**Propeller 2 Spin Manual**

# NOTICE: Work In Progress

This document is not complete; it is a work in progress.

The following indicators appear in different sections to describe their state:

* *{tbw}* - needs to be written
* *{rough}* - rough draft
* *{draft}* - revised; needs review/finalization
* no mark - considered official, though may be enhanced over time

#### COPYRIGHTS AND TRADEMARKS

This documentation is copyright © 2024 by Parallax Inc. By downloading or obtaining a printed copy of this documentation you agree that it is to be used with, or with products containing, the Parallax Propeller 2 P2X8C4M64P microcontroller. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use is permitted, subject to the following Conditions of Duplication: Parallax Inc. grants the user a conditional right to download, duplicate, and distribute this text without Parallax's permission. This right is based on the following conditions: the text, or any portion thereof, may not be duplicated for commercial use; it may be duplicated only for educational purposes when used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication.

Parallax, Propeller Spin, and the Parallax logos are trademarks of Parallax Inc. If you decide to use any trademarks of Parallax Inc. on your web page or in printed material, you must state that "(trademark) is a (registered) trademark of Parallax Inc." upon the first appearance of the trademark name in each printed document or web page. Other brand and product names herein are trademarks or registered trademarks of their respective holders.

#### DISCLAIMER OF LIABILITY

Parallax Inc. makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Parallax Inc. assume any liability arising out of the application or use of any product, and specifically disclaims any and all liability, including without limitation consequential or incidental damages even if Parallax Inc. has been advised of the possibility of such damages.

#### INTERNET DISCUSSION LISTS

Parallax maintains active web-based discussion forums for people interested in Parallax Propeller products, at [forums.parallax.com](https://forums.parallax.com/).

#### ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by commenting/suggesting on live documentation, or by sending an email to editor@parallax.com. We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our website, [www.parallax.com](https://www.parallax.com). Please check the individual product page’s free downloads for an errata file.

#### SUPPORTED HARDWARE AND FIRMWARE

This manual is valid with the following hardware and firmware versions:

| Hardware | Firmware |
| --- | --- |
| P2X8C4M64P | Rev B/C |

#### CREDITS

Authorship: Jeff Martin

With many thanks to everyone in the Propeller Community and staff at Parallax Inc.

# 

# TABLE OF CONTENTS

[NOTICE: Work In Progress 1](#_9udzb232cl2o)

[COPYRIGHTS AND TRADEMARKS 2](#_e7pljw5zvubg)

[DISCLAIMER OF LIABILITY 2](#_ehko7ecz5h9m)

[INTERNET DISCUSSION LISTS 2](#_cofidxyentvg)

[ERRATA 2](#_a2gz4w1zyntc)

[SUPPORTED HARDWARE AND FIRMWARE 2](#_sxilnx3rlhff)

[Hardware 2](#_c6ad30wjmjod)

[Firmware 2](#_vldzhfnfahml)

[CREDITS 2](#_wcjefvis5ns)

[TABLE OF CONTENTS 3](#_sph16lbbva3a)

[PREFACE 6](#_ogdq4x2znqss)

[2: Spin2 Language Reference 6](#_qs8sa1ouuyrp)

[Structure Of Propeller Applications 6](#_nsjoa09td118)

[Minimal Spin2 Object Example (single PUB block only) 6](#_lzlh668m61g3)

[Minimal PASM2 Object Example (single DAT block only) 6](#_q4oxja71ql9m)

[Spin2 Object Example (all blocks used) 6](#_nel3leixr96p)

[Block Designators 7](#_1b52al4ptbfl)

[Propeller 2 Object Blocks 7](#_klg1ama1nzjm)

[CON 7](#_t6aca2wdo2pu)

[VAR 8](#_1ap708yqkzj5)

[OBJ 8](#_s29j13uqmpys)

[PUB/PRI 8](#_vakzsbsoahag)

[DAT 8](#_3xir2se5o3u2)

[Comments 8](#_w16iaa6e7jpc)

[Categorical Listing Of Propeller 2 Spin Language (Spin2) 8](#_epgo7e5dz41r)

[Block Designators 8](#_7ywj2x2xcvl5)

[Configuration 8](#_bxpg2k7rfc3w)

[Cog Control 9](#_ea9gup7iozvf)

[I/O Control 9](#_buizvlv1gmty)

[Process Control 9](#_s2v6fzp25ccl)

[Flow Control 10](#_ri09opnk5hnv)

[Data / Memory 10](#_57dnp38dksb8)

[String Handling 11](#_mebxh8nyrbk3)

[Math 11](#_pm3gkfmvuksp)

[Debugging 12](#_8i943sn6dq95)

[Timing 13](#_41x4vj4rsiy2)

[In-line PASM 13](#_qigh1e2aal76)

[PASM Interfacing 13](#_j16v7sip9lwv)

[PASM 13](#_6385ed4zkuo3)

[Registers 13](#_w6n8gmrrf2ce)

[Unary Operators 14](#_i955jopi6m6p)

[Binary Operators 14](#_6r5vsgtc72i1)

[Syntax Symbols 14](#_3iwld29xtb2z)

[Spin Language Elements 15](#_ehgzucz6u4rq)

[Symbol Rules 15](#_amxd1web2xeu)

[Value Representations 15](#_3kusx48s7ip9)

[Syntax Definitions 15](#_2gav1ad5tqsa)

[Code Examples 16](#_yeod1hkl0f8y)

[ABORT 17](#_mzucodib2mnd)

[About the Call Stack 17](#_j4zgaptja4gd)

[Using ABORT 17](#_s7dfktzi2hlh)

[The Abort Trap ( \ ) 18](#_dg8ox9ta1b80)

[Example Use Of Abort 18](#_kq4tll4tsk4i)

[An Abort without Abort Trap (\) 19](#_bofnm2negry8)

[Zero or One Value Allowed 19](#_hq4zlag3jn5z)

[CASE / CASE\_FAST 20](#_67fg45p5mf4h)

[Indention is Critical 20](#_1cj8dsjcmfjr)

[Using CASE 20](#_uhe020mcqriz)

[Using OTHER 20](#_ooso0heik3fo)

[Using CASE\_FAST 21](#_egcajbper2jh)

[Code Variations 21](#_i1l64y7tgqe0)

[IF / IFNOT 23](#_ynxmxpml4tr)

[Indention is Critical 23](#_pta34qb8ebgv)

[Simple IF Statement 23](#_dfxec0o9vcuk)

[Combining Conditions 24](#_apea61ubvqis)

[Using IF with ELSE 24](#_yzgvze6pm0g3)

[Using IF with ELSEIF 24](#_8zoqn0xpahcf)

[Using IF with ELSEIF and ELSE 25](#_e3xr4qx8pbzn)

[The ELSEIFNOT Condition 26](#_lsn9f68pi971)

[REPEAT 27](#_p0qja0venbrs)

[Indention is Critical 27](#_l6o2kryf6a0p)

[Infinite Loops (Syntax 1 w/o Count) 28](#_cb621vqxkm9m)

[Simple Finite Loops (Syntax 1) 28](#_gh4lrwoszg32)

[Counted Finite Loops (Syntax 2 & 3) 28](#_qa5e50ggha2u)

[Conditional Loops (Syntax 3 and 4) 30](#_h2ea618alb8e)

[Other REPEAT Options 31](#_iykra55dk0kr)

[RETURN 32](#_p6ujo2mdsgw7)

[About the Call Stack 32](#_ttaakyjgtx6f)

[Using RETURN 32](#_3p5yg194m87x)

[Multiple Return Values 33](#_31p344imjis7)

[Propeller 2 Reserved Words (Spin2 + PASM2) 34](#_yafe38dy7cdm)

[\_ (leading underscore) 34](#_dpjv1sk2zuco)

[A - B 34](#_hd4bfchbjkj)

[C - D 34](#_1dm01l1nuslg)

[E - F 34](#_ky6mbj2ov5p4)

[G - H 34](#_y78ihk1mdgxj)

[I - J 34](#_sj8nujvlw6hl)

[L - M 34](#_44jg123saz5v)

[N - O 34](#_hl3y4yy9fsed)

[P 35](#_9povq3pnjven)

[Q - R 35](#_hhnkaf3kzx8g)

[S - T 35](#_64zabaxy953i)

[U, V, W 35](#_1pkmmgrz5cfo)

[X, Y, Z 35](#_fdbohmsf3tai)

[CHANGE LOG 37](#_swmvo3n67qaa)

[PARALLAX INCORPORATED 37](#_cenlv5sy0ptb)

# 

# 

# PREFACE

This manual is an in-depth description of the Spin 2 programming language of the Propeller 2 multicore microcontroller.

For additional documentation and resources, including programming tools, visit [www.parallax.com/P2](http://www.parallax.com/P2). The latest version of this manual, along with links to a commentable Google Doc version, are available from the Documentation section. In addition, there are links to more in-depth references for the Propeller 2 and its Spin2 and PASM2 languages, which may include commentable Google Docs.

# 2: Spin2 Language Reference

*{tbw}*

## Structure Of Propeller Applications

*{draft}*

The Propeller 2 supports programming in multiple languages. This document focuses primarily on the Parallax Spin2 language but also introduces a little of its close companion, PASM2 (Propeller 2 Assembly Language).

Spin2 applications are built from one or more objects. Objects are Spin2 text files describing executable code to perform a dedicated function. Spin2 objects often include a combination of public and private methods, constants, variables, data, and references to other objects in order to do their job.

To form an application, its objects are assembled together into a top-level executable object with an internal hierarchy of sub-objects. At run-time, each object instance gets its own set of variables, as defined by the object, to maintain its unique operating state.

The tables below show minimum and larger examples of complete object files.

| Minimal Spin2 Object Example (single PUB block only) |
| --- |
| **PUB MinimalSpin2Program()** 'first PUB method executes  REPEAT  PINWRITE(7..0, GETRND()) 'write a random pattern to P7..P0  WAITMS(100) 'wait 1/10th of a second, loop |

| Minimal PASM2 Object Example (single DAT block only) |
| --- |
| **DAT ORG** 'start PASM at hub $00000 for cog $000  loop DRVRND #0 ADDPINS 7 'write a random pattern to P7..P0  WAITX ##clkfreq\_/10 'wait 1/10th of a second, loop  JMP #loop |

| Spin2 Object Example (all blocks used) |
| --- |
| **CON**  \_clkfreq = 297\_000\_000 'set clock frequency  **OBJ**  vga : "VGA\_640x480\_text\_80x40" 'instantiate vga object  **VAR**  time, i 'declare object-wide variables  **PUB go()** 'this first public method executes, cog stops after  vga.start(8) 'start vga on base pin 8  SEND := @vga.print 'establish SEND pointer  SEND(4, $004040, 5, $00FFFF) 'set light cyan on dark cyan  time := GETCT() 'capture time  i := @text 'print file to vga screen  REPEAT @textend-i  SEND(byte[i++])  time := GETCT() - time 'capture time delta in clock cycles  time := MULDIV64(time, 1\_000\_000, clkfreq) 'get time delta in microseconds  SEND(12, "Time during printing was ", dec(time), " microseconds.") 'print time delta  **PRI dec(value) | flag, place, digit** 'private method prints decimals, three local variables  flag~ 'reset digit-printed flag  place := 1\_000\_000\_000 'start at the one-billion's place and work downward  REPEAT  IF flag ||= (digit := value / place // 10) || place == 1 'print a digit?  SEND("0" + digit) 'yes  IF LOOKDOWN(place : 1\_000\_000\_000, 1\_000\_000, 1\_000) 'also print a comma?  SEND(",") 'yes  WHILE place /= 10 'next place, done?  **DAT**  text FILE "VGA\_640x480\_text\_80x40.txt" 'include raw file data for printing  textend |

### Block Designators

Each object uses six different special-purpose blocks (CON, VAR, OBJ, PUB, PRI, and DAT) to organize its contents. An object may include few or many of each block type but must include at least one PUB block or one DAT block; public method or data block, respectively. If a PUB block is included, the object may contain both Spin2 and PASM2 source code. If a DAT block (but no PUB block) is included, the object may contain PASM2 source code or just data and constants (no code at all).

| Propeller 2 Object Blocks | | | |
| --- | --- | --- | --- |
| **Block Identifier** | **Block Contents** | **Spin2+PASM2**  **Applications** | **PASM2-only**  **Applications** |
| **CON** | Constant and data-structure declarations (CON is the initial/default block type) | Permitted | Permitted |
| **VAR** | Variable declarations | Permitted | Not Allowed |
| **OBJ** | Child-object declarations (creates compile-time object instantiations) | Permitted | Not Allowed |
| **PUB** | Public method for use by the parent object and within this object | Required | Not Allowed |
| **PRI** | Private method for use only within this object | Permitted | Not Allowed |
| **DAT** | Data declarations and/or PASM2 code | Permitted | Required |

#### CON

*{rough}*

CON blocks are used to declare symbolic constants and data structures which can be used throughout the object. There may be zero or more CON blocks in an object and the order in which they appear does not matter. Typically, objects have just one CON block, near the top of code, that defines every constant for that object; however, objects can be organized with multiple smaller CON blocks that each appear right above the code or data (PUB/PRI/DAT) that directly use those symbols.

Example CON Block:

**CON**

EnableFlow = 8 'single assignment

x = 5, y = -5, z = 1 'comma-separated assignments

HalfPi = 1.5707963268 'IEEE-754 single-precision float values

#0,a,b,c 'enum; a=0, b=1, c=2 (start=0, step=1)

#1,e,f,g 'enum; e=1, f=2, g=3 (start=1, step=1)

#4[2],i,j,k 'enum; i=4, j=6, k=8 (start=4, step=2)

sPoint(x, y) 'structure; two longs

sLine(sPoint a, sPoint b, BYTE color) 'structure; two sPoints and a byte

#### **VAR**

*{tbw}*

#### **OBJ**

*{tbw}*

#### PUB/PRI

*{tbw}*

#### **DAT**

*{tbw}*

### Comments

*{tbw}*

## Categorical Listing Of Propeller 2 Spin Language (Spin2)

In the following category lists, styles indicate different types of elements.

* **BLOCKS**, **DIRECTIVES**, and **COMMANDS**
* **METHODS**() and **ACCESSORS**[]
* CONSTANTS
* *PSEUDO-CONSTANTS*
* *variables*

### Block Designators

CON Declare constant block

VAR Declare variable block

OBJ Declare object reference block

PUB Declare public method block

PRI Declare private method block

DAT Declare data block

### Configuration

*{draft}*

*\_CLKFREQ* Set application (initial) clock frequency at compile-time

*\_CLKMODE* Set application (initial) clock mode at compile-time

*\_ERRFREQ* Set acceptable frequency variance (+/-) from \_CLKFREQ at compile-time

CLKFREQ\_ Compiled application clock frequency

CLKMODE\_ Compiled application clock mode

*clkfreq* Current clock frequency at run-time

*clkmode* Current clock mode at run-time

CLKSET() Safely set new clock frequency and mode at run-time

HUBSET() Execute system configuration instruction

*DOWNLOAD\_BAUD* Set the download baud rate

*DEBUG\_BAUD* Set debug subsystem serial baud rate

*DEBUG\_COGS* Select cogs to debug

*DEBUG\_DELAY* Delay (in milliseconds) application waits before transmitting debug messages

*DEBUG\_PIN\_TX* Select serial output pin for debug subsystem

*DEBUG\_PIN\_RX* Select serial input pin for host interaction with debug subsystem

*DEBUG\_LEFT* Set left coordinate where the Debug Message window appears on the host screen

*DEBUG\_TOP* Set top coordinate where the Debug Message window appears on the host screen

*DEBUG\_WIDTH* Set the width of the Debug Message window on the host screen

*DEBUG\_HEIGHT* Set the height of the Debug Message window on the host screen

*DEBUG\_DISPLAY\_LEFT* Set the overall left offset where any Debug displays appear on the host screen

*DEBUG\_DISPLAY\_TOP* Set the overall top offset where any Debug displays appear on the host screen

*DEBUG\_WINDOWS\_OFF* Prevent any debug windows from opening after downloading

*DEBUG\_TIMESTAMP* Enable message time-stamping

*DEBUG\_LOG\_SIZE* Set the max size (in bytes) of the Debug log file which collects debug messages

*DEBUG\_COGINIT* Invoke cogs' PASM-level debugger when a COGINIT occurs

*DEBUG\_MAIN* Same as above plus ready to single-step through main (non-interrupt) code.

### Cog Control

*{draft}*

COGSPIN() Start, or restart, a Spin2 cog by ID

COGINIT() Start, or restart, a PASM2 cog by ID

COGSTOP() Stop a cog by ID

COGID() Get current cog’s ID

COGCHK() Check if cog running, by ID

### I/O Control

*{draft}*

PINW() / PINWRITE() Drive pin(s) high and/or low

PINL() / PINLOW() Drive pin(s) low

PINH() / PINHIGH() Drive pin(s) high

PINT() / PINTOGGLE() Drive and toggle pin(s)

PINF() / PINFLOAT() Float pin(s)

PINR() / PINREAD() Read pin(s)

PINSTART() Start smart pin(s)

PINCLEAR() Clear smart pin(s)

WRPIN() Write to Mode register of smart pin(s)

WXPIN() Write to X register of smart pin(s)

WYPIN() Write to Y register of smart pin(s)

AKPIN() Acknowledge smart pin(s)

RDPIN() Read and acknowledge smart pin

RQPIN() Read, but don't acknowledge, smart pin

### Process Control

*{draft}*

LOCKNEW() Check out a new lock

LOCKRET() Release a lock

LOCKTRY() Try to capture a lock

LOCKREL() Release a lock

LOCKCHK() Check a lock's state

COGATN() Ask for a cog's attention

POLLATN() Check if any cog needs our attention

WAITATN() Wait for any cog to ask our attention

### Flow Control

[IF / IFNOT](#_ynxmxpml4tr) Conditionally execute block of code…

… [ELSE](#_ynxmxpml4tr) optionally execute another upon opposite condition,

… [ELSEIF / ELSEIFNOT](#_ynxmxpml4tr) optionally execute another upon opposite condition plus new condition

[CASE / CASE\_FAST](#_67fg45p5mf4h) Evaluate expression and execute block of code matching a condition…

… [OTHER](#_67fg45p5mf4h) optionally execute block of code when no match found

[REPEAT](#_p0qja0venbrs) Execute block of code repetitively an infinite or finite number of times with…

+ [WITH](#_p0qja0venbrs) optional loop counter,

+ [FROM… TO… STEP](#_p0qja0venbrs) optional loop counter plus intervals and steps, or…

+/… [UNTIL / WHILE](#_p0qja0venbrs) optional exit and continue conditions

… [NEXT](#_p0qja0venbrs) Skip rest of REPEAT block and jump to next loop iteration

… [QUIT](#_p0qja0venbrs) Exit from REPEAT loop

[RETURN](#_p6ujo2mdsgw7) Exit PUB/PRI with normal status and optional return value

[ABORT](#_mzucodib2mnd) Exit PUB/PRI with abort status and optional return value

### Data / Memory

*{draft}*

ALIGNW Word-align variable or data to main memory

ALIGNL Long-align variable or data to main memory

BYTE/ BYTE[] Declare byte-sized symbol, DAT data, or access byte of main memory

WORD/ WORD[] Declare word-sized symbol, DAT data, or access word of main memory

LONG/ LONG[] Declare long-sized symbol, DAT data, or access long of main memory

BYTEFIT Declare byte-sized data (in DAT) limited to range -$80 to $FF

WORDFIT Declare word-sized data (in DAT) limited to range -$8000 to $FFFF

FIELD[] Access bit field within main memory byte/word/long or from register

REG[] Access a cog register

BYTEFILL() Fill bytes of main memory with a value

WORDFILL() Fill words of main memory with a value

LONGFILL() Fill longs of main memory with a value

BYTEMOVE() Copy bytes from one region to another in main memory

WORDMOVE() Copy words from one region to another in main memory

LONGMOVE() Copy longs from one region to another in main memory

BYTESWAP() Swap two ranges of bytes in main memory

WORDSWAP() Swap two ranges of words in main memory

LONGSWAP() Swap two ranges of longs in main memory

BYTECOMP() Compare two ranges of bytes in main memory

WORDCOMP() Compare two ranges of words in main memory

LONGCOMP() Compare two ranges of longs in main memory

GETREGS() Copy cog register longs to main memory

SETREGS() Copy main memory longs to cog registers

FILL() Fill structure with byte value

COPY() Copy structure's contents to another structure

SWAP() Swap structure's contents with another structure

COMP() Compare contents of two structures

GETCRC() Compute CRC (up to 32-bits) of bytes from main memory

LOOKUP() Get value at index (1..N) from a list

LOOKUPZ() Get value at zero-based index (0..N−1) from a list

LOOKDOWN() Get index (1..N) of a matching value from a list

LOOKDOWNZ() Get zero-based index (0..N−1) of a matching value from a list

SEND() Special method pointer for efficient output of data

RECV() Special method pointer for efficient input of data

FILE Include binary file in DAT block

*varbase* Main memory base for object; used by method-pointer calls

### String Handling

*{draft}*

BYTE() Declare string of byte-sized values and return a pointer

WORD() Declare string of word-sized values and return a pointer

LONG() Declare string of long-sized values and return a pointer

STRING() Declare compile-time in-line z-string expression and return a pointer

LSTRING() Declare compile-time in-line l-string expression and return a pointer; zeros allowed

STRSIZE() Get size of z-string in bytes

STRCOPY() Copy z-string from one location to another; up to specified max

STRCOMP() Compare contents of two z-strings

@"" Declare compile-time in-line text z-string and return a pointer

### Math

*{draft}*

FLOAT() Declare compile-time floating-point expression

ROUND() Round compile-time floating-point expression to integer

TRUNC() Truncate compile-time floating-point expression at decimal

FSQRT Get square root of floating-point number

NAN() Check if floating-point value is a number

GETRND() Get random value

ROTXY() Rotate point by angle

POLXY() Convert from polar to cartesian coordinate

XYPOL() Convert from cartesian to polar coordinate

QSIN() Rotate horizontal vector by step ratio; returns y

QCOS() Rotate vertical vector by step ratio; returns x

MULDIV64() Multiply 32-bit values then divide the 64-bit product

FALSE Boolean false; same as 0 ($0000\_0000)

TRUE Boolean true; same as -1 ($FFFF\_FFFF)

NEGX Most-negative integer; -2\_147\_483\_648 ($8000\_0000)

POSX Most-positive integer; +2\_147\_483\_647 ($7FFF\_FFFF)

PI Single-precision floating-point value of Pi; 3.14159265

### Debugging

*{draft}*

*Lower-case 'x' means multiple variations exist.*

*DEBUG\_x* Configure system debug behaviors (see [Configuration](#_bxpg2k7rfc3w))

DEBUG Invoke PASM-level debugger (no parentheses)

DEBUG() Transmit debug message to host

IF() / IFNOT() Conditionally execute next Debug Command(s)

DLY() Delay (in milliseconds) *after* performing Debug's command(s)

PC\_KEY() Get the host's keyboard state

PC\_MOUSE() Get the host's mouse state

UDECx() / SDECx() / FDECx() \* Format as unsigned/signed/floating-point decimal value in debug message

UHEXx() / SHEXx() \* Format as unsigned/signed hexadecimal value in debug message

UBINx() / SBINx() \* Format as unsigned/signed binary value in debug message

ZSTRx() / LSTRx() \* Output zero-terminated/length-terminated string in debug message

BOOL() \* Format value as Boolean string ("TRUE"/"FALSE") in debug message

\* Alternate formatters exist (same name with "\_" appended) for value-only output; ex: BOOL\_, ZSTR\_, and UDEC\_.

TERM Create text terminal debug display

BITMAP Create bitmap debug display

PLOT Create general-purpose cartesian/polar plotter debug display

LOGIC Create multi-channel logic analyzer debug display

SCOPE Create multi-channel oscilloscope debug display

SCOPE\_XY Create multi-channel XY polar/log-scale oscilloscope debug display

FFT Create multi-channel Fast Fourier Transform log-scale debug display

SPECTRO Create spectrograph log-scale debug display

MIDI Create piano keyboard debug display

TITLE**,** POS**,** SIZE**,** RANGE**,** RATE**,** These display commands are used to configure graphical

SPACING**,** SAMPLES**,** ALT**,** SIGNED**,** debug displays during their creation

CHANNEL**,** DEPTH**,** MAG**,** LOGSCALE**,**

TRACE**,** DOTSIZE**,** SPARSE**,**

LINESIZE**,** TEXTSIZE**,** COLOR**,**

BACKCOLOR**,** LUTCOLORS**,** BLACK**,**

WHITE**,** ORANGE**,** BLUE**,** GREEN**,**

CYAN**,** RED**,** MAGENTA**,** YELLOW**,**

GRAY**,** GREY**,** LUTx**,** LUMA8x**,** HSVx**,**

RGBx**,** POLAR**,** CARTESIAN**,**

HIDEXY**,** LONGS\_xBIT**,** WORDS\_xBIT**,**

BYTES\_xBIT**,** UPDATE

OPACITY**,** PRECISE**,** ORIGIN**,** These display commands are used to adjust graphical

SET**,** SCROLL**,** LINESIZE**,** DOT**,** debug displays while feeding them data

LINE**,** CIRCLE**,** OVAL**,** BOX**,** OBOX**,**

COLOR**,** BACKCOLOR**,** LUTCOLORS**,**

BLACK**,** WHITE**,** ORANGE**,** BLUE**,**

GREEN**,** CYAN**,** RED**,** MAGENTA**,**

YELLOW**,** GRAY**,** GREY**,** LUTx**,** LUMA8x**,**

HSVx**,** RGBx**,** TEXTSTYLE**,**

TEXTANGLE**,** TEXT**,** SPRITEDEF**,**

SPRITE**,** TRACE**,** AUTO**,** TRIGGER**,**

HOLDOFF**,** CLEAR**,** SAVE**,** CLOSE

### Timing

*{draft}*

GETCT() Get count from 32-bit system counter

POLLCT() Check if system counter passed target

WAITCT() Wait for system counter to pass target

WAITUS() Wait for target microseconds

WAITMS() Wait for target milliseconds

GETMS() Get milliseconds since boot up

GETSEC() Get seconds since boot up

### In-line PASM

*{draft}*

ORG Start in-line PASM2 code; optionally positioned and limited

ORGF Prefill cog registers with zeros up to specified address

RES Reserve, but don't fill, one or more cog registers

FIT Limit in-line PASM2 code to desired length

END End in-line PASM2 code

### PASM Interfacing

*{draft}*

CALL() Call register or main-resident PASM2 code from Spin2

REGEXEC() Load and execute PASM2 code from register memory

REGLOAD() Load, but don't execute, PASM2 code into register memory

### PASM

ASMCLK (deprecated) Set clock mode for PASM2-only applications

### Registers

*{draft}*

PR0 Propeller register 0; general purpose Spin2/PASM2 use

PR1 Propeller register 1; general purpose Spin2/PASM2 use

PR2 Propeller register 2; general purpose Spin2/PASM2 use

PR3 Propeller register 3; general purpose Spin2/PASM2 use

PR4 Propeller register 4; general purpose Spin2/PASM2 use

PR5 Propeller register 5; general purpose Spin2/PASM2 use

PR6 Propeller register 6; general purpose Spin2/PASM2 use

PR7 Propeller register 7; general purpose Spin2/PASM2 use

IJMP3 Interrupt jump 3; for PASM2 interrupt routines

IRET3 Interrupt return 3; for PASM2 interrupt routines

IJMP2 Interrupt jump 2; for PASM2 interrupt routines

IRET2 Interrupt return 2; for PASM2 interrupt routines

IJMP1 Interrupt jump 1; for PASM2 interrupt routines

IRET1 Interrupt return 1; for PASM2 interrupt routines

PA Pointer register A; for PASM2 indirect memory access

PB Pointer register B; for PASM2 indirect memory access

PTRA Data pointer passed from COGINIT; for PASM2

PTRB Code pointer passed from COGINIT; for PASM2

DIRA Direction register for 32-bit port A (P31:P0)

DIRB Direction register for 32-bit port B (P63:P32)

OUTA Output register for 32-bit port A (P31:P0)

OUTB Output register for 32-bit port B (P63:P32)

INA Input register for 32-bit port A (P31:P0) [read only]

INB Input register for 32-bit port B (P63:P32) [read only]

### Unary Operators

*{tbw}*

### Binary Operators

*{tbw}*

### Syntax Symbols

*{tbw}*

## 

## Spin Language Elements

*{tbw}*

### Symbol Rules

*{tbw}*

### Value Representations

*{tbw}*

### Syntax Definitions

In addition to detailed descriptions, the following pages contain syntax definitions that concisely demonstrate all the options of that element. The syntax definitions use special symbols to indicate when and how certain element features are to be used.

**BOLDCAPS** Items in bold uppercase should be typed in as shown.

*Italics* Items in italics should be replaced by user text; symbols, operators, expressions, etc.

. .. : , # Periods, double-periods, colons, commas, pound signs, pipes, back slashes, square brackets

| \ [ ] ( ) and parentheses should be typed in where shown.

{ } Curly brace symbols enclose optional items. Enter the enclosed item if desired. Do not enter the curly braces.

⦅┆⦆ Hollow parentheses symbols enclose mutually exclusive items, each separated by a dash-bar. Enter one, and only one, of the encoded items. Do not enter the hollow parentheses or dash-bar.

… Repetition symbol indicates that the previous item, or group, can be repeated numerous times. Repeat the last item(s) if desired. Do not enter the repetition symbol.

⇢ Indent symbol indicates following items should be intended by at least one space.

↪ New Line+Indent symbol indicates following items should appear on a new line, indented by at least one space.

Thin line Separates various syntax structure options.

Heavy line Separates instruction from the value it returns.

Since elements are limited to specific blocks, all syntax definitions begin with an indication of the type of block required. For example, the following syntax indicates that the BYTEFILL command and its parameters must appear in either a PUB or PRI block, but it may be one of many commands within that same block.

⦅PUB┆PRI⦆

BYTEFILL (*StartAddress, Value, Count*)

### 

### Code Examples

*{draft}*

Much of this document's descriptions feature short code examples that follow these conventions:

normal code Commands, directives, etc. appear in monospaced font to represent exact code

<pseudocode> Items in angle brackets represent a concept, not literal executable code; it must be replaced with valid code before execution

**bold code** Just like normal code, but marks start of block for better readability

highlighted code Highlights emphasize the main topics to note

... Ellipsis means "more code here" where it's important to know code belongs there but specifics are irrelevant to the example

**PUB/PRI (implied)** If an example is a short snippet of Spin2 code, remember that it is implied to be within a PUB or PRI block that was left out for brevity

## ABORT

*{draft}*

**Command:** Exit with abort status from PUB/PRI method with optional return *Value*.

⦅**PUB**┆**PRI**⦆

**ABORT** {*Value*}

**Returns:** Either zero, or *Value* if provided.

* ***Value*** is an optional expression whose value is to be returned, with abort status, from the method.

**Explanation**

ABORT is one of two commands (ABORT and RETURN) that immediately terminates a PUB or PRI method.

ABORT causes the PUB or PRI method to exit with abort status— meaning it pops the call stack repeatedly from any depth of method calls until it either reaches a caller that includes an [Abort Trap](#_dg8ox9ta1b80) and delivers a value to that caller, or the call stack is emptied and the cog terminates. All callers in-between are terminated without warning.

It may be easier to write that application using small, specialized methods that are called in a nested fashion, each meant to deal with a specific sub-event in the chain. When one of the methods determines a course of action, or it recognizes a fatal error condition, it can issue an ABORT (and optionally a special value) that completely collapses the nested call chain and prevents all the intermediate methods from continuing.

When ABORT appears without the optional *Value*, it returns zero (FALSE).

#### About the Call Stack

When methods are called, a hidden mechanism keeps track of important information. This mechanism is a “call stack;” an area of RAM used to store return addresses, return values, method parameters, and local variables for intermediate use. As more methods are called, additional sets of information are stacked; the call stack grows taller. As methods exit, this temporary information is removed and the call stack gets shorter. This process is called "pushing” onto the stack and “popping” off of the stack, respectively.

The ABORT command repetitively pops data off the call stack until it reaches a caller with an Abort Trap (see below); returning to some higher-level caller potentially many steps up the nested call chain. Any return points along the way between an aborting method and an abort trapping method are ignored and terminated. In this way, ABORT allows code to back out of very deep and complicated logic to handle a serious issue at a high level.

#### Using ABORT

Any method can choose to issue an ABORT command in addition to a normal RETURN command. Usually an abort is triggered by a special condition, so an IF or CASE construct is used.

if <bad condition>

abort 'If bad condition detected, abort without value

—or—

case <test value>

...

<bad condition>: abort <value> 'If bad condition detected, abort with value

...

<bad condition> is a test that determines the method should abort and <value> is a value to return upon aborting.

#### The Abort Trap ( \ )

It’s up to the caller to check for a method's abort status and handle it, otherwise that caller is eliminated in the event of an abort.

To trap an ABORT, the call to the *abortable* method must be preceded with the [Abort Trap](#_dg8ox9ta1b80) symbol, a backslash (\). For example, when a method named MayAbort might abort itself, or if it calls other methods that could abort, its caller can use the following approach to trap the abort:

**\**MayAbort 'Call MayAbort with trap; ignore abort value

<continue regardless> 'Perform rest of code regardless

—or—

if **\**MayAbort 'Call MayAbort with trap and, if non-zero...

<handle abort case> 'Process abort

When a method is called with an Abort Trap, a single *abort value* will always be returned; 0 (FALSE) by default. If the method (MayAbort in this case) executes an ABORT command with a value, that value will be returned to the trapping caller. Example: ABORT True would return TRUE (-1) to the caller in the example above.

In the example above, if MayAbort used a RETURN instead, execution would still return to the direct, trapping caller, but the value would always be 0 (FALSE) regardless of what RETURN specified, since the Abort Trap always retrieves the special abort value.

#### Example Use Of Abort

This code demonstrates the ABORT command for a robot that avoids objects approaching it and beeps if its motors get stuck. CheckSensors, Beep, and MotorStuck are assumed to be defined elsewhere.

**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**

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>

Three methods manage robot behavior:

* Main(): [high-level] - Decides movement based on sensor readings
* Move(): [mid-level] - Manages short-distance movement in a direction
* DriveMotors(): [low-level] - Handles motor control and stall detection

This approach, with ABORT, allows critical events in low-level code to be handled by high-level code without complex communication, even with nested mid-level methods.

In the Main() method, when an approaching object is sensed, it calls Move() with the Abort Trap symbol, \ , preceding it. The Move() method itself *never* aborts; however, it calls DriveMotors() which *can* abort. Move() always returns TRUE when done, but if DriveMotors() detects a stalled motor, it aborts with FALSE and pops everything from the stack back up to Main(), which handles the abort with a beep. Note that Move() is oblivious to an abort; it ceases to exist when one happens.

#### An Abort without Abort Trap (\)

If the Abort Trap were missing in the example above, the abort action pops the whole stack and terminates the cog entirely.

#### Zero or One Value Allowed

Whenever a method is called with an Abort Trap (\), it will always have only one return value regardless of how many return values its declaration has. This overriding return value is either zero (0; a.k.a FALSE) when ABORT is given with no *Value* specified, or is the specified *Value*.

## CASE / CASE\_FAST

**Command:** Compare expression against matching expression(s) and execute code block if match found.

⦅**PUB**┆**PRI**⦆

**CASE**/**CASE\_FAST** *CaseExpression*

⇢ *MatchExpression* : { ↪ } *Code*

{ ⇢ *MatchExpression* : { ↪ } *Code* }…

{ ⇢ **OTHER** : { ↪ } *Code* }

* ***CaseExpression*** is the expression to compare.
* ***MatchExpression*** is a single or comma-delimited set of value-expressions and/or range-expressions, to compare *CaseExpression* against. Each *MatchExpression* must be followed by a colon (:).
* ***Code*** is a block of one or more lines of code to execute when the *CaseExpression* matches the associated *MatchExpression* or OTHER. The first, or only, statement in *Code* may appear to the right of the colon on the *MatchExpression*/OTHER line, or below it and indented from *MatchExpression*/OTHER itself.

**Explanation**

CASE and CASE\_FAST are part of three flow control constructs (IF/IFNOT, CASE/CASE\_FAST, and REPEAT) that conditionally executes a block of code. CASE/CASE\_FAST is the preferred structure to use, as opposed to IF..ELSEIF..ELSE, when you need to compare the equality of *CaseExpression* to a number of different values.

CASE/CASE\_FAST compares *CaseExpression* against the values of each *MatchExpression*, in order, and if a match is found, executes the associated *Code*. If no previous matches are found, the *Code* associated with the optional OTHER command are executed.

#### Indention is Critical

IMPORTANT: Indention is critical. The Spin2 language relies on indention (of one space or more) on lines following flow control constructs to determine if they belong to that construct or not. Your code editor may also visually indicate such groupings. If using the Propeller Tool, Ctrl + I toggles block-group indicators on/off.

#### Using CASE

CASE is handy where one of many actions needs to be performed depending on the value of an expression. The following example assumes A, X, and Y are variables defined earlier.

case X+Y 'Test X+Y

10, 15: pintoggle(0) 'X+Y = 10 or 15? Toggle P0

A\*2 : pintoggle(1) 'X+Y = A\*2? Toggle P1

30..40: pintoggle(2) 'X+Y in 30 to 40? Toggle P2

X += 5 'Add 5 to X

Since the *MatchExpression* lines are indented from the CASE line, they belong to the CASE structure and are executed based on the *CaseExpression* comparison results. The last line, X += 5, is not indented from CASE, so it is executed after, and regardless of, the CASE results.

This example compares the value of X+Y against 10 or 15, A\*2, and the range 30 through 40. If X+Y equals 10 or 15, P0 is toggled. If X+Y equals A\*2, P1 is toggled. If X+Y is in the range 30 through 40, inclusive, then P2 is toggled. Whether or not any match was found, the X += 5 line is executed next.

#### Using OTHER

The optional OTHER component of CASE is similar to the optional ELSE component of an IF structure. It should appear at the end of the CASE block if there's an action needed when no other match is found. For example:

case X+Y 'Test X+Y

10, 15: pintoggle(0) 'X+Y = 10 or 15? Toggle P0

25 : pintoggle(1) 'X+Y = 25? Toggle P1

20..30: pintoggle(2) 'X+Y in 20 to 30? Toggle P2

OTHER : pintoggle(3) 'Otherwise toggle P3

X += 5 'Add 5 to X

This example includes an OTHER statement. If X+Y does not equal 10, 15, 25, or is not in the range 20 to 30, the *Code* block following OTHER is executed. After the CASE construct, the X += 5 line is executed.

There is an important concept to note about this example: the *MatchExpressions* are evaluated in order, so the first valid match will execute and ignore the rest. If we had arranged the last two *MatchExpressions* above OTHER so that the range of 20..30 is checked before 25, we’d have a bug in our code. Here's similar code with that bug:

case X+Y 'Test X+Y

10, 15: pintoggle(0) 'X+Y = 10 or 15? Toggle P0

20..30: pintoggle(2) 'X+Y in 20 to 30? Toggle P2

25 : pintoggle(1) 'X+Y = 25? Toggle P1 <-- THIS NEVER RUNS

The above example contains an error because, while X+Y could be equal to 25, that exact match expression would never be tested since the previous one, range of 20..30 would be tested first and satisfies the case, so its block is executed and no further match expressions are checked.

#### Using CASE\_FAST

CASE\_FAST is like CASE, but rather than sequentially comparing the target to a list of possible matches, it uses an indexed jump table of up to 256 entries to immediately branch to the appropriate *Code* block, saving time at a possible cost of larger compiled code. However, the resulting CASE\_FAST code will be smaller than an equivalent CASE construct including several match values, if those values are all contiguous and don't include match ranges.

CASE\_FAST *MatchExpressions* can contain single or comma-delimited values and/or range-expressions, values can be contiguous or not, but all values must all be unique constants that are within 255 of each other. The order of *MatchExpressions* doesn't matter in CASE\_FAST since, at run-time, *CaseExpression* feeds into an indexed jump table.

CASE\_FAST chr

0: BYTEFILL(@screen, " ", screen\_size)

col := row := 0

1: col := row := 0

2..7: flag := chr

RETURN

8: IF col

col--

9: REPEAT

out(" ")

WHILE col & 7

10: RETURN

11: color := $00

12: color := $80

13: newline()

OTHER: out(chr)

#### Code Variations

Each *Code* block can be many lines and may also appear below, and indented from, the *MatchExpression* itself. The following two examples show these variations.

case A 'Test A

4 **:** pintoggle(0) 'A = 4? Toggle P0

Z+1 : pintoggle(1) 'A = Z+1? Toggle P1

pintoggle(2) 'And toggle P2

10..15: pintoggle(3) 'A in 10 to 15? Toggle P3

case A 'Test A

4**:** 'A = 4?

pintoggle(0) 'Toggle P0

Z+1: 'A = Z+1?

pintoggle(1) 'Toggle P1

pintoggle(2) 'And toggle P2

10..15: 'A in 10 to 15?

pintoggle(3) 'Toggle P3

## IF / IFNOT

**Command:** Test condition(s) and execute a block of code if true (IF) or if false (IFNOT).

⦅**PUB**┆**PRI**⦆

**IF**/**IFNOT** *Condition(s)*

⇢ *If/IfNot-Code*

{ **ELSEIF**/**ELSEIFNOT** *Condition*(s)

⇢ *ElseIf/ElseIfNot-Code* }…

{ **ELSE**

⇢ *Else-Code* }

* ***Condition(s)*** is one or more Boolean expressions to test.
* ***If/IfNot-Code*** is a block of code to execute when the IF/IFNOT Condition(s) is valid.
* ***ElseIf/ElseIfNot-Code*** is an optional block of code to execute when all the previous Condition(s) are invalid and the ELSEIF/ELSEIFNOT Condition(s) is value.
* ***Else-Code*** is an optional block of code to execute when all the previous Condition(s) are invalid.

**Explanation**

IF and IFNOT are part of three flow control constructs (IF/IFNOT, CASE/CASE\_FAST, and REPEAT) that conditionally executes a block of code. IF/IFNOT can optionally be combined with one or more ELSEIF and ELSEIFNOT commands and/or one ELSE command to form sophisticated conditional structures.

IF and IFNOT tests *Condition(s)* and, if valid, executes the indented *If/IfNot-Code* block immediately below it. If *Condition(s)* is invalid, the following optional ELSEIF *Condition(s)* and/or ELSEIFNOT *Condition(s)* are tested, in order, until a valid condition is found, then the associated *ElseIf/ElseIfNot-Code* block is executed. The optional *Else-Code* block is executed if no previous valid conditions are found.

A “valid” condition is one that evaluates to TRUE for a positive conditional statement (IF or ELSEIF) or evaluates to FALSE for a negative conditional statement (IFNOT or ELSEIFNOT). The remaining explanation below often uses the positive conditional (IF or ELSEIF) but similar concepts apply to the negative conditional (IFNOT or ELSEIFNOT).

#### Indention is Critical

IMPORTANT: Indention is critical. The Spin2 language relies on indention (of one space or more) on lines following flow control constructs to determine if they belong to that construct or not. Your code editor may also visually indicate such groupings. If using the Propeller Tool, Ctrl + I toggles block-group indicators on/off.

#### Simple IF Statement

The most common form of the IF command performs an action if, and only if, a condition is true. This is written as an IF statement followed by one or more indented lines of code. For example:

if X > 10 'If X is greater than 10

pintoggle(0) 'Toggle P0

pintoggle(1) 'Toggle P1

This example tests if X is greater than 10; if it is, I/O pin 0 is toggled. Whether or not the IF condition was true, I/O pin P1 is toggled next.

Since the pintoggle(0) line is indented from the IF line, it belongs to the *If-Code* block and is executed only if the IF condition is true. The next line, pintoggle(1), is not indented from the IF line, so it is executed next whether or not the IF’s *Condition(s)* was true. Here’s another version of the same example:

if X > 10 'If X is greater than 10

pintoggle(0) 'Toggle P0

pintoggle(1) 'Toggle P1

waitms(1000) 'Wait for 1 second

This example is very similar to the first, except there are now two lines of code indented from the IF statement. In this case, if X is greater than 10, P0 is toggled, then P1 is toggled, and finally the waitms line is executed. If, however, X was not greater than 10, the pintoggle(0) and pintoggle(1) lines are skipped (since they are indented and part of the *If-Code* block) and the waitms line is executed (since it is not indented; it is not part of the *If-Code* block).

#### Combining Conditions

The *Condition(s)* field is evaluated as one single Boolean condition, but it can be made up of multiple expressions by combining them with the AND and OR operators. For example:

if X > 10 AND X < 100 'If X greater than 10 and less than 100

This IF statement would be true if, and only if, X is greater than 10 and X is also less than 100. In other words, it’s true if X is in the range 11 to 99. Sometimes statements like these can be a little difficult to read. To make it easier to read, parentheses may be used to group each sub-condition, such as with the following.

if **(**X > 10**)** AND **(**X < 100**)** 'If X greater than 10 and less than 100

#### Using IF with ELSE

The next common form of the IF command performs an action if a condition is true or a different action if that condition is false. This is written as an IF statement followed by its *If-Code* block, then an ELSE followed by its *Else-Code* block, as shown below:

if X > 100 'If X is greater than 100

pintoggle(0) 'Toggle P0

else 'Else, X <= 100

pintoggle(1) 'Toggle P1

Here, if X is greater than 100 then I/O pin 0 is toggled, otherwise X must be less than or equal to 100 and I/O pin 1 is toggled. This IF...ELSE construct, as written, always performs either a toggle on P0 or a toggle on P1; never both, and never neither.

Remember, the code that logically belongs to the *If-Code* or the *Else-Code* blocks must be indented from the IF or the ELSE, respectively, by at least one space. Also note that the ELSE must be lined up horizontally with the IF statement; they must both begin on the same column so the compiler knows that the ELSE goes with that IF.

For every IF/IFNOT statement, there can be zero or one ELSE component. ELSE, if included, must be the last component in an IF/IFNOT statement, appearing after all potential ELSEIF/ELSEIFNOTs.

#### Using IF with ELSEIF

IF can perform some action if a condition is true or a different action if that condition is false but another condition is true. This is written as an IF statement followed by its *If-Code* block, then one or more ELSEIF statements followed by their respective *ElseIf-Code* blocks. Here’s an example:

if X > 100 'If X is greater than 100

pintoggle(0) 'Toggle P0

elseif X == 90 'Else If X = 90

pintoggle(1) 'Toggle P1

Here, if X is greater than 100 then I/O pin 0 is toggled, otherwise if X equals 90 then I/O pin 1 is toggled, and if neither of those conditions are true, neither P0 nor P1 is toggled. This is a slightly shorter way of writing the following code:

if X > 100 'If X is greater than 100

pintoggle(0) 'Toggle P0

else 'Otherwise,

if X == 90 'If X = 90

pintoggle(1) 'Toggle P1

Both of these examples perform the same actions, but the first is shorter and easier to read. Note that the ELSEIF, just like the ELSE, must be lined up with the same column as the IF that it is associated with.

Each IF conditional statement can have zero or more ELSEIF statements associated with it. Look at the following:

if X > 100 'If X is greater than 100

pintoggle(0) 'Toggle P0

elseif X == 90 'Else If X = 90

pintoggle(1) 'Toggle P1

elseif X > 50 'Else If X > 50

pintoggle(2) 'Toggle P2

We have three conditions and three possible actions here. Just like the previous example, if X is greater than 100, P0 is toggled, otherwise, if X equals 90, P1 is toggled, but if neither of those conditions are true and X is greater than 50, P2 is toggled. If none of those conditions are true, then none of those actions would occur.

There is an important concept to note about this example: the conditions are evaluated in order, so a valid condition will execute and ignore the rest. If we had arranged the two ELSEIFs so that the “X > 50” were checked first, we’d have a bug in our code. Here's the code with that bug:

if X > 100 'If X is greater than 100

pintoggle(0) 'Toggle P0

elseif X > 50 'Else If X > 50

pintoggle(2) 'Toggle P2

elseif X == 90 'Else If X = 90 <-- ERROR, ABOVE CONDITION

pintoggle(1) 'Toggle P1 <-- SUPERSEDES THIS AND

' THIS CODE NEVER RUNS

The above example contains an error. While X could be equal to 90, the elseif X == 90 statement would never be tested since the previous one, elseif X > 50, is tested first and is also true, so its block would be executed and no further conditions of that IF structure are tested. Also, if X were 50 or less, the last ELSEIF condition is tested, but of course, it will never be true.

#### Using IF with ELSEIF and ELSE

An enhancement to the above IF conditional structures performs an alternate action if none of the previous conditions were true. This is written as an IF, one or more ELSEIFs, and finally an ELSE. Here’s an example:

if X > 100 'If X is greater than 100

pintoggle(0) 'Toggle P0

elseif X == 90 'Else If X = 90

pintoggle(1) 'Toggle P1

elseif X > 50 'Else If X > 50

pintoggle(2) 'Toggle P2

else 'Otherwise,

pintoggle(3) 'Toggle P3

In the above example, P3 is toggled if none of the prior IF or ELSEIF conditions are true.

#### The ELSEIFNOT Condition

The ELSEIFNOT condition behaves exactly like ELSEIF except that it uses negative logic; it executes its Elseif not-Code block only if its *Condition(s)* evaluates to FALSE. Multiple ELSEIFNOT and ELSEIF conditions can be combined in a single IF conditional construct, in any order, between the IF and the optional ELSE.

## REPEAT

**Command:** Execute code block repetitively.

⦅**PUB**┆**PRI**⦆

**REPEAT**{*Count*}

{ ⇢ *Code* }

⦅**PUB**┆**PRI**⦆

**REPEAT** *Count* **WITH** *Variable*

{ ⇢ *Code* }

⦅**PUB**┆**PRI**⦆

**REPEAT** *Variable* **FROM** *Start* **TO** *Finish* {**STEP** *Delta*}

{ ⇢ *Code* }

⦅**PUB**┆**PRI**⦆

**REPEAT**⦅**UNTIL**┆**WHILE**⦆ *Condition(s)*

{ ⇢ *Code* }

⦅**PUB**┆**PRI**⦆

**REPEAT**

{ ⇢ *Code* }

⦅**UNTIL**┆**WHILE**⦆ *Condition(s)*

* ***Count*** is an optional expression indicating the finite number of times to execute *Code*. *Count* should be a positive value. If *Count* is omitted, syntax 1 creates an infinite loop made up of *Code*.
* ***Code*** is an optional block of one or more lines of code to execute repeatedly. Omitting *Code* is rare, but may be useful for an endless pause (syntax 1 without *Count*) to keep I/O pins active, or in syntax 4 and 5 when *Condition(s)* alone achieve the needed effects. Indention is required for all statements in the block.
* ***Variable*** is a user-defined variable that will be iterated from 0 to *Count*-1 (syntax 2) or from *Start* to *Finish* and optionally by *Delta* units per iteration (syntax 3). *Variable* can be used in *Code* to determine or utilize the iteration count.
* ***Start*** is an expression that determines the starting value of *Variable* in syntax 3. If *Start* is less than *Finish*, *Variable* will be incremented each iteration; it will be decremented otherwise.
* ***Finish*** is an expression that determines the ending value of *Variable* in syntax 3. If *Finish* is greater than *Start*, *Variable* will be incremented each iteration; it will be decremented otherwise.
* ***Delta*** is an optional expression that determines the units in which to increment/decrement *Variable* each iteration (syntax 3). If omitted, *Variable* is incremented/decremented by 1 each iteration.
* ***Condition(s)*** is one or more Boolean expression(s) used by syntax 4 and 5 to continue or terminate the loop. When preceded by UNTIL, *Condition(s)* terminates the loop when true. When preceded by WHILE, *Conditions(s)* terminates the loop when false.

**Explanation**

REPEAT is the very flexible looping structure for Spin2 code. It can be used to create any type of loop, including: infinite, finite, with/without loop counter, and conditional zero-to-many/one-to-many loops.

#### Indention is Critical

IMPORTANT: Indention is critical. The Spin2 language relies on indention (of one space or more) on lines following flow control constructs to determine if they belong to that construct or not. Your code editor may also visually indicate such groupings. If using the Propeller Tool, Ctrl + I toggles block-group indicators on/off.

#### Infinite Loops (Syntax 1 w/o Count)

Most forms of REPEAT can be infinite loops, but the form used most often for this purpose is syntax 1 without the *Count* field. For example:

repeat 'Repeat endlessly

pintoggle(25) 'Toggle P25

waitms(1000) 'Pause for 1,000 ms

This code repeats the pintoggle(25) and waitms(1000) lines endlessly. Both lines are indented from the REPEAT so they belong to the REPEAT loop.

Since *Code* is really an optional part of REPEAT, the REPEAT command by itself can be used as an endless loop that does nothing but keep the cog active— including its influence on I/O pins. This can be intentional, but may be unintentional due to improper indentation. For example:

repeat 'Repeat endlessly

pintoggle(25) 'Toggle P25 <-- This is never run

The above example is erroneous; the last line is never executed because the REPEAT above it is an endless loop that has no *Code*— there is nothing indented immediately below it, so the cog simply sits in an endless loop at the REPEAT line that does nothing but keep the cog active and consuming power.

#### Simple Finite Loops (Syntax 1)

Most loops are finite in nature; they execute a limited number of iterations only. The simplest form is syntax 1 with the *Count* field included.

For example:

repeat 10 'Repeat 10 times

pintoggle(25) 'Toggle P25

waitms(500) 'Pause for 500 ms

byte[$7000]++ 'Increment RAM location $7000

This code toggles P25 ten times, waiting 500 ms in-between, then increments the value in RAM location $7000.

Note that *Count* should be a positive value; negatives will yield unwanted results. Also, the *Count* field may be any numeric expression but the expression is evaluated only once, the first time the loop is entered. This means that any changes to the expression’s variables within the loop will not affect the number of iterations of the loop. The next example assumes the variable Index was created previously.

Index := 10 'Set loop to repeat 10 times

repeat Index 'Repeat Index times

pintoggle(25) 'Toggle P25

waitms(500) 'Pause for 500 ms

Index := 20 'Change Index to 20

In the above example, Index is 10 upon entering the REPEAT loop the first time. Each time through the loop, however, Index is set to 20, but the loop continues to execute only 10 times.

#### Counted Finite Loops (Syntax 2 & 3)

Quite often it is necessary to count the loop iterations so the loop’s code can perform differently based on that count. The REPEAT command makes it easy to do this with syntax 2 and 3. The next example assumes the variable Index was created previously.

repeat 10 with Index 'Repeat 10 times using Index as the counter

byte[$7000][Index]++ 'Increment RAM locations $7000 to $7009

The code above loops 10 times, adjusting the value in the variable Index on each iteration; starting at 0 and ending after 9. The first time through the loop, Index will be 0 and each iteration afterwards Index will be 1 greater than the previous iteration: ..1, 2, 3…9. After the tenth iteration, Index will be incremented to 10 and the loop will terminate, causing the next code following the REPEAT loop structure to execute, if any exists. The code in the loop uses Index as an offset to affect memory, byte[$7000][Index]++; in this case it is incrementing each of the byte-sized values in RAM locations $7000 to $7009 by 1, one at a time.

Note that *Count* should be a positive value; negatives will yield unwanted results.

Syntax 3 serves a similar function as the example above, though with added flexibility.

repeat Index from 1 to 10 'Repeat 10 times

byte[$7000][Index]++ 'Increment RAM locations $7000 to $7009

Like the previous example, the code above loops 10 times, adjusting the variable Index. The difference is that syntax 3 allows starting at any arbitrary value; 1 in this case. The first time through the loop, Index will be 1 (as indicated by the “from 1”) and continues iterating until Index equals 11. Be careful of *Start* and *Finish* values; this example affects RAM locations $7001 to $7010 instead of the previous example's $7000 to $7009.

The REPEAT command automatically determines whether the range suggested by *Start* and *Finish* is increasing or decreasing. Since the above example used 1 to 10, the range is an increasing range; adjusting Index by +1 every time. To get the count to go backwards, simply reverse the *Start* and *Finish* values. Here's an example:

repeat Index from 9 to 0 'Repeat 10 times

byte[$7000][Index]++ 'Increment RAM $7009 down through $7000

This example also loops 10 times, but counts with Index from 9 down to 0; adjusting Index by -1 each time. The contents of the loop still increments the values in RAM, but from locations $7009 down to $7000. After the tenth iteration, Index will equal -1.

Since the *Start* and *Finish* fields can be expressions, they can contain variables. The next example assumes that S and F are variables created previously.

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

The above example uses a nested loop. The outer loop (the first one) repeats 2 times. The inner loop repeats with Index from S to F, which were previously set to 0 and 9, respectively. The inner loop increments the values in RAM locations $7000 to $7009, in that order, because the inner loop is counting iterations from 0 to 9. Then, the inner loop terminates (with Index being set to 10) and the last two lines set S to 9 and F to 0, effectively swapping the *Start* and *Finish* values. Since this is still inside the outer loop, the outer loop then executes its contents again (for the second time) causing the inner loop to repeat with Index from 9 down to 0. The inner loop increments the values in RAM locations $7009 to $7000, in that order (reverse of the previous time) and terminates with Index equaling -1. The last two lines set S and F again, but the outer loop does not repeat a third time.

REPEAT loops don’t have to be limited to incrementing or decrementing by 1 either. If the REPEAT command uses the optional STEP *Delta* syntax, it will increment or decrement the *Variable* by the *Delta* amount. In the syntax 2 form, REPEAT is actually always using a *Delta* value, but when the “STEP *Delta*” component is omitted, it uses either +1 or -1 by default, depending on the range of *Start* and *Finish*. The following example includes the optional *Delta* value to increment by 2.

repeat Index from 0 to 8 step 2 'Repeat 5 times

byte[$7000][Index]++ 'Increment even RAM $7000 to $7008

Here, REPEAT loops five times, with Index set to 0, 2, 4, 6, and 8, respectively. This code effectively increments every other RAM location (the even numbered locations) from $7000 to $7008 and terminates with Index equaling 10.

The *Delta* field can be positive or negative, regardless of the natural ascending/descending range of the *Start* and *Finish* values, and can even be adjusted within the loop to achieve interesting effects. For example, assuming Index and D are previously defined variables, the following code sets Index to the following sequence: 5, 6, 6, 5, 3.

D := 2

repeat Index from 5 to 10 step D

--D

This loop started out with Index at 5 and a *Delta* (D) of +2. But each iteration of the loop decrements D by one, so at the end of iteration 1, Index = 5 and D = +1. Iteration 2 has Index = 6 and D = 0. Iteration 3 has Index = 6 and D = -1. Iteration 4 has Index = 5 and D = -2. Iteration 5 has Index = 3 and D = -3. The loop then terminates because Index plus *Delta* (3 + -3) is outside the range of *Start* to *Finish* (5 to 10).

#### Conditional Loops (Syntax 3 and 4)

The final forms of REPEAT, syntax 3 and 4, are finite loops with conditional exits and have flexible options allowing for the use of either positive or negative logic and the creation of zero-to-many or one-to-many iteration loops. These two forms of REPEAT are usually referred to as “repeat while” or “repeat until” loops.

Let’s look at the REPEAT form described by syntax 3. It consists of the REPEAT command followed immediately by either WHILE or UNTIL then *Condition(s)* and finally, on the lines below it, optional *Code*. Since this form tests *Condition(s)* at the start of every iteration, it creates a zero-to-many loop; the *Code* block will execute zero or more times, depending on the *Condition(s)*. For example, assume that X is a variable created earlier:

X := 0

repeat while X < 10 'Repeat while X is less than 10

byte[$7000][X] := 0 'Increment RAM value

X++ 'Increment X

This example first sets X to 0, then repeats the loop while X is less than 10. The code inside the loop clears RAM locations based on X (starting at location $7000) and increments X.

After the 10th iteration of the loop, X equals 10, making the condition while X < 10 false and the loop terminates.

This loop is said to use “positive” logic because it continues “WHILE” a condition is true. It could also be written with “negative” logic using UNTIL, instead. Such as:

X := 0

repeat until X > 9 'Repeat until X is greater than 9

byte[$7000][X] := 0 'Increment RAM value

X++ 'Increment X

The above example performs the same way as the previous, but the REPEAT loop uses negative logic because it continues “UNTIL” a condition is true; i.e: it continues while a condition is false.

In either example, if X was equal to 10 or higher before the first iteration of the REPEAT loop, the condition would cause the loop to never execute at all, which is why we call it a zero-to-many loop.

The REPEAT form described by syntax 4 is very similar to syntax 3, but the condition is tested at the end of every iteration, making it a one-to-many loop. For example:

X := 0

repeat

byte[$7000][X] := 0 'Increment RAM value

X++ 'Increment X

while X < 10 'Repeat while X is less than 10

This works the same as the previous examples, looping 10 times, except that the condition is not tested until the end of each iteration. However, unlike the previous examples, even if X was equal to 10 or higher before the first iteration, the loop would run once then terminate, which is why we call it a one-to-many loop.

#### Other REPEAT Options

There are two other commands that affect the behavior of REPEAT loops: NEXT and QUIT. See their respective descriptions for more information.

## RETURN

**Command:** Exit with normal status from PUB/PRI method with optional return *Value(s)*.

⦅**PUB**┆**PRI**⦆

**RETURN** {*Value(s)*}

**Returns:** Either the method's current defined results, or *Value(s)* if provided.

* ***Value(s)*** is an optional set of one or more expressions whose values are to be returned from the method. Up to 15 comma-delimited values can be provided, matching the defined set of PUB/PRI method results.

**Explanation**

RETURN is one of two commands (ABORT and RETURN) that immediately terminates a PUB or PRI method.

RETURN cau the PUB or PRI method to exit with normal status— meaning it pops the call stack once and returns to the direct caller of the method, optionally delivering one or more "return" values in the process.

Every PUB or PRI method has an implied RETURN at its end, so there's no need to manually place one; however, RETURN can be used in multiple places within the method to create multiple exit points— such as in IF or CASE statements.

When RETURN appears without the optional *Value(s)*, it returns the current value(s) of the PUB/PRI's defined *Result(s)*, which can be set or changed in many places within the method. If *Value(s)* are provided with the RETURN, they are returned to the caller instead of the current *Result(s)*.

#### About the Call Stack

When methods are called, a hidden mechanism keeps track of important information. This mechanism is a “call stack;” an area of RAM used to store return addresses, return values, method parameters, and local variables for intermediate use. As more methods are called, additional sets of information are stacked; the call stack grows taller. As methods exit, this temporary information is removed and the call stack gets shorter. This process is called "pushing” onto the stack and “popping” off of the stack, respectively.

The RETURN command pops the most recent data off the call stack to facilitate returning to the immediate caller; the one who directly called the method that just returned.

#### Using RETURN

The following example demonstrates two uses of RETURN. Assume that DisplayDivByZeroError is a method defined elsewhere.

**PUB Add(Num1, Num2): Sum**

Sum:= Num1 + Num2 'Add Num1 + Num2

return

**PUB Divide(Dividend, Divisor): Result**

if Divisor == 0 'Check if Divisor = 0

DisplayDivByZeroError 'If so, display error

return 0 'and return with 0

return Dividend / Divisor 'Otherwise return quotient

The Add() method sets its Sum variable equal to Num1 plus Num2, then executes RETURN. The RETURN causes Add() to return the value of Sum to the caller. Note that this RETURN was not really required because the Compiler automatically adds it to the end of methods.

The Divide() method uses RETURN in a different way. If Divisor equals zero, it calls a DisplayDivByZeroError() method and then executes return 0, so the method returns to the caller with the single value 0. Otherwise, it executes return Dividend / Divisor, returning to the caller with the result of the division. In this case, the last RETURN was used to perform the calculation and return the result all in one step rather than separately affecting the declared Result variable beforehand.

#### Multiple Return Values

PUB and PRI methods will return multiple values just by defining their method that way. For example:

PUB Divide(Dividend, Divisor): Quotient, Remainder

Quotient := Dividend / Divisor 'Calculate quotient

Remainder := Dividend // Divisor 'Calculate remainder

Here, the Divide() method defines two results, Quotient and Remainder, and sets their values during internal division calculations. The resulting values are automatically returned to the caller upon method exit. However, the Divide method could also calculate and return the results in one RETURN statement, without ever explicitly setting the Quotient and Remainder result variables:

PUB Divide(Dividend, Divisor): Quotient, Remainder

return Dividend/Divisor, Dividend//Divisor 'Return quotient & remainder

Either way, all results will be returned in their defined order, and all must be specified if any are provided via the RETURN statement. The caller can either ignore them all or receive them all.

Below is an example of a caller. The Display() method calls Divide() and retrieves the two results by providing a comma-delimited list of local receiving variables in the proper order, Q, R := Divide(...).

PUB Display() | Q, R

Q, R := Divide(1023, 16) 'Call Divide() to get whole and part

debug(udec(Q), udec(R)) 'Display results

When combined with one of the above and executed, Display() will send "Q = 63, R = 15" to the Debug Output window.

# Propeller 2 Reserved Words (Spin2 + PASM2)

Predefined symbols recognized by the compiler to have special meaning.

## \_ (leading underscore)

| \_C  \_C\_AND\_NZ  \_C\_AND\_Z  \_C\_EQ\_Z  \_C\_NE\_Z | \_C\_OR\_NZ  \_C\_OR\_Z  \_CLKFREQ  \_CLKMODE  \_CLR | \_E  \_GE  \_GT  \_LE  \_LT | \_NC  \_NC\_AND\_NZ  \_NC\_AND\_Z  \_NC\_OR\_NZ  \_NC\_OR\_Z | \_NE  \_NZ  \_NZ\_AND\_C  \_NZ\_AND\_NC  \_NZ\_OR\_C | \_NZ\_OR\_NC  \_RET\_  \_SET  \_Z  \_Z\_AND\_C | \_Z\_AND\_NC  \_Z\_EQ\_C  \_Z\_NE\_C  \_Z\_OR\_C  \_Z\_OR\_NC |
| --- | --- | --- | --- | --- | --- | --- |

## A - B

| ABORT  ABS  ADD  ADDBITS  ADDCT1  ADDCT2 | ADDCT3  ADDPINS  ADDPIX  ADDS  ADDSX  ADDX | AKPIN  ALIGNL  ALIGNW  ALLOWI  ALT  ALTB | ALTD  ALTGB  ALTGN  ALTGW  ALTI  ALTR | ALTS  ALTSB  ALTSN  ALTSW  AND  ANDC | ANDN  ANDZ  ARCHIVE  ASMCLK  AUGD  AUGS | AUTO  BACKCOLOR  BITC  BITH  BITL  BITMAP | BITNC  BITNOT  BITNZ  BITRND  BITZ  BLACK | BLNPIX  BLUE  BMASK  BOOL  BOOL\_  BOX | BRK  BYTE  BYTECOMP  BYTEFILL  BYTEFIT  BYTEMOVE | BYTESWAP  BYTES\_1BIT  BYTES\_2BIT  BYTES\_4BIT |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## C - D

| CALL  CALLA  CALLB  CALLD  CALLPA  CALLPB  CARTESIAN  CASE  CASE\_FAST  CHANNEL  CIRCLE  CLEAR | CLKFREQ  CLKFREQ\_  CLKMODE  CLKMODE\_  CLKSET  CLOSE  CMP  CMPM  CMPR  CMPS  CMPSUB  CMPSX | CMPX  COGATN  COGBRK  COGCHK  COGEXEC  COGEXEC\_NEW  COGEXEC\_NEW\_PAIR  COGID  COGINIT  COGSPIN  COGSTOP  COLOR | COMP  CON  COPY  CRCBIT  CRCNIB  CYAN  DAT  DEBUG  DEBUG\_BAUD  DEBUG\_COGINIT  DEBUG\_COGS  DEBUG\_DELAY | DEBUG\_DISPLAY\_LEFT  DEBUG\_DISPLAY\_TOP  DEBUG\_HEIGHT  DEBUG\_LEFT  DEBUG\_LOG\_SIZE  DEBUG\_MAIN  DEBUG\_PIN\_TX  DEBUG\_PIN\_RX  DEBUG\_TIMESTAMP  DEBUG\_TOP  DEBUG\_WIDTH  DEBUG\_WINDOWS\_OFF | DECMOD  DECOD  DEPTH  DEV  DIRA  DIRB  DIRC  DIRH  DIRL  DIRNC  DIRNOT  DIRNZ | DIRRND  DIRZ  DJF  DJNF  DJNZ  DJZ  DLY  DOT  DOTSIZE  DOWNLOAD\_BAUD  DRVC  DRVH | DRVL  DRVNC  DRVNOT  DRVRND  DRVZ |
| --- | --- | --- | --- | --- | --- | --- | --- |

## E - F

| ELSE  ELSEIF  ELSEIFNOT  ENCOD  END | EVENT\_ATN  EVENT\_CT1  EVENT\_CT2  EVENT\_CT3  EVENT\_FBW | EVENT\_INT  EVENT\_PAT  EVENT\_QMT  EVENT\_SE1  EVENT\_SE2 | EVENT\_SE3  EVENT\_SE4  EVENT\_XFI  EVENT\_XMT  EVENT\_XRL | EVENT\_XRO  EXECF  FABS  FALSE  FBLOCK | FDEC  FDEC\_  FDEC\_ARRAY  FDEC\_ARRAY\_  FDEC\_REG\_ARRAY | FDEC\_REG\_ARRAY\_  FFT  FGE  FGES  FILE | FIT  FIELD  FILL  FLE  FLES | FLOAT  FLTC  FLTH  FLTL  FLTNC | FLTNOT  FLTNZ  FLTRND  FLTZ  FRAC | FROM  FSQRT  FVAR  FVARS |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## G - H

| GETBRK  GETBYTE  GETCRC | GETCT  GETMS  GETNIB | GETPTR  GETQX  GETQY | GETREGS  GETRND  GETSCP | GETSEC  GETWORD  GETXACC | GRAY  GREEN  GREY | HIDEXY  HOLDOFF  HSV16 | HSV16W  HSV16X  HSV8 | HSV8W  HSV8X  HUBEXEC | HUBEXEC\_NEW  HUBEXEC\_NEW\_PAIR  HUBSET |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## I - J

| IF  IF\_00  IF\_0000  IF\_0001  IF\_0010  IF\_0011  IF\_01  IF\_0100  IF\_0101  IF\_0110  IF\_0111  IF\_0X | IF\_10  IF\_1000  IF\_1001  IF\_1010  IF\_1011  IF\_11  IF\_1100  IF\_1101  IF\_1110  IF\_1111  IF\_1X  IF\_A | IF\_AE  IF\_ALWAYS  IF\_B  IF\_BE  IF\_C  IF\_C\_AND\_NZ  IF\_C\_AND\_Z  IF\_C\_EQ\_Z  IF\_C\_NE\_Z  IF\_C\_OR\_NZ  IF\_C\_OR\_Z  IF\_DIFF | IF\_E  IF\_GE  IF\_GT  IF\_LE  IF\_LT  IF\_NC  IF\_NC\_AND\_NZ  IF\_NC\_AND\_Z  IF\_NC\_OR\_NZ  IF\_NC\_OR\_Z  IF\_NE  IF\_NOT\_00 | IF\_NOT\_01  IF\_NOT\_10  IF\_NOT\_11  IF\_NZ  IF\_NZ\_AND\_C  IF\_NZ\_AND\_NC  IF\_NZ\_OR\_C  IF\_NZ\_OR\_NC  IF\_SAME  IF\_X0  IF\_X1  IF\_Z | IF\_Z\_AND\_C  IF\_Z\_AND\_NC  IF\_Z\_EQ\_C  IF\_Z\_NE\_C  IF\_Z\_OR\_C  IF\_Z\_OR\_NC  IFNOT  IJMP1  IJMP2  IJMP3  IJNZ  IJZ | INA  INB  INCMOD  INT\_OFF  IRET1  IRET2  IRET3  JATN  JCT1  JCT2  JCT3  JFBW | JINT  JMP  JMPREL  JNATN  JNCT1  JNCT2  JNCT3  JNFBW  JNINT  JNPAT  JNQMT  JNSE1 | JNSE2  JNSE3  JNSE4  JNXFI  JNXMT  JNXRL  JNXRO  JPAT  JQMT  JSE1  JSE2  JSE3 | JSE4  JXFI  JXMT  JXRL  JXRO |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## L - M

| LINE  LINESIZE  LOC  LOCKCHK  LOCKNEW  LOCKREL | LOCKRET  LOCKTRY  LOGIC  LOGSCALE  LONG  LONGCOMP | LONGFILL  LONGMOVE  LONGSWAP  LONGS\_16BIT  LONGS\_1BIT  LONGS\_2BIT | LONGS\_4BIT  LONGS\_8BIT  LOOKDOWN  LOOKDOWNZ  LOOKUP  LOOKUPZ | LSTR  LSTR\_  LSTRING  LUMA8  LUMA8W  LUMA8X | LUT1  LUT2  LUT4  LUT8  LUTCOLORS  MAG | MAGENTA  MERGEB  MERGEW  MIDI  MIXPIX  MODC | MODCZ  MODZ  MOV  MOVBYTS  MUL  MULDIV64 | MULPIX  MULS  MUXC  MUXNC  MUXNIBS  MUXNITS | MUXNZ  MUXQ  MUXZ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## N - O

| NAN  NEG  NEGC  NEGNC | NEGNZ  NEGX  NEGZ  NEWCOG | NEXT  NIXINT1  NIXINT2  NIXINT3 | NOP  NOT  OBJ  OBOX | ONES  OPACITY  OR  ORANGE | ORC  ORG  ORGF  ORGH | ORIGIN  ORZ  OTHER  OUTA | OUTB  OUTC  OUTH  OUTL | OUTNC  OUTNOT  OUTNZ  OUTRND | OUTZ  OVAL |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## P

| P\_ADC  P\_ADC\_100X  P\_ADC\_10X  P\_ADC\_1X  P\_ADC\_30X  P\_ADC\_3X  P\_ADC\_EXT  P\_ADC\_FLOAT  P\_ADC\_GIO  P\_ADC\_SCOPE  P\_ADC\_VIO  P\_AND\_AB  P\_ASYNC\_IO  P\_ASYNC\_RX  P\_ASYNC\_TX  P\_BITDAC  P\_CHANNEL  P\_COMPARE\_AB  P\_COMPARE\_AB\_FB  P\_COUNT\_HIGHS | P\_COUNT\_RISES  P\_COUNTER\_HIGHS  P\_COUNTER\_PERIODS  P\_COUNTER\_TICKS  P\_DAC\_124R\_3V  P\_DAC\_600R\_2V  P\_DAC\_75R\_2V  P\_DAC\_990R\_3V  P\_DAC\_DITHER\_PWM  P\_DAC\_DITHER\_RND  P\_DAC\_NOISE  P\_EVENTS\_TICKS  P\_FILT0\_AB  P\_FILT1\_AB  P\_FILT2\_AB  P\_FILT3\_AB  P\_HIGH\_100UA  P\_HIGH\_10UA  P\_HIGH\_150K  P\_HIGH\_15K | P\_HIGH\_1K5  P\_HIGH\_1MA  P\_HIGH\_FAST  P\_HIGH\_FLOAT  P\_HIGH\_TICKS  P\_INVERT\_A  P\_INVERT\_B  P\_INVERT\_IN  P\_INVERT\_OUT  P\_INVERT\_OUTPUT  P\_LEVEL\_A  P\_LEVEL\_A\_FBN  P\_LEVEL\_B\_FBN  P\_LEVEL\_B\_FBP  P\_LOCAL\_A  P\_LOCAL\_B  P\_LOGIC\_A  P\_LOGIC\_A\_FB  P\_LOGIC\_B\_FB  P\_LOW\_100UA | P\_LOW\_10UA  P\_LOW\_150K  P\_LOW\_15K  P\_LOW\_1K5  P\_LOW\_1MA  P\_LOW\_FAST  P\_LOW\_FLOAT  P\_MINUS1\_A  P\_MINUS1\_B  P\_MINUS2\_A  P\_MINUS2\_B  P\_MINUS3\_A  P\_MINUS3\_B  P\_NCO\_DUTY  P\_NCO\_FREQ  P\_NORMAL  P\_OE  P\_OR\_AB  P\_OUTBIT\_A  P\_OUTBIT\_B | P\_PASS\_AB  P\_PERIODS\_HIGHS  P\_PERIODS\_TICKS  P\_PLUS1\_A  P\_PLUS1\_B  P\_PLUS2\_A  P\_PLUS2\_B  P\_PLUS3\_A  P\_PLUS3\_B  P\_PULSE  P\_PWM\_SAWTOOTH  P\_PWM\_SMPS  P\_PWM\_TRIANGLE  P\_QUADRATURE  P\_REG\_UP  P\_REG\_UP\_DOWN  P\_REPOSITORY  P\_SCHMITT\_A  P\_SCHMITT\_A\_FB  P\_SCHMITT\_B\_FB | P\_STATE\_TICKS  P\_SYNC\_IO  P\_SYNC\_RX  P\_SYNC\_TX  P\_TRANSITION  P\_TRUE\_A  P\_TRUE\_B  P\_TRUE\_IN  P\_TRUE\_OUT  P\_TRUE\_OUTPUT  P\_TT\_00  P\_TT\_01  P\_TT\_10  P\_TT\_11  P\_USB\_PAIR  P\_XOR\_AB  PA  PB  PC\_KEY  PC\_MOUSE | PI  PINCLEAR  PINF  PINFLOAT  PINH  PINHIGH  PINL  PINLOW  PINR  PINREAD  PINSTART  PINT  PINTOGGLE  PINW  PINWRITE  PLOT  POLAR  POLLATN  POLLCT  POLLCT1 | POLLCT2  POLLCT3  POLLFBW  POLLINT  POLLPAT  POLLQMT  POLLSE1  POLLSE2  POLLSE3  POLLSE4  POLLXFI  POLLXMT  POLLXRL  POLLXRO  POLXY  POP  POPA  POPB  POS  POSX | PR0  PR1  PR2  PR3  PR4  PR5  PR6  PR7  PRECISE  PRECOMPILE  PRI  PTRA  PTRB  PUB  PUSH  PUSHA  PUSHB |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

## Q - R

| QCOS  QDIV  QEXP  QFRAC  QLOG  QMUL | QROTATE  QSIN  QSQRT  QUIT  QVECTOR  RANGE | RATE  RCL  RCR  RCZL  RCZR  RDBYTE | RDFAST  RDLONG  RDLUT  RDPIN  RDWORD  RECV | RED  REG  REGEXEC  REGLOAD  REP  REPEAT | RES  RESI0  RESI1  RESI2  RESI3  RET | RETA  RETB  RETI0  RETI1  RETI2  RETI3 | RETURN  REV  RFBYTE  RFLONG  RFVAR  RFVARS | RFWORD  RGB16  RGB24  RGB8  RGBEXP  RGBI8 | RGBI8W  RGBI8X  RGBSQZ  ROL  ROLBYTE  ROLNIB | ROLWORD  ROR  ROTXY  ROUND  RQPIN |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## S - T

| SAL  SAMPLES  SAR  SAVE  SBIN  SBIN\_  SBIN\_BYTE  SBIN\_BYTE\_  SBIN\_BYTE\_ARRAY  SBIN\_BYTE\_ARRAY\_  SBIN\_LONG  SBIN\_LONG\_  SBIN\_LONG\_ARRAY  SBIN\_LONG\_ARRAY\_  SBIN\_REG\_ARRAY  SBIN\_REG\_ARRAY\_  SBIN\_WORD  SBIN\_WORD\_  SBIN\_WORD\_ARRAY  SBIN\_WORD\_ARRAY\_ | SCA  SCAS  SCOPE  SCOPE\_XY  SCROLL SDEC  SDEC\_  SDEC\_BYTE  SDEC\_BYTE\_  SDEC\_BYTE\_ARRAY  SDEC\_BYTE\_ARRAY\_  SDEC\_LONG  SDEC\_LONG\_  SDEC\_LONG\_ARRAY  SDEC\_LONG\_ARRAY\_  SDEC\_REG\_ARRAY  SDEC\_REG\_ARRAY\_  SDEC\_WORD  SDEC\_WORD\_  SDEC\_WORD\_ARRAY | SDEC\_WORD\_ARRAY\_  SEND  SET  SETBYTE  SETCFRQ  SETCI  SETCMOD  SETCQ  SETCY  SETD  SETDACS  SETINT1  SETINT2  SETINT3  SETLUTS  SETNIB  SETPAT  SETPIV  SETPIX  SETQ | SETQ2  SETR  SETREGS  SETS  SETSCP  SETSE1  SETSE2  SETSE3  SETSE4  SETWORD  SETXFRQ  SEUSSF  SEUSSR  SHEX  SHEX\_  SHEX\_BYTE  SHEX\_BYTE\_  SHEX\_BYTE\_ARRAY  SHEX\_BYTE\_ARRAY\_  SHEX\_LONG | SHEX\_LONG\_  SHEX\_LONG\_ARRAY  SHEX\_LONG\_ARRAY\_  SHEX\_REG\_ARRAY  SHEX\_REG\_ARRAY\_  SHEX\_WORD  SHEX\_WORD\_  SHEX\_WORD\_ARRAY  SHEX\_WORD\_ARRAY\_  SHL  SHR  SIGNED  SIGNX  SIZE  SKIP  SKIPF  SPACING  SPARSE  SPECTRO  SPLITB | SPLITW  SPRITE  SPRITEDEF  SQRT  STALLI  STEP  STRCOPY  STRCOMP  STRING  STRSIZE  SUB  SUBR  SUBS  SUBSX  SUBX  SUMC  SUMNC  SUMNZ  SUMZ  SWAP | TERM  TEST  TESTB  TESTBN  TESTN  TESTP  TESTPN  TEXT  TEXTANGLE  TEXTSIZE  TEXTSTYLE  TITLE  TJF  TJNF  TJNS  TJNZ  TJS  TJV  TJZ  TO | TRACE  TRGINT1  TRGINT2  TRGINT3  TRIGGER  TRUE  TRUNC |
| --- | --- | --- | --- | --- | --- | --- | --- |

## U, V, W

| UBIN  UBIN\_  UBIN\_BYTE  UBIN\_BYTE\_  UBIN\_BYTE\_ARRAY  UBIN\_BYTE\_ARRAY\_  UBIN\_LONG  UBIN\_LONG\_  UBIN\_LONG\_ARRAY  UBIN\_LONG\_ARRAY\_  UBIN\_REG\_ARRAY  UBIN\_REG\_ARRAY\_  UBIN\_WORD  UBIN\_WORD\_  UBIN\_WORD\_ARRAY | UBIN\_WORD\_ARRAY\_  UDEC  UDEC\_  UDEC\_BYTE  UDEC\_BYTE\_  UDEC\_BYTE\_ARRAY  UDEC\_BYTE\_ARRAY\_  UDEC\_LONG  UDEC\_LONG\_  UDEC\_LONG\_ARRAY  UDEC\_LONG\_ARRAY\_  UDEC\_REG\_ARRAY  UDEC\_REG\_ARRAY\_  UDEC\_WORD  UDEC\_WORD\_ | UDEC\_WORD\_ARRAY  UDEC\_WORD\_ARRAY\_  UHEX  UHEX\_  UHEX\_BYTE  UHEX\_BYTE\_  UHEX\_BYTE\_ARRAY  UHEX\_BYTE\_ARRAY\_  UHEX\_LONG  UHEX\_LONG\_  UHEX\_LONG\_ARRAY  UHEX\_LONG\_ARRAY\_  UHEX\_REG\_ARRAY  UHEX\_REG\_ARRAY\_  UHEX\_WORD | UHEX\_WORD\_  UHEX\_WORD\_ARRAY  UHEX\_WORD\_ARRAY\_  UNTIL  UPDATE  VAR  VARBASE  WAITATN  WAITCT  WAITCT1  WAITCT2  WAITCT3  WAITFBW  WAITINT  WAITMS | WAITPAT  WAITSE1  WAITSE2  WAITSE3  WAITSE4  WAITUS  WAITX  WAITXFI  WAITXMT  WAITXRL  WAITXRO  WC  WCZ  WFBYTE  WFLONG | WFWORD  WHILE  WHITE  WINDOW  WITH  WMLONG  WORD  WORDCOMP  WORDFILL  WORDSWAP  WORDFIT  WORDMOVE  WORDS\_1BIT  WORDS\_2BIT  WORDS\_4BIT | WORDS\_8BIT  WRBYTE  WRC  WRFAST  WRLONG  WRLUT  WRNC  WRNZ  WRPIN  WRWORD  WRZ  WXPIN  WYPIN  WZ |
| --- | --- | --- | --- | --- | --- | --- |

## X, Y, Z

| X\_16P\_2DAC8\_WFWORD  X\_16P\_4DAC4\_WFWORD  X\_1ADC8\_0P\_1DAC8\_WFBYTE  X\_1ADC8\_8P\_2DAC8\_WFWORD  X\_1P\_1DAC1\_WFBYTE  X\_2ADC8\_0P\_2DAC8\_WFWORD  X\_2ADC8\_16P\_4DAC8\_WFLONG  X\_2P\_1DAC2\_WFBYTE  X\_2P\_2DAC1\_WFBYTE  X\_32P\_4DAC8\_WFLONG  X\_4ADC8\_0P\_4DAC8\_WFLONG  X\_4P\_1DAC4\_WFBYTE  X\_4P\_2DAC2\_WFBYTE  X\_4P\_4DAC1\_WFBYTE  X\_8P\_1DAC8\_WFBYTE  X\_8P\_2DAC4\_WFBYTE | X\_8P\_4DAC2\_WFBYTE  X\_ALT\_OFF  X\_ALT\_ON  X\_DACS\_0\_0\_0\_0  X\_DACS\_0\_0\_X\_X  X\_DACS\_0\_X\_X\_X  X\_DACS\_0N0\_0N0  X\_DACS\_0N0\_X\_X  X\_DACS\_1\_0\_1\_0  X\_DACS\_1\_0\_X\_X  X\_DACS\_1N1\_0N0  X\_DACS\_3\_2\_1\_0  X\_DACS\_OFF  X\_DACS\_X\_0\_X\_X  X\_DACS\_X\_X\_0\_0  X\_DACS\_X\_X\_0\_X | X\_DACS\_X\_X\_0N0  X\_DACS\_X\_X\_1\_0  X\_DACS\_X\_X\_X\_0  X\_DDS\_GOERTZEL\_SINC1  X\_DDS\_GOERTZEL\_SINC2  X\_IMM\_16X2\_1DAC2  X\_IMM\_16X2\_2DAC1  X\_IMM\_16X2\_LUT  X\_IMM\_1X32\_4DAC8  X\_IMM\_2X16\_2DAC8  X\_IMM\_2X16\_4DAC4  X\_IMM\_32X1\_1DAC1  X\_IMM\_32X1\_LUT  X\_IMM\_4X8\_1DAC8  X\_IMM\_4X8\_2DAC4  X\_IMM\_4X8\_4DAC2 | X\_IMM\_4X8\_LUT  X\_IMM\_8X4\_1DAC4  X\_IMM\_8X4\_2DAC2  X\_IMM\_8X4\_4DAC1  X\_IMM\_8X4\_LUT  X\_PINS\_OFF  X\_PINS\_ON  X\_RFBYTE\_1P\_1DAC1  X\_RFBYTE\_2P\_1DAC2  X\_RFBYTE\_2P\_2DAC1  X\_RFBYTE\_4P\_1DAC4  X\_RFBYTE\_4P\_2DAC2  X\_RFBYTE\_4P\_4DAC1  X\_RFBYTE\_8P\_1DAC8  X\_RFBYTE\_8P\_2DAC4  X\_RFBYTE\_8P\_4DAC2 | X\_RFBYTE\_LUMA8  X\_RFBYTE\_RGB8  X\_RFBYTE\_RGBI8  X\_RFLONG\_16X2\_LUT  X\_RFLONG\_32P\_4DAC8  X\_RFLONG\_32X1\_LUT  X\_RFLONG\_4X8\_LUT  X\_RFLONG\_8X4\_LUT  X\_RFLONG\_RGB24  X\_RFWORD\_16P\_2DAC8  X\_RFWORD\_16P\_4DAC4  X\_RFWORD\_RGB16  X\_WRITE\_OFF  X\_WRITE\_ON  XCONT  XINIT | XOR  XORC  XORO32  XORZ  XSTOP  XYPOL  XZERO  YELLOW  ZEROX  ZSTR  ZSTR\_ |
| --- | --- | --- | --- | --- | --- |

##### 

# CHANGE LOG

| **Date** | **Notes** |
| --- | --- |
| 6/7/2024 | First public release. |

# PARALLAX INCORPORATED

| **Parallax Inc.**  **599 Menlo Drive, Suite 100**  **Rocklin, CA 95765**  **USA** | **Office: +1 916-624-8333**  **Toll Free US: 888-512-1024** | [**sales@parallax.com**](mailto:sales@parallax.com)  [**support@parallax.com**](mailto:support@parallax.com) | [**www.parallax.com/p2**](http://www.parallax.com/p2)  [**forums.parallax.com**](https://forums.parallax.com) |
| --- | --- | --- | --- |
| **Purchase of the P2X8C4M64P does not include any license to emulate any other device nor to communicate via any specific proprietary protocol; P2X8C4M64P connectivity objects and code examples provided or referenced by Parallax, Inc. are NOT licensed and are provided for research and development purposes only; end users must seek permission to use licensed protocols for their applications and products from the protocol license holders.**  **Parallax Inc. makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Parallax Inc. assume any liability arising out of the application or use of any product, and specifically disclaims any and all liability, including without limitation consequential or incidental damages even if Parallax Inc. has been advised of the possibility of such damages.**  **Copyright © 2024 Parallax Inc. All rights are reserved. Parallax, the Parallax logo, the P2 logo, and Propeller are trademarks of Parallax Inc.** | | | |

# 