисунок), 51
Процессор
  Регистры (таблица), 500
Процессоры (Cogs), 26, 115
Псевдо-вещественные числа, 167
Пункты меню, 55–60
Пустые Папки, 65

Р

Рабочая папка, 64, 164
Рабочие и Библиотечные папки, 164
Рабочие и Библиотечные папки (рисунок), 165
Равенство, логическое ‘==’, ‘===’, 320
Разделяемые ресурсы, 26
Разделяемые Ресурсы
  Взаимоисключающие, 26
  Общие, 26
Разрядная сетка, 291
Расположение выводов, 18
Расположение нескольких объектов (рисунок), 53
Распространение Знака 15 ‘~~’, 306
Распространение Знака 7 или Пост-Очистка ‘~’, 305
Реверс, побитовое ‘><’, ‘><=’, 313
Регистр
  Конфигурации видео, 368
  Масштаба видео, 371
  Параметра загрузки процессора, 330
  ФАПЧ (PLL), 332
  Частоты, 256
Регистр VCFG (таблица), 368
Регистр VSCL (таблица), 371
Регистры, 499–500
Регистры CTRA и CTRB (таблица), 240
Регистры процессора (таблица), 500
Регистры специальных функций, 28
Регистры Специальных Функций (таблица), 353
Режим редактирования
  Замена, 79
Режим редактирования
  Вставка, 79
  Выравнивание, 79
Режимы просмотра, 50, 74
Режимы просмотра (рисунок), 76
Режимы редактирования, 79–83
Режимы редактирования (рисунок), 79
Резервирование памяти, 255, 501
Рисунки
  Верхний объектный файл, установка, 134
  Вид Объекта, 64
  Внешние соединения, 21
  Временная диаграмма для синхронизированной задержки, 376
  Временная диаграмма для фиксированной задержки, 375
  Встроенный браузер, 47
  Выделение и перемещение блока, 84
  Выполнение в отдельном Cog, 115
  Два процессора, выполняющих приложение, 127
  Загрузка, 106
  Иерархия Объекта, 104
  Индикаторы Блок-Групп, 89
  Информация о компиляции, 171
  Информация об объекте, 67, 69, 150
  Карта Основной Памяти, 36
  Меню компиляции, 133
  Найти/Заменить, 61
  Номера строк, 78
  Объект Propeller, 103
  Организация экрана, Propeller Tool, 46
  Отметки, 77
  Пример приложения, 131
  Просмотр библиотеки Propeller, 161
  Просмотр нескольких объектов, 51
  Рабочие и Библиотечные папки, 165
  Расположение нескольких объектов, 53
  Режимы просмотра, 76
  Режимы редактирования, 79
  Символы шрифта Propeller, 38
  Совпадение скобок, 140
  Строка статуса, 53
  Схема для макетирования на ИМС Propeller, 107
  Таблица символов, 73
  Файлы объекта, 104
  Форматирование блока кода, 87
  Чередование символов, 39

С

Сброс, 22
Сброс, программный, 34
Свободное пространство памяти, 255
Связь с хостом, 22
Связь с ЭППППЗУ, 22
Сдвиг влево, побитовое ‘<<’, ‘<<=’, 310
Сдвиг вправо, побитовое ‘>>’, ‘>>=’, 311
Сдвиговый регистр с линейной ОС, 308
Семафор, 35, 268, 466

Семафоры, правила, 269
Символы
- - (Декремент, пре- или пост-), 299
- (Отрицание), 299
! (Побитовое НЕ), 317
#>, #>= (Ограничение по минимуму), 303
&, &= (Побитовое И), 314
**, **= (Умножение, Вернуть старшее), 302
*, *= (Умножение, Вернуть младшее), 301
-, -= (Вычитание), 299
/, /= (Деление), 302
//, //= (Остаток от деления Mod), 303
:= (Присвоение констант), 297
? (Случайное), 308
@ (Адрес идентификатора), 324
@@ (Адрес Объекта Плюс Идентификатора), 325
^, ^= (Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ), 316
^^ (Квадратный Корень), 304
|, |= (Побитовое ИЛИ), 315
|| (Абсолютное значение), 305
|< (Побитовое Дешифровать), 309
~ (Распространение Знака 7 или Пост-Очистка), 305
~~ (Распространение Знака 15 или Пост-Установка), 306
~>, ~>= (Арифметический Сдвиг Вправо), 307
+ (Положительное), 298
+ + (Инкремент, пре- или пост-), 300
+, += (Add), 298
<#, <#= (Ограничение по максимуму), 304
<, <= (Логическое Меньше), 321
<-, <-= (Побитовое Циклический Сдвиг Влево), 312
<<, <<= (Побитовое Сдвиг влево), 310
<>, <>= (Логическое Не Равно), 321
= (Присвоение констант), 296
=<, =<= (Логическое Меньше или Равно), 322
==, === (Логическое Равенство), 320
=>, =>= (Логическое Больше или Равно), 323
>, >= (Логическое Больше), 322
->, ->= (Побитовое Циклический Сдвиг Вправо), 312
>| (Побитовое Шифровать), 310
><, ><= (Побитовое Реверс), 313
>>, >>= (Побитовое Сдвиг вправо), 311
AND, AND= (Логическое И), 317
NOT (Логическое НЕ), 319
OR, OR= (Логическое ИЛИ), 318
Знакогенератор, 37
Таблица (рисунок), 73
чередование, 38
Чередование (рисунок), 39
Символы (spin), 360–62
Символы (таблица), 360–62, 360–62
Символы шрифта Propeller (рисунок), 38
Синхронизированные задержки, 374
Системный генератор, 26
Системный генератор (техн.хар), 20
Системный Счетчик, 28
Сложение ‘+’, ‘+=’, 298
Случайное ‘?’, 308
Совпадающие скобки (рисунок), 140
Сопротивление XOUT, 35
Составное выражение, 138
Состоян. входов P63- P32, 28
Состояние входов P31 - P0, 28
Сочетания клавиш, 90–99
Перечень по клавишам, 95–99
Перечень по функциям, 90–94
Список параметров, 123
Список служебных слов (таблица), 539
Ссылка на объект, 104, 288
Ссылка Объект-Константа, #, 164
Ссылка объект-метод, ., 131
Старт нового процессора, 218, 221, 443
Стек вызовов, 187, 349
Строка статуса, 171
Строка статуса (рисунок), 53
Строки с ноль-терминаторами, 357, 359
Строки, размер, 359
Строки, сравнение, 356
Структура ассемблера Propeller, 394
Структура Объектов Propeller, 174
Структура Регистра CLK (таблица), 33
Схема для макетирования на ИМС Propeller (рисунок), 107
Счетчик
Регистры, 239
Управление, 28
ФАПЧ, 28
частота, 28
Счетчик, применение
Аналог-цифр. преобразов. (АЦП), 239

Измерение duty-cycle, 239
Измерение состояния неск. линий, 239
Измерение частоты, 239
Измерение ширины импульсов, 239
Подсчет импульсов, 239
Синтез частоты, 239
Цифро-аналог. преобразов. (ЦАП), 239

T

Таблица Anti-Log, 542, 544
Таблица Log, 542
Таблица Sin, 40
Таблица Знакогенератора, 37
Таблицы
Воздействия, Ассемблер, 450
Инструкции Ассемблера Propeller, 413–15
Истинности Побитовое И, 314
Истинности Побитовое ИЛИ, 315
Истинности Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ, 316
Истинности Побитовое НЕ, 317
Константы установки режимов генератора, 209, 210
Математические/логические операторы, 292
Описание выводов, 19
Поле PLLDIV, 240
Пример распределения данных в памяти, 244
Примеры использования линий B/B, 32
Регистр VCFG, 368
Регистр VSCL, 371
Регистры CTRA и CTRB, 240
Регистры процессора, 500
Регистры специальных функций, 28
Регистры Специальных Функций, 353
Символы, 360–62, 360–62
Служебные слова, 539
Сочетания клавиш - перечень по клавишам, 95–99
Структура Регистра CLK, 33
Технические характеристики, 20
Уровни приоритетов операторов, 293
Условия, Ассемблер, 456
Таблицы Log и Anti-Log, 40
Таблицы данных, 194, 244, 276, 283, 285, 362, 384, 387
Таблицы истинности
Побитовое И, 314
Побитовое ИЛИ, 315
Побитовое ИСКЛЮЧАЮЩЕЕ-ИЛИ, 316
Побитовое НЕ, 317
Таблицы математических функций, 540
Технические характеристики, 20
Тип памяти
Byte, 20, 192
Long, 20, 274, 382
Word, 20
Тип переменной
Byte, 20, 192
Long, 20, 274, 382
Word, 20
Типы корпусов, 18

У

Указатели блоков, 117, 176
Указатель адреса в процессоре, 488
Умножение, Вернуть младшее ‘*’, ‘*=’, 301
Умножение, Вернуть старшее ‘**’, ‘**=’, 302
Унарные операторы (asm), 405
Унарные операции (spin), 180
Унарный / Бинарный, 293
Управление потоками (spin), 177
Управление потоком (asm), 402
Управление процессами (asm), 400
Управление Процессом (spin), 177
Управление Процессорами (spin), 177
Уровень приоритета, 291, 294
Уровни приоритетов операторов (таблица), 293
Условия (asm), 446
Условия, Ассемблер (таблица), 456
Условные операторы (asm), 400
Установка, пост ‘~~’, 306

Ф

Файлы объекта (рисунок), 104
ФАПЧ (PLL), 332
Фиксированная задержка, 215
Фиксированные задержки, 373
Фонт Parallax, 70
Формат файла, 43
Форматирование блока кода (рисунок), 87

Ц

Целые и вещественные числа, 167
Циклический Сдвиг Влево, побитовое ‘<-’, ‘<-=’, 312
Циклический Сдвиг Вправо, побитовое ‘->’, ‘->=’, 312
Циклы
примеры, 123, 147

Ч

Частота внешнего резонатора, 392
Численные базы, 183
Чтение/запись
Byte основной памяти, 495, 534
Long основной памяти, 276, 384, 496, 535
Word основной памяти, 498, 535

Ш

Широтно-имп. модуляция (ШИМ), 239
Шифровать, побитовое ‘>|’, 310

Э

ЭППППЗУ, 21

Я

Ядра (Cogs), 26
Язык Spin, 173
_CLKFREQ, 206
_CLKMODE, 209–12
_FREE, 255
_STACK, 355
_XINFREQ, 392–93
ABORT, 187–91
AND, 259
BYTE, 192–97
BYTEFILL, 198
BYTEMOVE, 199
CASE, 200–202
CHIPVER, 203
CLKFREQ, 204–5
CLKMODE, 208
CLKSET, 213–14
CNT, 215–16
COGID, 217
COGINIT, 218–19
COGNEW, 221–26
CON, 228–34
CONSTANT, 235–36
CTRA, CTRB, 239–42
DIRB, DIRB, 249–51
ELSE, 259
ELSEIF, 260
ELSEIFNOT, 262
FILE, 252
FLOAT, 253–54
FROM, 340
FRQA, FRQB, 256
IF, 257–62
IFNOT, 263
INA, INB, 263–65
LOCKCLR, 266–67
LOCKNEW, 268–70
LOCKRET, 271
LOCKSET, 272–73
LONG, 274–76
LONGFILL, 281
LONGMOVE, 282
LOOKDOWN, LOOKDOWNZ, 283–84
LOOKUP, LOOKUPZ, 285–86
NEXT, 287
OBJ, 288–90
OR, 259
OUTA, OUTB, 326–29
PHSA, PHSB, 332
PRI, 333
QUIT, 338
RESULT, 347–48
RETURN, 349–50
ROUND, 351–52
SPR, 353–54
STEP, 340, 344
STRCOMP, 356–57
STRING, 358
STRSIZE, 359
TO, 340
TRUNC, 363
UNTIL, 341, 345
VAR, 364–65
VSCL, 371–72
WAITCNT, 373–76

WAITPEQ, 377–78, 377–78
WAITPNE, 379
WAITVID, 380–81
WHILE, 341, 345
WORD, 382–88, 382–88, 382–88
WORDFILL, 390
WORDMOVE, 391
Бинарные операции, 181
Директивы, 179
Константы (предопределенные), 237–38
Конфигурация, 176
Операторы, 291–325
Память, 178
Правила Синтаксиса, 185
Символы, 360–62
Указатели блоков, 176
Унарные операции, 180
Управление потоками, 177
Управление Процессом, 177
Управление Процессорами, 177
Язык Spin,перечень элементов по категориям, 176
Язык Ассемблера, 394
ABS, 416
ABSNEG, 417
ADD, 417–18
ADDABS, 418–19
ADDS, 420
ADDSX, 421–22, 421–22
ADDX, 423–24
AND, 425
ANDN, 426
CALL, 427–29
CLKSET, 429–30
CMP, 431
CMPSUB, 433–34
CMPX, 438–40
CNT, 499–500
COGID, 441–42
COGINIT, 442–44
COGSTOP, 444–45
CTRA, CTRB, 499–500
DIRA, DIRB, 499–500
DJNZ, 447–49
FIT, 453
FRQA, FRQB, 499–500
HUBOP, 454
INA, INB, 499–500
JMP, 458–59
LOCKCLR, 463–64, 463–64
LOCKNEW, 465
LOCKRET, 466
LOCKSET, 467–68
MAX, 468
MAXS, 469
MINS, 470–71
MOV, 471–72
MOVD, 472–73
MOVI, 473–74
MOVS, 474–75
MUXC, 475–76
MUXNC, 476–77
MUXNZ, 477–78
MUXZ, 478–79
NEG, 479–80
NEGC, 480–81
NEGNC, 481–82
NEGNZ, 482
NEGZ, 483
NOP, 483–84
NR, 450–51
OR, 488
ORG, 488–93
OUTA, OUTB, 499–500
PAR, 499–500
PHSA, PHSB, 499–500
RCL, 494
RCR, 494–95
RDBYTE, 495–96
RDLONG, 496–97, 496–97, 496–97, 496–97, 496–97
RDWORD, 498
RES, 501
RET, 504
REV, 504–5
ROL, 505–6
ROR, 506–7
SAR, 507–8, 507–8
SHL, 509
SHR, 509–10
SUB, 510–11
SUBABS, 512
SUBS, 513
SUBSX, 514–15
SUBX, 516–17
SUMC, 517–18

SUMNC, 519
SUMNZ, 520
SUMZ, 521
TEST, 523–24
TJNZ, 525
TJZ, 526
VCFG, 499–500
VSCL, 499–500
WAITCNT, 528–29
WAITPEQ, 529–30
WAITPNE, 530–31
WAITVID, 531–32
WC, 450–51
WR, 450–51
WRBYTE, 534
WRLONG, 534–35, 534–35, 534–35, 534–35, 534–35
WRWORD, 535–36
WZ, 450–51
XOR, 537–38
Бинарные операторы, 406
Воздействия, 402, 450–51
Воздействия (таблица), 450
Директивы, 400
Доступ к Основной Памяти, 402
Конфигурация, 400
Общие элементы синтаксиса, 408
Операторы, 486–87
Определения синтаксиса, 408
Регистры, 499–500
Сводная таблица, 413–15
Структура, 394
Унарные операторы, 405
Управление потоком, 402
Управление процессами, 400
Условия, 446
Условия (таблица), 456
Условные операторы, 400