Parallax Propeller 2
Spin2 Language Documentation
2021-03-23
v35L

Document Status

Version	Date	Progress
	2020_02_06	Started document.
v34t	2020_07_15	DEBUG added, documentation up-to-date.
v34u	2020_07_19	DEBUG improved, documentation up-to-date.
v35	2020_11_18	DEBUG improved with anti-aliasing throughout, QSIN / QCOS added, documentation up-to-date.
v35e	2021_01_06	DEBUG_BAUD symbol added. Spin2 stack-locating bug fixed.
v35f	2021_01_29	DEBUG fixes. Was erring at 63 DEBUGs, now goes to 255. Was not always resetting the DEBUG.log file.
v35g	2021_02_13	DEBUG fixes. Line-clipping routine was causing floating-point exceptions and memory-access violations.
v35h	2021-02-15	● The first 16 LUT registers in the Spin2 interpreter were freed to allow for streamer 'imm-->LUT' usage. This is intended to support 1/2/4-bit video, via interrupt, within the same cog that the interpreter is running in. The inline-PASM limit went from $134 down to $124, in order to compensate. ● A new DEBUG_WINDOWS_OFF symbol was added to inhibit any DEBUG windows from opening after a download. DEBUG_BAUD can now be set to alter the baud rate that DEBUG uses with PNut.exe.
v35i	2021-02-20	● Added command-line DEBUG-only mode for presenting flash-programmed DEBUG data and displays. ● Fixed Floating-point error in SCOPE_XY.
v35j	2021-03-16	Fixed problem with DEBUG_BAUD <> 2_000_000 not working on some boards.
v35k	2021-03-19	Added DOWNLOAD_BAUD to existing DEBUG_BAUD for overriding default 2 Mbaud download and DEBUG.
v35L	2021-03-23	Added complete command-line interface to PNut.exe and included batch files for invoking PNut.exe and returning error status to STDOUT, STDERR, and ERRORLEVEL. See "Command Line options for PNut.exe".

OVERVIEW

The Spin2 language is designed to be very simple, but highly capable. Spin2 does not hide the underlying binary phenomena that makes computers work, but allows you to exploit it for effective programming. Assembly language is also supported in Spin2, as both in-line sequences and stand-alone programs.

A person with programming experience will be able to get a solid understanding of Spin2 in a very short amount of time. Learning Spin2 will pay dividends by allowing you to focus on your ideas, without having to navigate a myriad of typecasts and usage rules. Your brain will delight in staying busy, with compile+download+execute times of under 1 second.

In Spin2:

- There are no variable types, just three word sizes: BYTE (8 bits), WORD (16 bits), and LONG (32 bits), with bit fields supported for each.
- All math operations are performed at 32 bits and there are both signed and unsigned operators for where distinctions matter.
- Programs, called objects, can easily incorporate many other objects written by other authors with no foreknowledge of your project.
- Objects compile to compact, hardware-accelerated bytecode blocks which trigger short sequences of cog-resident interpreter code.
- Code is case-insensitive, so you can capitalize how you'd like.
- Symbolic names can be up to 32 characters in length.

In this documentation, all keywords are in UPPERCASE for clarity and anything in lowercase represents a user-defined symbolic name.

Program Structure

Spin2 programs are built from one or more objects. Objects are files which contain at least one public method, along with optional constants, child objects, variables, additional methods, and data. Objects are assembled together into a top-level object with an internal hierarchy of sub-objects. Each object instance, at run-time, gets its own set of variables, as defined by the object, to maintain its unique operating state.

Different parts of an object are declared within blocks, which all begin with 3-letter block identifiers.

The compiler can actually generate PASM-only programs, as well as Spin2+PASM programs, depending upon which blocks are present in the .spin2 file.

Block Identifier	Block Contents	Spin2+PASM Programs	PASM-only Programs
CON	Constant declarations (CON is the initial/default block type)	Permitted	Permitted
OBJ	Child-object instantiations	Permitted	Not Allowed
VAR	Variable declarations	Permitted	Not Allowed
PUB	Public method for use by the parent object and within this object	Required	Not Allowed
PRI	Private method for use within this object	Permitted	Not Allowed
DAT	Data declarations, including PASM code	Permitted	Required

Here are some minimal Spin2 and PASM-only programs. If you copy and paste these into PNut.exe, you can hit F10 to run them.

Minimal Spin2 Program

```
PUB MinimalSpin2Program()         'first PUB method executes

  REPEAT
    PINWRITE(63..56, GETRND())    'write a random pattern to P63..P56
    WAITMS(100)                   'wait 1/10th of a second, loop
```

Minimal PASM Program

```
DAT     ORG                       'start PASM at hub $00000 for cog $000

loop    DRVRND  #56 ADDPINS 7     'write a random pattern to P63..P56
        WAITX   ##clkfreq_/10     'wait 1/10th of a second, loop
        JMP     #loop
```

Here is a Spin2 program which contains every block type.

All-Block Spin2 Program

```
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 elapsed 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
```

A breakdown of each block type follows.

CON Blocks

CON blocks are used to define symbolic constants which can be used throughout the file.

- Symbolic constants resolve to 32-bit values.
- Symbolic constants can be assigned using '=' or by just expressing their names in an enumeration list.
- Symbolic constants can be referenced by every block within the file, including CON blocks.
- Symbolic constants can be referenced by the parent object's methods via 'objectname.constantname' syntax.
- If a decimal point is present, the value is encoded in IEEE-754 single-precision format.

CON

Direct Constant Assignments

```
CON  EnableFlow = 8                'single assignments
     DisableFlow = 4
     ColorBurstFreq = 3_579_545

     PWM_base = 8
     PWM_pins = PWM_base ADDPINS 7

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

     HalfPi = 1.5707963268         'single-precision float values
     QuarPi = HalfPi / 2.0

     j = ROUND(4000.0 / QuarPi)    'float to integer
```

CON

Enumerated Constant Assignments

```
CON  #0,a,b,c,d            'a=0, b=1, c=2, d=3    (start=0, step=1)
     #1,e,f,g,h            'e=1, f=2, g=3, h=4    (start=1, step=1)

     #4[2],i,j,k,l         'i=4, j=6, k=8, l=10   (start=4, step=2)
     #-1[-1],m,n,p         'm=-1, n=-2, p=-3      (start=-1, step=-1)

     #16                   'start=16, step=1
     q                     'q=16
     r[0]                  'r=17   ([0] is a step multiplier)
     s                     's=17
     t                     't=18
     u[2]                  'u=19   ([2] is a step multiplier)
     v                     'v=21
     w                     'w=22

CON  e0,e1,e2              'e0=0, e1=1, e2=2      (start=0, step=1)
                           '..enumeration is reset at each CON
```

OBJ Blocks

OBJ blocks are used to instantiate child objects into the current (parent) object. Child objects' methods can be executed and their constants can be referenced by the parent object at run time.

- Up to 32 different child objects can be incorporated into a parent object.
- Child objects can be instantiated singularly or in arrays of up to 255.
- Up to 1024 child objects are allowed per parent object.

OBJ

Child-Object Instantiations

```
OBJ  vga       : "VGA_Driver"     'instantiate "VGA_Driver.spin2" as "vga"

     mouse     : "USB_Mouse"      'instantiate "USB_Mouse.spin2" as "mouse"

     v[16]     : "VocalSynth"     'instantiate an array of 16 objects
                                  '..v[0] through v[15]
```

From within a parent-object method, a child-object method can be called by using the syntax:

```
object_name.method_name({any_parameters})
```

From within a parent-object method, a child-object constant can be referenced by using the syntax:

```
object_name.constant_name
```

VAR Blocks

VAR blocks are used to declare symbolic variables which can be utilized by all methods within the object.

- Variables can be longs (32 bits), words (16 bits), and bytes (8 bits).
- Variables can be declared as singles or arrays.
- Variables are packed in memory in the order they are declared, beginning at a long-aligned address.
- Variables are initialized to zero at run time.
- Each object's first 15 longs of variable memory are accessed via special bytecodes for improved efficiency.
- Each instance of an object will require one long, plus its declared amount of VAR space, plus 0..3 bytes to long-align for the next object's variable space.

VAR

Variable Declarations

```
VAR  CogNum                      'The default variable size is LONG (32 bits).
     CursorMode
     PosX                        'The first 15 longs have special bytecodes for faster/smaller code.
     Posy
     SendPtr                     'So, declare your most common variables first, as longs.

     BYTE StringChr              'byte variable (8 bits)
     BYTE StringBuff[64]         'byte variable array (64 bytes)
     BYTE a,b,c[1000],d          'comma-separated declarations

     WORD CurrentCycle           'word variable (16 bits)
     WORD Cycles[200]            'word variable array (200 words)
```

```
        WORD e,f[5],g,h[10]         'comma-separated declarations

        LONG Value                  'long variable
        LONG Values[15]             'long variable array (15 longs)
        LONG i[100],j,k,l           'comma-separated declarations

        ALIGNW                      'word-align to hub memory, advances variable pointer as necessary

        ALIGNL                      'long-align to hub memory, advances variable pointer as necessary
        BYTE Bitmap[640*480]        '..useful for making long-aligned buffers for FIFO-wrapping
```

PUB and PRI Blocks

PUB and PRI blocks are used to define public and private executable Spin2 methods.

- PUB methods are available to the parent object, as well as to the object they are defined in.
- PRI methods are available only to the object they are defined in.
- The first PUB method in an object is what executes when that object is run as the top-level object.
- Methods can have from 0 to 127 input parameters, all of which are single longs.
- Methods can have from 0 to 15 output results, all of which are single longs.
- Methods can have up to 64KB of local variables, which can be bytes, words, and longs (default), in both singles and arrays.
- Local variable size overrides (BYTE/WORD) apply only to the variable being declared, not subsequent variables.
- Results are initialized to zero on method entry, while local variables are undefined.
- Parameters, then results, and then local variables are packed into stack memory in the order they are declared.
- In-line PASM code can access the first 16 longs of parameters...results...locals via registers with the same symbolic names.

PUB/PRI syntax is as follows:

PUB/PRI methodname({parameter{,...}}) {: result{,...}} {| {ALIGNW/ALIGNL} {BYTE/WORD/LONG} localvar{[count]}{,...}}

PUB / PRI Declarations (method code would go below each declaration)	Input Parameters (longs)	Output Results (longs)	Local Variables (longs, words, bytes)
PUB go()	0	0	0
PUB SetupADC(pins)	1	0	0
PUB StartTx(pin, baud) : Okay	2	1	0
PRI RotateXY(X, Y, Angle) : NewX, NewY | p,q,r	3	2	3 longs
PRI Shuffle() | i, j	0	0	2 longs
PRI FFT1024(DataPtr) | a, b, x[1024], y[1024]	1	0	1+1+1024+1024 longs
PRI ReMix() : Length, SampleRate | WORD Buff[20000], k	0	2	20000 words + 1 long
PRI StrCheck(StrPtrA, StrPtrB) : Pass | i, BYTE Str[64]	2	1	1 long + 64 bytes

DAT Blocks

DAT blocks are used to express data and PASM code.

- Data are packed in memory in the order they are declared, beginning at a long-aligned address.
- Data are expressed using the following syntax: {symbolname} **BYTE/WORD/LONG data{[count]} {,data...}**
- Symbols that precede data and PASM instructions resolve to addresses
  - In Spin2+PASM programs, hub addresses are relative to the start of the object and can be referenced as follows:
    - 'SymbolName' will return the data at the symbol, in accordance with its size (byte/word/long).
    - '@SymbolName' will return the address of the data.
    - '@@SymbolName' will convert an '@Symbol' in the data to an absolute address (see "DAT Data Pointers")
  - In PASM-only programs, hub addresses are absolute.

DAT
Symbols and Data

```
DAT                                          'symbols without data take the size of the previous declaration

HexChrs     BYTE    "0123456789ABCDEF"       'HexChrs is a byte symbol that points to "0"
symbol0                                      'symbol0 is a byte symbol that points after "F"

Pattern     WORD    $CCCC,$3333,$AAAA,$5555  'Pattern is word symbol that points to $CCCC
symbol1                                      'symbol1 is a word symbol that points after $55555

Billions    LONG    1_000_000_000            'Billions is a long symbol that points to 1_000_000_000
symbol2                                      'symbol2 is a long symbol that points after 1_000_000_000

DoNothing   NOP                              'DoNothing is a long symbol that points to a NOP instruction
symbol3                                      'symbol3 is a long symbol that points after the NOP instruction

symbol4     BYTE                             'symbol4 is a byte symbol that points to $78
symbol5     WORD                             'symbol5 is a word symbol that points to $5678
symbol6     LONG                             'symbol6 is a long symbol that points to $12345678

            LONG    $12345678                'long value $12345678

            BYTE    100[64]                  '64 bytes of value 100

            BYTE    10, WORD 500, LONG $FC000 'BYTE/WORD/LONG overrides allowed for single values
            BYTE    FVAR 99, FVARS -99       'FVAR/FVARS overrides allowed, can be read via RFVAR/RFVARS

FileDat     FILE    "Filename"               'include binary file, FileDat is a byte symbol that points to file

            ALIGNW                           'word-align to hub by emitting a zero byte, if necessary
            ALIGNL                           'long-align to hub by emitting 1 to 3 zero bytes, if necessary
```

DAT
Data Pointers

```
DAT

Str0          BYTE    "Monkeys",0     'strings with symbols
Str1          BYTE    "Gorillas",0
Str2          BYTE    "Chimpanzees",0
Str3          BYTE    "Humanzees",0

StrList       WORD    @Str0           'in Spin2, these are offsets of strings relative to start of object
              WORD    @Str1           'in Spin2, @@StrList[i] will return address of Str0..Str3 for i = 0..3
              WORD    @Str2           'in PASM-only programs, these are absolute addresses of strings
              WORD    @Str3           '(use of WORD supposes offsets/addresses are under 64KB)
```

DAT
Cog-exec

```
DAT           ORG                     'begin a cog-exec program (no symbol allowed before ORG)
                                      'COGINIT(16, @IncPins, 0) will launch this program in a free cog
IncPins       MOV     DIRA,#$FF       'to Spin2 code, IncPins is the 'MOV' instruction (long)
Loop          ADD     OUTA,#1         'to Spin2 code, @IncPins is the hub address of the 'MOV' instruction
              JMP     #Loop           'to PASM code, Loop is the cog address ($001) of the 'ADD' instruction

              ORG                     'set cog-exec mode, cog address = $000, cog limit = $1F8 (reg, both defaults)
              ORG     $100            'set cog-exec mode, cog address = $100, cog limit = $1F8 (reg, default limit)
              ORG     $120,$140       'set cog-exec mode, cog address = $120, cog limit = $140 (reg)
              ORG     $200            'set cog-exec mode, cog address = $200, cog limit = $400 (LUT, default limit)
              ORG     $300,$380       'set cog-exec mode, cog address = $300, cog limit = $380 (LUT)

              ADD     reg,#1          'in cog-exec mode, instructions force alignment to cog/LUT registers

              ORGF    $040            'fill to cog address $040 with zeros (no symbol allowed before ORGF)

              FIT     $020            'test to make sure cog address has not exceeded $020

x             RES     1               'reserve 1 register, advance cog address by 1, don't advance hub address
y             RES     1               'reserve 1 register, advance cog address by 1, don't advance hub address
z             RES     1               'reserve 1 register, advance cog address by 1, don't advance hub address
buff          RES     16              'reserve 16 registers, advance cog address by 16, don't advance hub address
```

DAT
Hub-exec

```
DAT           ORGH                    'begin a hub-exec program (no symbol allowed before ORGH)
                                      'COGINIT(32+16, @IncPins, 0) will launch this program in a free cog
IncPins       MOV     DIRA,#$FF       'In Spin2, IncPins is the 'MOV' instruction (long)
Loop          ADD     OUTA,#1         'In Spin2, @IncPins is the hub address of the 'MOV' instruction
              JMP     #Loop           'In PASM, Loop is the hub address ($00404) of the 'ADD' instruction

              ORGH                    'set hub-exec mode, hub origin = $00400, origin limit = $100000 (both defaults)
              ORGH    $1000           'set hub-exec mode, hub origin = $01000, origin limit = $100000 (default limit)
              ORGH    $FC000,$FC800   'set hub-exec mode, hub origin = $FC000, origin limit = $FC800

              FIT     $2000           'test to make sure hub address has not exceeded $2000
```

There are some differences between Spin2+PASM programs and PASM-only programs, when it comes to hub-exec code:

Spin2+PASM Programs	• Hub-exec code must use relative addressing, since it is not located at its place of origin.
	• The LOC instruction can be used to get addresses of data assets within relative hub-exec code.
	• ORGH must specify at least $400, so that pure hub-exec code will be assembled.
	• The default ORGH address of $400 is always appropriate, unless you are writing code which will be moved to its actual ORGH address at runtime, so that it can use absolute addressing.
	
	`DAT    ORGH            'set hub-exec mode and set origin to $400`
	`       ORGH    $FC000  'set hub-exec mode and set origin to $FC000`

PASM-Only Programs	• Hub-exec code may use absolute and relative addressing, since origin always matches hub address.
	• ORGH fills hub memory with zeros, up to the specified address.
	
	`DAT    ORGH            'set hub-exec mode at current hub address`
	`       ORGH    $400    'set hub-exec mode and fill hub memory with zeros to $400`

Spin2 Language

Constants

Constants resolve to 32-bit values and can be expressed as follows:

Constants	Examples	Descriptions

Decimal	`1`
	`-150`
	`3_000_000`
		• Decimal values use digits '0'..'9'
		• Underscores '_' are allowed after the first digit for placeholding

Hexadecimal	`$1B`
	`$AA55`
	`$FFFF_FFFF`
		• Hex values start with '$' and use digits '0'..'9' and 'A'..'F'
		• Underscores '_' are allowed after the first digit for placeholding

Double Binary	`%%21`
	`%%01_23`
	`%%3333_2222_1111_0000`
		• Double binary values start with '%%' and use digits '0'..'3'
		• Underscores '_' are allowed after the first digit for placeholding

Binary	`%0110`
	`%1_1111_1000`
	`%0001_0010_0011_0100`
		• Binary values start with '%' and use digits '0' and '1'
		• Underscores '_' are allowed after the first digit for placeholding

Float	`-1.0`
	`1_250_000.0`
	`1e9`
	`-1.23456e-7`
		• Float values use digits '0'..'9' and have a '.' and/or 'e' in them
		• Floats are encoded in IEEE-754 single-precision 32-bit format
		• Underscores '_' are allowed after the first digit for placeholding
		• Floats are not part of Spin2, but a library can offer floating-point functions

Character	`"H"`	• A single character in quotes resolves to a 7-bit ASCII value

Variables

In Spin2, there are both user-defined and permanent variables. The user-defined variable sources are listed below and the permanent variables are shown in the table.

- VAR variables (hub)
- PUB/PRI parameters, return values, and local variables (hub)
- DAT symbols (hub)
- Cog registers

Variables (all LONG)	Variable Name	Address or Offset	Description	Useful in Spin2	Useful in Spin2-PASM	Useful in PASM-Only
Hub Locations	**CLKMODE** **CLKFREQ**	**$00040** **$00044**	Clock mode value Clock frequency value	Yes Yes	Yes Yes	No No
Hub VAR	**VARBASE**	+0	Object base pointer, @VARBASE is VAR base, used by method-pointer calls	Maybe	No	No
Cog Registers	**PR0** **PR1** **PR2** **PR3** **PR4** **PR5** **PR6** **PR7**	**$1D8** **$1D9** **$1DA** **$1DB** **$1DC** **$1DD** **$1DE** **$1DF**	Spin2 <-> PASM communication	Yes Yes Yes Yes Yes Yes Yes Yes	Yes Yes Yes Yes Yes Yes Yes Yes	No No No No No No No No
	IJMP3 **IRET3** **IJMP2** **IRET2** **IJMP1** **IRET1**	**$1F0** **$1F1** **$1F2** **$1F3** **$1F4** **$1F5**	Interrupt JMP's and RET's	No No No No No No	Yes Yes Yes Yes Yes Yes	Yes Yes Yes Yes Yes Yes
	PA **PB** **PTRA** **PTRB**	**$1F6** **$1F7** **$1F8** **$1F9**	Pointer registers Data pointer passed from COGINIT Code pointer passed from COGINIT	No No No No	Yes Yes Yes Yes	Yes Yes Yes Yes
	DIRA **DIRB** **OUTA** **OUTB** **INA** **INB**	**$1FA** **$1FB** **$1FC** **$1FD** **$1FE** **$1FF**	Output enables for P31..P0 Output enables for P63..P32 Output states for P31..P0 Output states for P63..P32 Input states from P31..P0 Input states from P63..P32	Yes Yes Yes Yes Yes Yes	Yes Yes Yes Yes Yes Yes	Yes Yes Yes Yes Yes Yes

In Spin2, all variables can be indexed and accessed as bitfields. Additionally, symbolic hub variables can have BYTE/WORD/LONG size overrides:

Variable Usage	Example	Description
Plain	**AnyVar** **HubVar.WORD** **BYTE[address]** **REG[register]**	Hub or permanent register variable Hub variable with BYTE/WORD/LONG size override Hub BYTE/WORD/LONG by address Register, 'register' may be symbol declared in ORG section
With Index	**AnyVar[index]** **HubVar.BYTE[index]** **LONG[address][index]** **REG[register][index]**	Hub or permanent register variable with index Hub variable with size override and index Hub BYTE/WORD/LONG by address with index Register with index
With Bitfield	**AnyVar.[bitfield]** **HubVar.LONG.[bitfield]** **WORD[address].[bitfield]** **REG[register].[bitfield]**	Hub or permanent register variable with bitfield Hub variable with size override and bitfield Hub BYTE/WORD/LONG by address with bitfield Register with bitfield
With Index and Bitfield	**AnyVar[index].[bitfield]** **HubVar.BYTE[index].[bitfield]** **LONG[address][index].[bitfield]** **REG[register][index].[bitfield]**	Hub or permanent register variable with index and bitfield Hub variable with size override, index, and bitfield Hub BYTE/WORD/LONG by address with index and bitfield Register with index and bitfield

A bitfield is a 10-bit value which contains a base-bit number in bits 4..0 and an additional-bits number in bits 9..5. Bitfields can be defined in a few different ways:

Bitfield	Bit Range	Details
.[%00000_00000]	**0**	0 additional bits above the base bit 0, a single-bit bitfield
.[%00000_11111]	**31**	0 additional bits above the base bit 31, a single-bit bitfield

`.[%00010_01111]`	`17..15`	2 additional bits above the base bit 15, a three-bit bitfield
`.[%11110_00000]`	`30..0`	30 additional bits above the base bit 0, a 31-bit bitfield
`.[%11111_10000]`	`15..0, 31..16`	31 additional bits above the base bit 16, wraps around, a 32-bit bitfield
`.[%00001_11111]`	`0, 31`	1 additional bit above the base bit 31, wraps around, a 2-bit bitfield
`.[23]`	`23`	Just the base bit, adds no extra bits
`.[31..20]`	`31..20`	'Top..Bottom' syntax allowed within '`.[]`', wraps if Top < Bottom
`.[5 ADDBITS 7]`	`12..5`	ADDBITS can be used to compute the bitfield
`.[BitfieldCon]`	`13..9`	`CON BitfieldCon = 9 ADDBITS 4            'BitfieldCon useful in PASM, too`
`.[BitfieldVar]`	`?`	`BitfieldVar := BaseBit ADDBITS ExtraBits  'wraps if BaseBit + ExtraBits > 31`

In addition to bitfields, there are also pinfields, which are used to select a range of I/O pins within the same 32-pin block (P63..P32 or P31..P0). Pinfields are 11-bit values which contain a base-pin number in bits 5..0 and an additional-pins number in bits 10..6. Pinfields are used by instructions which interface to pins.

Pinfield	Pin Range	Details
`PINLOW(%00000_000000)`	`0`	0 additional pins above the base pin 0, a single-pin pinfield
`PINLOW(%00000_111111)`	`63`	0 additional pins above the base pin 63, a single-pin pinfield
`PINLOW(%00011_100000)`	`35..32`	3 additional pins above the base pin 32, a four-pin pinfield
`PINLOW(%11111_001000)`	`7..0, 31..8`	31 additional pins above the base pin 8, wraps around, a 32-pin pinfield
`PINLOW(19)`	`19`	Just the base pin, adds no extra pins
`PINLOW(49..40)`	`49..40`	'Top..Bottom' syntax allowed within '`.[]`', wraps if Top < Bottom
`PINLOW(11 ADDPINS 4)`	`15..11`	ADDPINS can be used to compute the pinfield
`PINLOW(PinfieldCon)`	`53..50`	`CON PinfieldCon = 50 ADDPINS 3           'PinfieldCon useful in PASM, too`
`PINLOW(PinfieldVar)`	`?`	`PinfieldVar := BasePin ADDPINS ExtraPins  'wraps if BasePin + ExtraPins > 31`

Expressions

- Run-time expressions can incorporate constants, variables, and methods' return values
- Compile-time expressions can use only constants.
- All expressions can use operators.

Here are some examples of expressions:

Expression	Details
`BYTE[i++]`	Byte pointed to by 'i', post-increment 'i'
`(digit := value / place // 10) OR place == 1`	Boolean with buried 'digit' assignment
`place /= 10`	Divide 'place' by 10
`"0" + digit`	Get 'digit' character
`PINREAD(17..12)`	Read pins 17..12

Operators

Below is a table of all the operators available for use in Spin2 methods. Compile-time expressions can use the unary, binary, ternary and float operators.

Var-Prefix Operators	Term (method only)	Priority (term)	Assign (method only)	Priority (assign)	Description	Float Exp
`++ (pre)`	`++var`	1	`++var`	1	Pre-increment	
`-- (pre)`	`--var`	1	`--var`	1	Pre-decrement	
`?? (pre)`	`??var`	1	`??var`	1	Iterate long per XORO32, return pseudo-random	
Var-Postfix Operators	**Term (method only)**	**Priority (term)**	**Assign (method only)**	**Priority (assign)**	**Description**	**Float Exp**
`(post) ++`	`var++`	1	`var++`	1	Post-increment	
`(post) --`	`var--`	1	`var--`	1	Post-decrement	
`(post) !!`	`var!!`	1	`var!!`	1	Post-logical NOT (0 → -1, non-0 → 0)	
`(post) !`	`var!`	1	`var!`	1	Post-bitwise NOT	
`(post) \`	`var\x`	1	`var\x`	1	Post-assign x	
`(post) ~`	`var~`	1	`var~`	1	Post-clear all bits	
`(post) ~~`	`var~~`	1	`var~~`	1	Post-set all bits	
Address Operators	**Term (method only)**	**Priority (term)**			**Description**	**Float Exp**
`@`	`@symbol`	1			Hub address of VAR/PUB/PRI variable or DAT symbol	
`@`	`@method`	1			Pointer to method, may be @object[[i]].method	
`@@`	`@@x`	1			Hub address of object + x, 'DAT x long @dat_symbol'	
`#`	`#reg_symbol`	1			Register address of cog/LUT DAT symbol	
Unary Operators	**Term**	**Priority (term)**	**Assign (method only)**	**Priority (assign)**	**Description**	**Float Exp**
`!!, NOT`	`!!x`	12	`!!= var`	1	Logical NOT (0 → -1, non-0 → 0)	
`!`	`!x`	2	`!= var`	1	Bitwise NOT (1's complement)	
`-`	`-x`	2	`-= var`	1	Negate (2's complement)	✔
`ABS`	`ABS x`	2	`ABS= var`	1	Absolute value	✔
`ENCOD`	`ENCOD x`	2	`ENCOD= var`	1	Encode MSB, 0..31	
`DECOD`	`DECOD x`	2	`DECOD= var`	1	Decode, 1 << (x & $1F)	
`BMASK`	`BMASK x`	2	`BMASK= var`	1	Bitmask, (2 << (x & $1F)) - 1	
`ONES`	`ONES x`	2	`ONES= var`	1	Sum all '1' bits, 0..32	
`SQRT`	`SQRT x`	2	`SQRT= var`	1	Square root of unsigned value	
`QLOG`	`QLOG x`	2	`QLOG= var`	1	Unsigned value to logarithm {5'whole, 27'fraction}	
`QEXP`	`QEXP x`	2	`QEXP= var`	1	Logarithm to unsigned value	
Binary Operators	**Term**	**Priority (term)**	**Assign (method only)**	**Priority (assign)**	**Description**	**Float Exp**
`>>`	`x >> y`	3	`var >>= y`	17	Shift x right by y bits, insert 0's	
`<<`	`x << y`	3	`var <<= y`	17	Shift x left by y bits, insert 0's	
`SAR`	`x SAR y`	3	`var SAR= y`	17	Shift x right by y bits, insert MSB's	
`ROR`	`x ROR y`	3	`var ROR= y`	17	Rotate x right by y bits	
`ROL`	`x ROL y`	3	`var ROL= y`	17	Rotate x left by y bits	
`REV`	`x REV y`	3	`var REV= y`	17	Reverse y LSBs of x and zero-extend	
`ZEROX`	`x ZEROX y`	3	`var ZEROX= y`	17	Zero-extend above bit y	
`SIGNX`	`x SIGNX y`	3	`var SIGNX= y`	17	Sign-extend from bit y	
`&`	`x & y`	4	`var &= y`	17	Bitwise AND	
`^`	`x ^ y`	5	`var ^= y`	17	Bitwise XOR	
`|`	`x | y`	6	`var |= y`	17	Bitwise OR	
`*`	`x * y`	7	`var *= y`	17	Signed multiply	✔
`/`	`x / y`	7	`var /= y`	17	Signed divide, return quotient	✔
`+/`	`x +/ y`	7	`var +/= y`	17	Unsigned divide, return quotient	
`//`	`x // 7`	7	`var //= y`	17	Signed divide, return remainder	
`+//`	`x +// y`	7	`var +//= y`	17	Unsigned divide, return remainder	
`SCA`	`x SCA y`	7	`var SCA= y`	17	Unsigned scale, (x * y) >> 32	
`SCAS`	`x SCAS y`	7	`var SCAS= y`	17	Signed scale, (x * y) >> 30	
`FRAC`	`x FRAC y`	7	`var FRAC= y`	17	Unsigned fraction, (x << 32) / y	
`+`	`x + y`	8	`VAR += y`	17	Add	✔
`-`	`x - y`	8	`var -= y`	17	Subtract	✔
`#>`	`x #> y`	9	`var #>= y`	17	Force x => y, signed	✔
`<#`	`x <# y`	9	`var <#= y`	17	Force x <= y, signed	✔
`ADDBITS`	`x ADDBITS y`	10	`var ADDBITS= y`	17	Make bitfield, (x & $1F) | (y & $1F) << 5	
`ADDPINS`	`x ADDPINS y`	10	`var ADDPINS= y`	17	Make pinfield, (x & $3F) | (y & $1F) << 6	
`<`	`x < y`	11			Signed less than (returns 0 or -1)	✔
`+<`	`x +< y`	11			Unsigned less than (returns 0 or -1)	
`<=`	`x <= y`	11			Signed less than or equal (returns 0 or -1)	✔
`+<=`	`x +<= y`	11			Unsigned less than or equal (returns 0 or -1)	

`==`	`x == y`	11			Equal (returns 0 or -1)	✔
`<>`	`x <> y`	11			Not equal (returns 0 or -1)	✔
`>=`	`x >= y`	11			Signed greater than or equal (returns 0 or -1)	✔
`+>=`	`x +>= y`	11			Unsigned greater than or equal (returns 0 or -1)	
`>`	`x > y`	11			Signed greater than (returns 0 or -1)	✔
`+>`	`x +> y`	11			Unsigned greater than (returns 0 or -1)	
`<=>`	`x <=> y`	11			Signed comparison (<,=,> returns -1,0,1)	✔
`&&, AND`	`x && y`	13	`var &&= y`	17	Logical AND (x <> 0 AND y <> 0, returns 0 or -1)	
`^^, XOR`	`x ^^ y`	14	`var ^^= y`	17	Logical XOR (x <> 0 XOR y <> 0, returns 0 or -1)	
`||, OR`	`x || y`	15	`var ||= y`	17	Logical OR (x <> 0 OR y <> 0, returns 0 or -1)	
Ternary Operator	**Term**	**Priority (term)**			**Description**	**Float Exp**
`? :`	`x ? y : z`	16			If x <> 0 then choose y, else choose z	
Assign Operator			**Assign (method only)**	**Priority (assign)**	**Description**	**Float Exp**
`:=`			`var := x` `v1,v2 := x,y`	17	Set var to x Set v1 to x, set v2 to y, etc. ('_' on left = ignore)	
Equate Operator			**Assign (CON block only)**	**Priority (equate)**	**Description**	**Float Exp**
`=`			`symbol = x`	17	Set symbol to x in CON block	
Float Operators	**Term (constant only)**				**Description**	**Float Exp**
`FLOAT()`	`FLOAT(x)`				Convert integer x to float	✔
`ROUND()`	`ROUND(x)`				Convert float x to rounded integer	✔
`TRUNC()`	`TRUNC(x)`				Convert float x to truncated integer	✔

Built-in Methods

Hub Methods	Details
`HUBSET(Value)`	Execute HUBSET instruction using Value
`CLKSET(NewCLKMODE, NewCLKFREQ)`	Safely establish new clock settings, updates CLKMODE and CLKFREQ
`COGSPIN(CogNum, Method({Pars}), StkAddr)`	Start Spin2 method in a cog, returns cog's ID if used as an expression element, -1 = no cog free
`COGINIT(CogNum, PASMaddr, PTRAvalue)`	Start PASM code in a cog, returns cog's ID if used as an expression element, -1 = no cog free
`COGSTOP(CogNum)`	Stop cog CogNum
`COGID() : CogNum`	Get this cog's ID
`COGCHK(CogNum) : Running`	Check if cog CogNum is running, returns -1 if running or 0 if not
`LOCKNEW() : LockNum`	Check out a new LOCK from inventory, LockNum = 0..15 if successful or < 0 if no LOCK available
`LOCKRET(LockNum)`	Return a certain LOCK to inventory
`LOCKTRY(LockNum) : LockState`	Try to capture a certain LOCK, LockState = -1 if successful or 0 if another cog has captured the LOCK
`LOCKREL(LockNum)`	Release a certain LOCK
`LOCKCHK(LockNum) : LockState`	Check a certain LOCK's state, LockState[31] = captured, LockState[3:0] = current or last owner cog
`COGATN(CogMask)`	Strobe ATN input(s) of cog(s) according to 16-bit CogMask
`POLLATN() : AtnFlag`	Check if this cog has received an ATN strobe, AtnFlag = -1 if ATN strobed or 0 if not strobed
`WAITATN()`	Wait for this cog to receive an ATN strobe

Pin Methods	Details
`PINW | PINWRITE(PinField, Data)`	Drive PinField pin(s) with Data
`PINL | PINLOW(PinField)`	Drive PinField pin(s) low
`PINH | PINHIGH(PinField)`	Drive PinField pin(s) high
`PINT | PINTOGGLE(PinField)`	Drive and toggle PinField pin(s)
`PINF | PINFLOAT(PinField)`	Float PinField pin(s)
`PINR | PINREAD(PinField) : PinStates`	Read PinField pin(s)
`PINSTART(PinField, Mode, Xval, Yval)`	Start PinField smart pin(s): DIR=0, then WRPIN=Mode, WXPIN=Xval, WYPIN=Yval, then DIR=1
`PINCLEAR(PinField)`	Clear PinField smart pin(s): DIR=0, then WRPIN=0
`WRPIN(PinField, Data)`	Write 'mode' register(s) of PinField smart pin(s) with Data
`WXPIN(PinField, Data)`	Write 'X' register(s) of PinField smart pin(s) with Data
`WYPIN(PinField, Data)`	Write 'Y' register(s) of PinField smart pin(s) with Data
`AKPIN(PinField)`	Acknowledge PinField smart pin(s)
`RDPIN(Pin) : Zval`	Read Pin smart pin and acknowledge, Zval[31] = C flag from RDPIN, other bits are RDPIN data
`RQPIN(Pin) : Zval`	Read Pin smart pin without acknowledge, Zval[31] = C flag from RQPIN, other bits are RQPIN data

Timing Methods	Details
`GETCT() : Count`	Get 32-bit system counter
`POLLCT(Tick) : Past`	Check if system counter has gone past 'Tick', returns -1 if past or 0 if not past
`WAITCT(Tick)`	Wait for system counter to get past 'Tick'
`WAITUS(Microseconds)`	Wait Microseconds, uses CLKFREQ
`WAITMS(Milliseconds)`	Wait Milliseconds, uses CLKFREQ
`GETSEC() : Seconds`	Get seconds since booting, uses 64-bit system counter and CLKFREQ, rolls over every 136 years.
`GETMS() : Milliseconds`	Get milliseconds since booting, uses 64-bit system counter and CLKFREQ, rolls over every 49.7 days.

PASM interfacing	Details
`CALL(RegOrHubAddr)`	CALL PASM code at Addr, PASM code should avoid registers $130..$1D7 and LUT
`REGEXEC(HubAddr)`	Load a self-defined chunk of PASM code at HubAddr into registers and CALL it. See REGEXEC description.
`REGLOAD(HubAddr)`	Load a self-defined chunk of PASM code or data at HubAddr into registers. See REGLOAD description.

Math Methods	Details
`ROTXY(x, y, angle32bit) : rotx, roty`	Rotate (x,y) by angle32bit and return rotated (x,y)
`POLXY(length, angle32bit) : x, y`	Convert (length,angle32bit) to (x,y)

`XYPOL(x, y) : length, angle32bit`	Convert (x,y) to (length,angle32bit)
`QSIN(length, angle, twopi) : y`	Rotate (length,0) by (angle / twopi) * 2Pi and return y. Use 0 for twopi = $1_0000_0000. Twopi is unsigned.
`QCOS(length, angle, twopi) : x`	Rotate (length,0) by (angle / twopi) * 2Pi and return x. Use 0 for twopi = $1_0000_0000. Twopi is unsigned.
`MULDIV64(mult1,mult2,divisor) : quotient`	Divide the 64-bit product of 'mult1' and 'mult2' by 'divisor', return quotient (unsigned operation)
`GETRND() : Rnd`	Get random long (from xoroshiro128** PRNG, seeded on boot with thermal noise from ADC)

Memory Methods	Details
`GETREGS(HubAddr, CogAddr, Count)`	Move Count registers at CogAddr to longs at HubAddr
`SETREGS(HubAddr, CogAddr, Count)`	Move Count longs at HubAddr to registers at CogAddr
`BYTEMOVE(Dest, Source, Count)`	Move Count bytes from Source to Dest
`WORDMOVE(Dest, Source, Count)`	Move Count words from Source to Dest
`LONGMOVE(Dest, Source, Count)`	Move Count longs from Source to Dest
`BYTEFILL(Dest, Value, Count)`	Fill Count bytes at Dest with Value
`WORDFILL(Dest, Value, Count)`	Fill Count words at Dest with Value
`LONGFILL(Dest, Value, Count)`	Fill Count longs at Dest with Value

String Methods	Details
`STRSIZE(Addr) : Size`	Count bytes in zero-terminated string at Addr, return string size, not including zero terminator
`STRCOMP(AddrA,AddrB) : Match`	Compare zero-terminated strings at AddrA and AddrB, return -1 if match or 0 if mismatch
`STRING("Text",9) : StringAddress`	Compose a zero-terminated string (quoted characters and values 1..255 allowed), return address of string

Index ↔️ Value Methods	Details
`LOOKUP(Index: v1, v2..v3, etc) : Value`	Lookup value (values and ranges allowed) using 1-based index, return value (0 if index out of range)
`LOOKUPZ(Index: v1, v2..v3, etc) : Value`	Lookup value (values and ranges allowed) using 0-based index, return value (0 if index out of range)
`LOOKDOWN(Value: v1, v2..v3, etc) : Index`	Determine 1-based index of matching value (values and ranges allowed), return index (0 if no match)
`LOOKDOWNZ(Value: v1, v2..v3, etc) : Index`	Determine 0-based index of matching value (values and ranges allowed), return index (0 if no match)

USING METHODS

Methods that return single results can be used as terms in expressions:

```
  x := GETRND() +// 100    'Get a random number between 0 and 99

  BYTEMOVE(ToStr, FromStr, STRSIZE(FromStr) + 1)
```

Methods which return multiple results (like POLXY) can be used to supply multiple parameters to other methods:

```
  x,y := SumPoints(POLXY(rho1,theta1), POLXY(rho2,theta2))
```

…**where**…

```
SumPoints(x1, y1, x2, y2) : x, y
  RETURN x1+x2, y1+y2
```

Multiple method results can be assigned to variables or ignored by using an underscore in lieu of a variable name::

```
  x,y := ROTXY(xin,yin,theta)     'use both the x and y results
  _,y := ROTXY(xin,yin,theta)     'use only the y result
  x,_ := ROTXY(xin,yin,theta)     'use only the x result
```

User-defined methods which return one or more results can also be used as instructions, where the return values are ignored. However, built-in methods such as STRSIZE, which return results, must be used as expression terms.

ABORT

Spin2 has an "abort" mechanism for instantly returning, from any depth of nested method calls, back to a base caller which used '\' before the method name. A single return value can be conveyed from the abort point back to the base caller:

```
PRI Sub1() : Error    'Sub1 calls Sub2 with an ABORT trap
  Error := \Sub2()    '\ means call method and trap any ABORT
  \Sub2()             'in this case, the ABORT value is ignored

PRI Sub2()            'Sub2 calls Sub3
  Sub3()              'Sub3 never returns here due to the ABORT
  PINHIGH(0)          'PINHIGH never executes

PRI Sub3()            'Sub3 ABORTs, returning to Sub1 with ErrorCode
  ABORT ErrorCode     'ABORT and return ErrorCode
  PINLOW(0)           'PINLOW never executes
```

Regardless of how many return values a particular method may have, when that method is called with a preceding "\", there will be only one return value, which may be ignored.

If no value is specified after ABORT, then zero will be returned.

If a method is called with a preceding "\", but no ABORT occurs, then zero will be returned.

If an ABORT executes without a "\" trap somewhere in the call chain, the cog returns past the top-level method and executes COGSTOP(COGID), shutting itself down.

The abort mechanism is intended as a means to return from a deeply nested subroutine where some error situation has developed, but it can be used for any purpose. Basically, it's a way to return to a base caller without having to check for a condition to do so at every level of the call chain. It returns all the way back to the caller with the "\" abort trap, carrying the ABORT value. You can compose hierarchical levels of "\" abort traps and ABORT points.

METHOD POINTERS

Method pointers are LONG values which point to a method and are then used to call that method indirectly.

To establish a method pointer, you can assign a long variable using "@" before the method name. Note that there are no parentheses after the method name:

```
LongVar := @SomeMethod                          'a method within the current object
LongVar := @SomeObject.SomeMethod               'a method within a child object
LongVar := @SomeOject[index].SomeMethod         'a method within an indexed child object
```

Method pointers can be generated on-the-fly and passed as parameters:

```
SetUpIO(@InMethod,@OutMethod)
```

Method pointers are then used in the following ways to call methods:

```
LongVar()                                       'no parameters and no return values
LongVar(Par1, Par2)                             'two parameters and no return values
Var := LongVar():1                              'no parameters and one return value
Var1,Var2 := LongVar(Par1):2                    'one parameters and two return values
Var1,Var2 := POLXY(LongVar(Par1,Par2,Par3):2)   'three parameters and two return values
```

There is no compile-time awareness of how many parameters the method pointed to actually has. You need to code your method pointer usage such that you supply the proper number of parameters and specify the proper number of return values after a ":", so that there is agreement with the method pointed to.

Method pointers can be passed through object hierarchies to enable direct calling of any method from anywhere. They can also be used to dynamically point to different methods which have the same numbers of parameters and return values.

How Method Pointers Work

An @method expression generates a 32-bit value which has two bit fields:

[31..20] = Index of the method, relative to the method's object base. The index of the first method will be twice the number of objects instantiated

[19..0] = Address of the method's VAR base. The method's VAR base, in turn, contains the address of the method's object base.

By putting the method's index and VAR base address together into the 32-bit value, and having the VAR base contain the method's object base address, a complete method pointer is established in a single long, which can be treated as any other variable.

To accommodate method pointers, each object instance reserves the first long of its VAR space for the object base address. When an @method expression executes, that first long is written with the object's base address.

SEND

SEND is a special method pointer which is inherited from the calling method and, in turn, conveyed to all called methods. It's purpose is to provide an efficient output mechanism for data.

SEND can be assigned like a method pointer, but it must point to a method which takes one parameter and has no return values:

```
SEND := @OutMethod
```

When used as a method, SEND will pass all parameters, including any return values from called methods, to the method SEND points to:

```
SEND("Hello! ", GetDigit()+"0", 13)
```

Any methods called within the SEND parameters will inherit the SEND pointer, so that they can do SEND methods, too:

```
PUB Go()
  SEND := @SetLED
  REPEAT
    SEND(Flash(),$01,$02,$04,$08,$10,$20,$40,$80)

PRI Flash() : x
  REPEAT 2
    SEND($00,$FF,$00)
  RETURN $AA

PRI SetLED(x)
  PINWRITE(56 ADDPINS 7, !x)
  WAITMS(125)
```

In the above example, the following values are output in repeating sequence: $00, $FF, $00, $00, $FF, $00, $AA, $01, $02, $04, $08, $10, $20, $40, $80 (but inverted for LEDs)

Though a called method inherits the current SEND pointer, it may change it for its own purposes. Upon return from that method, the SEND pointer will be back to what it was before the method was called. So, the SEND pointer value is propagated in method calls, but not in method returns.

RECV

RECV, like SEND, is a special method pointer which is inherited from the calling method and, in turn, conveyed to all called methods. It's purpose is to provide an efficient input mechanism for data.

RECV can be assigned like a method pointer, but it must point to a method which takes no parameters and returns a single value:

```
RECV := @InMethod
```

An example of using RECV:

```
VAR i

PUB Go()
  RECV := @GetPattern
  REPEAT
    PINWRITE(56 ADDPINS 7, !RECV())
    WAITMS(125)

PRI GetPattern() : Pattern
  RETURN DECOD(i++ & 7)
```

In the above example, the following values are output in repeating sequence: $01, $02, $04, $08, $10, $20, $40, $80 (but inverted for LEDs)

Though a called method inherits the current RECV pointer, it may change it for its own purposes. Upon return from that method, the RECV pointer will be back to what it was before the method was called. So, the RECV pointer value is propagated in method calls, but not in method returns.

FLOW CONTROL

Spin2 has three basic flow-control constructs:

IF / IFNOT + ELSEIF / ELSEIFNOT + ELSE	- Conditional execution with random decision logic
CASE / CASE_FAST	- Conditional execution with single target and multiple match tests
REPEAT	- Looped execution with various modes

All these constructs use relative indentation to determine which code falls under their control:

```
  IF cog                              'if cog <> 0
    COGSTOP(cog-1)                    '..then stop cog
    PINCLEAR(av_base_pin_ ADDPINS 4)  '..then clear pin mode(s)
```

The flow-control constructs can be nested in any order:

```
  CASE flag
    0: 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)

    2:    col := chr // cols
    3:    row := chr // rows
    4..7: background0_[flag-$04] := chr << 8
  flag := 0
```

IF / IFNOT + ELSEIF / ELSEIFNOT + ELSE

The IF construct begins with IF or IFNOT and optionally employs ELSEIF, ELSEIFNOT, and ELSE. To all be part of the same decision tree, these keywords must have the same level of indentation.

The indented code under IF or ELSEIF executes if <condition> is not zero. The code under IFNOT or ELSEIFNOT executes if <condition> is zero. The code under ELSE executes if no other indented code executed:

```
IF / IFNOT <condition>                - Initial IF or IFNOT
  <indented code>
ELSEIF / ELSEIFNOT <condition>        - Optional ELSEIF or ELSEIFNOT
  <indented code>
ELSE                                  - Optional final ELSE
```

```
  <indented code>
```

CASE / CASE_FAST

The CASE construct sequentially compares a target value to a list of possible matches. When a match is found, the related code executes.

Match values/ranges must be indented past the CASE keyword. Multiple match values/ranges can be expressed with comma separators. Any additional lines of code related to the match value/range must be indented past the match value/range:

```
CASE target                         - CASE with target value
  <match> : <code>                  - match value and code
    <indented code>
  <match..match> : <code>           - match range and code
    <indented code>
  <match>,<match..match> : <code>   - match value, range, and code
    <indented code>
  OTHER : <code>                    - optional OTHER case, in case no match found
    <indented code>
```

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, saving time at a possible cost of larger compiled code. If there are only contiguous match values and no match ranges, the resulting code will actually be smaller than a normal CASE construct with more than several match values.

For CASE_FAST to compile, the match values/ranges must be unique constants which are all within 255 of each other.

See CASE_FAST example under "FLOW CONTROL" above.

REPEAT

All looping is achieved through REPEAT constructs, which have several forms:

```
REPEAT                              - Repeat forever (useful for putting at end of program if you don't want the cog to stop and cease driving its I/O's)
  <indented code>

REPEAT <count>                      - Repeat <count> times, if <count> is zero then <indented code> is skipped
  <indented code>

REPEAT <variable> FROM <first> TO <last>               - Repeat while iterating <variable> from <first> to <last>, stepping by +/-1
  <indented code>                                      - After completion, <variable> = <last> +/-1

REPEAT <variable> FROM <first> TO <last> STEP <delta>  - Repeat while iterating <variable> from <first> to <last>, stepping by +/-<delta>
  <indented code>                                      - After completion, <variable> = <last> +/-<delta>

REPEAT WHILE <condition>            - Repeat while <condition> is not zero, <condition> is evaluated before <indented code> executes
  <indented code>

REPEAT UNTIL <condition>            - Repeat until <condition> is not zero, <condition> is evaluated before <indented code> executes
  <indented code>

REPEAT                              - Repeat while <condition> is not zero, <condition> is evaluated after <indented code> executes
  <indented code>
WHILE <condition>                   - WHILE must have same indentation as REPEAT

REPEAT                              - Repeat until <condition> is not zero, <condition> is evaluated after <indented code> executes
  <indented code>
UNTIL <condition>                   - UNTIL must have same indentation as REPEAT
```

Within REPEAT constructs, there are two special instructions which can be used to change the course of execution: NEXT and QUIT. NEXT will immediately branch to the point in the REPEAT construct where the decision to loop again is made, while QUIT will exit the REPEAT construct and continue after it. These instructions are usually used conditionally:

```
REPEAT
  <indented code>
  IF <condition>                    - Optionally force the next iteration of the REPEAT
    NEXT
```

```
  <indented code>
  IF <condition>                    - Optionally quit the REPEAT
    QUIT
  <indented code>
```

IN-LINE PASM CODE

Spin2 methods can execute in-line PASM code by preceding the PASM code with an **'ORG** {$000..$12F}' and terminating it with an **END**.

```
PUB go() | x

  REPEAT

    ORG
        GETRND  WC      'rotate a random bit into x
        RCL     x,#1
    END

    PINWRITE(56 ADDPINS 7, x)     'output x to the P2 Eval board's LEDs
    WAITMS(100)
```

Your PASM code will be assembled with a RET instruction added at the end to ensure that it returns to Spin2, in case no early _RET_ or RET executes.

Here's the internal Spin2 procedure for executing in-line PASM code:

- Save the current streamer address for restoration after the PASM code executes.
- Copy the method's first 16 long variables, including any parameters, return values, and local variables, from hub RAM to cog registers $1E0..$1EF.
- Copy the in-line PASM-code longs from hub RAM into cog registers, starting at the ORG address (default is $000).
- CALL the PASM code.
- Restore the 16 longs in cog registers $1E0..$1EF back to hub RAM, in order to update any modified method variables.
- Restore the streamer address and resume Spin2 bytecode execution.

Within your in-line PASM code, you can do all these things:

- Read and write the following register areas:
  - $000..$12F, which your PASM code loads into. You can even load different PASM programs at different addresses within this range and CALL them from Spin2.
  - $1D8..$1DF, which are general-purpose registers, named PR0..PR7, available to both PASM and Spin2 code.
  - $1E0..$1EF, which temporarily contain the method's first 16 long hub RAM variables and are temporarily assigned the same symbolic names.
  - $1F0..$1FF, which include IJMP3, IRET3, IJMP2, IRET2, IJMP1, IRET1, PA, PB, PTRA, PTRB, DIRA, DIRB, OUTA, OUTB, INA, and INB.
  - Avoid writing to $130..$1D7 and LUT RAM, since the Spin2 interpreter occupies these areas. You can look in "Spin2_interpreter.spin2" to see the interpreter code.
- Use the streamer temporarily.
- Use up to 5 levels of the hardware stack for nested CALLs, including CALLs to hub RAM.
- Declare and reference regular and local symbols. These symbols will not be accessible outside of your PASM code.
- Declare BYTE, WORD, and LONG data.
- Use the RES, ORGF, and FIT directives. The directives ORG, ORGH, ALIGNW, ALIGNL, and FILE are not allowed within in-line PASM code.
- Establish an interrupt which executes your code remaining in cog registers $000..$12F. Spin2 accommodates interrupts and only stalls them briefly, when necessary.
- Return to Spin2, at any point, by executing an _RET_ or RET instruction.

CALLING PASM FROM SPIN2

You can do a **CALL(address)** in Spin2 to execute PASM code in either cog register space or hub RAM.

```
PUB go() | x

  REPEAT
    CALL(@random)
    PINWRITE(56 ADDPINS 7, PR0)
    WAITMS(100)


DAT     ORGH    'hub PASM program to rotate a random bit into pr0

random  GETRND  WC
 _RET_  RCL     PR0,#1
```

Here's the internal Spin2 procedure for executing a CALL:

- Save the current streamer address for restoration after the PASM code executes.
- CALL the PASM code.
- Restore the streamer address and resume Spin2 bytecode execution.

Within code which you CALL, you can do all these things:

- Read and write the following register areas:
  - $000..$12F, which may contain PASM code and/or data which you previously loaded.
  - $1D8..$1DF, which are general-purpose registers, named PR0..PR7, available to both PASM and Spin2 code.
  - $1E0..$1EF, which are available for scratchpad use, but will likely be rewritten when Spin2 resumes.
  - $1F0..$1FF, which include IJMP3, IRET3, IJMP2, IRET2, IJMP1, IRET1, PA, PB, PTRA, PTRB, DIRA, DIRB, OUTA, OUTB, INA, and INB.
  - Avoid writing to $130..$1D7 and LUT RAM, since the Spin2 interpreter occupies these areas. You can look in "Spin2_interpreter.spin2" to see the interpreter code.
- Use the streamer temporarily.
- Use up to 5 levels of the hardware stack for nested CALLs, including CALLs to hub RAM.
- Establish an interrupt which executes your code remaining in cog registers $000..$12F. Spin2 accommodates interrupts and only stalls them briefly, when necessary.
- Return to Spin2, at any point, by executing an _RET_ or RET instruction.

REGLOAD and REGEXEC

The Spin2 instructions **REGLOAD(HubAddress)** and **REGEXEC(HubAddress)** are used to load or load-and-execute PASM code and/or data chunks from hub RAM into cog registers.

The chunk of PASM code and/or data must be preceded with two words which provide the starting register and the number of registers (longs) to load, minus 1.

```
PUB go()

  REGLOAD(@chunk)    'load self-defined chunk from hub into registers

  REPEAT
    CALL(#start)     'call program within chunk at register address
    WAITMS(100)

DAT

chunk   WORD    start,finish-start-1  'define chunk start and size-1
        ORG     $120                  'org can be $000..$130-size

start   DRVRND  #56 ADDPINS 7         'some code
 _RET_  DRVNOT  #0                    'more code + return
finish
```

REGEXEC works like REGLOAD, but it also CALLs to the start register of the chunk after loading it.

In the example below, REGEXEC launches a chunk of code in upper register memory which sets up a timer interrupt and then returns to Spin2. Meanwhile, as the Spin2 method repeatedly randomizes pins 60..63 every 100ms, the chunk of code loaded into upper register memory perpetuates the timer interrupt and toggles pins 56..59 every 500ms. Note that registers $000..$127 are still free for other code chunks and interrupts 2 and 3 are still unused.

```
PUB go()

  REGEXEC(@chunk)                       'load self-defined chunk and execute it
                                        'chunk starts timer interrupt and returns
  REPEAT
    PINWRITE(60 ADDPINS 3, GETRND())    'randomize pins 60..63
    WAITMS(100)                         'pins 56..59 toggle via interrupt

DAT

chunk   WORD    start,finish-start-1    'define chunk start and size-1
        ORG     $128                    'org can be $000..$130-size

start   MOV     IJMP1,#isr              'set int1 vector
        SETINT1 #1                      'set int1 to ct-passed-ct1 event
        GETCT   PR0                     'get ct
 _ret_  ADDCT1  PR0,bigwait             'set initial ct1 target, return to Spin2

isr     DRVNOT  #56 ADDPINS 3           'interrupt service routine, toggle 56..59
        ADDCT1  PR0,bigwait             'set next ct1 target
        RETI1                           'return from interrupt

bigwait LONG    20_000_000 / 2          '500ms second on RCFAST
finish
```

DEBUG

The Spin2 compiler contains a stealthy debugger program that can be automatically downloaded with your application. It uses the last 16KB of RAM plus a few bytes for each Spin2 DEBUG statement and one instruction for each PASM DEBUG statement. You place DEBUG statements in your application which contain output commands that will serially transmit the state of variables and equations as your application runs. Each time a DEBUG statement is encountered during execution, the debugger is invoked and it outputs the message for that statement. Debugging is initiated in PNut by adding the Ctrl key to the usual F10 to 'run' or F11 to 'program', or in Propeller Tool by enabling Debug Mode with Ctrl+D then using F10 or F11 as is normal. This compiles your application with all the DEBUG statements, adds the debugger to the download, and then brings up the DEBUG Output window which begins receiving messages at the start of your application.

Things to know about the DEBUG system

- To use the debugger, you must configure at least a 10 MHz clock derived from a crystal or external input. You cannot use RCFAST or RCSLOW.
- The debugger occupies the top 16 KB of hub RAM, remapped to $FC000..$FFFFF and write-protected. The hub RAM at $7C000..$7FFFF will no longer be available.
- Data defining each DEBUG statement is stored within the debugger image in the top 16 KB of RAM, minimizing impact on your application code.
- In Spin2, each DEBUG statement adds three bytes, plus any code needed to reference variables and resolve run-time expressions used in the DEBUG statement.
- In PASM, each DEBUG statement adds one instruction (long).
- DEBUG statements are ignored by the compiler when not compiling for DEBUG mode, so you don't need to comment them out when debugging is not in use.
- If no DEBUG statements exist in your application, you will still get notification messages when cogs are started.
- Debugging is invoked by holding CTRL (in PNut), or enabling Debug with CTRL+D (in Propeller Tool), before the usual F9..F11 keys, tocompile, download, and program to flash.
- During execution, as DEBUG statements are encountered, text messages are sent out serially on P62 at 2 Mbaud in 8-N-1 format.
- DEBUG messages always start with "CogN ", where N is the cog number, followed by two spaces, and they always end with CR+LF (new line).
- Up to 255 DEBUG statements can exist within your application, since the BRK instruction is used to interrupt and select the particular DEBUG statement definition.
- You can define several symbols to modify debugger behavior: DEBUG_COG, DEBUG_DELAY, DEBUG_BAUD, DEBUG_PIN, DEBUG_TIMESTAMP, etc. See table.
- Each time a debug-enabled cog is started, a debug message is output to indicate the cog number, code address (PTRB), parameter (PTRA), and 'load' or 'jump' mode.
- For Spin2, DEBUG statements can output expression and variable values, hub byte/word/long arrays, and register arrays.
- For PASM, DEBUG statements can output register values/arrays, hub byte/word/long arrays, and constants. PASM syntax is used: implied register or #immediate.
- DEBUG output data can be displayed in decimal, hex, or binary, signed or unsigned, and sized to byte, word, long, or auto. Hub character strings are also supported.
- DEBUG output commands show both the source and value: "DEBUG(UHEX(x))" might output "x = $123".
- DEBUG commands which output data can have multiple sets of parameters, separated by commas: SDEC(x,y,z) and LSTR(ptr1,size1,ptr2,size2)
- Commas are automatically output between data: "DEBUG(UHEX_BYTE(d,e,f), SDEC(g))" might output "d = $45, e = $67, f = $89, g = -1_024".
- All DEBUG output commands have alternate versions, ending in "_" which output only the value: DEBUG(UHEX_BYTE_(d,e,f)) might output "$45, $67, $89".
- DEBUG statements can contain comma-separated strings and characters, aside from commands: DEBUG(`"We got here! Oh, Nooooo...", 13, 13)`
- DEBUG statements may contain IF() and IFNOT() commands to gate further output within the statement. An initial IF/IFNOT will gate the entire message.
- DEBUG statements may contain a final DLY(milliseconds) command to slow down a cog's messaging, since messages may stream at the rate of ~10,000 per second.
- DEBUG serial output can be redirected to a different pin, at a different baud rate, for displaying/logging elsewhere.
- LOCK[15] is allocated by the debugger and used among all cogs during their debug interrupts to time-share the DEBUG serial-transmit pin.
- Command-line supports DEBUG-only mode: PNut -debug {CommPort if not 1} {BaudRate if not 2_000_000}

Commands for use in DEBUG statements

Conditionals	Details
`IF(condition)`	If condition <> 0 then continue at the next command within the DEBUG statement, else skip all remaining commands and output CR+LF. If used as the first command in the DEBUG statement, IF will gate ALL output for the statement, including the "CogN "+CR+LF. This way, DEBUG messages can be entirely suppressed, so that you can filter what is important.
`IFNOT(condition)`	If condition = 0 then continue at the next command within the DEBUG statement, else skip all remaining commands and output CR+LF. If used as the first command in the DEBUG statement, IFNOT will gate ALL output for the statement, including the "CogN "+CR+LF. This way, DEBUG messages can be entirely suppressed, so that you can filter what is important.

String Output *	Details	Output
`ZSTR(hub_pointer)`	Output zero-terminated string at hub_pointer	`"Hello!"`
`LSTR(hub_pointer,size)`	Output 'size' characters of string at hub_pointer	`"Goodbye."`

Decimal Output, unsigned *	Details	Min Output	Max Output
`UDEC(value)`	Output unsigned decimal value	0	4_294_967_295
`UDEC_BYTE(value)`	Output byte-size unsigned decimal value	0	255
`UDEC_WORD(value)`	Output word-size unsigned decimal value	0	65_535
`UDEC_LONG(value)`	Output long-size unsigned decimal value	0	4_294_967_295
`UDEC_REG_ARRAY(reg_pointer,size)`	Output register array as unsigned decimal values	0	4_294_967_295
`UDEC_BYTE_ARRAY(hub_pointer,size)`	Output hub byte array as unsigned decimal values	0	255
`UDEC_WORD_ARRAY(hub_pointer,size)`	Output hub word array as unsigned decimal values	0	65_535
`UDEC_LONG_ARRAY(hub_pointer,size)`	Output hub long array as unsigned decimal values	0	4_294_967_295
Decimal Output, signed *	**Details**	**Min Output**	**Max Output**

SDEC(value)	Output signed decimal value	-2_147_483_648	2_147_483_647
SDEC_BYTE(value)	Output byte-size signed decimal value	-128	127
SDEC_WORD(value)	Output word-size signed decimal value	-32_768	32_767
SDEC_LONG(value)	Output long-size signed decimal value	-2_147_483_648	2_147_483_647
SDEC_REG_ARRAY(reg_pointer,size)	Output register array as signed decimal values	-2_147_483_648	2_147_483_647
SDEC_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as signed decimal values	-128	127
SDEC_WORD_ARRAY(hub_pointer,size)	Output hub word array as signed decimal values	-32_768	32_767
SDEC_LONG_ARRAY(hub_pointer,size)	Output hub long array as signed decimal values	-2_147_483_648	2_147_483_647
Hexadecimal Output, unsigned *	**Details**	**Min Output**	**Max Output**
UHEX(value)	Output auto-size unsigned hex value	$0	$FFFF_FFFF
UHEX_BYTE(value)	Output byte-size unsigned hex value	$00	$FF
UHEX_WORD(value)	Output word-size unsigned hex value	$0000	$FFFF
UHEX_LONG(value)	Output long-size unsigned hex value	$0000_0000	$FFFF_FFFF
UHEX_REG_ARRAY(reg_pointer,size)	Output register array as unsigned hex values	$0000_0000	$FFFF_FFFF
UHEX_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as unsigned hex values	$00	$FF
UHEX_WORD_ARRAY(hub_pointer,size)	Output hub word array as unsigned hex values	$0000	$FFFF
UHEX_LONG_ARRAY(hub_pointer,size)	Output hub long array as unsigned hex values	$0000_0000	$FFFF_FFFF
Hexadecimal Output, signed *	**Details**	**Min Output**	**Max Output**
SHEX(value)	Output auto-size signed hex value	-$8000_0000	$7FFF_FFFF
SHEX_BYTE(value)	Output byte-size signed hex value	-$80	$7F
SHEX_WORD(value)	Output word-size signed hex value	-$8000	$7FFF
SHEX_LONG(value)	Output long-size signed hex value	-$8000_0000	$7FFF_FFFF
SHEX_REG_ARRAY(reg_pointer,size)	Output register array as signed hex values	-$8000_0000	$7FFF_FFFF
SHEX_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as signed hex values	-$80	$7F
SHEX_WORD_ARRAY(hub_pointer,size)	Output hub word array as signed hex values	-$8000	$7FFF
SHEX_LONG_ARRAY(hub_pointer,size)	Output hub long array as signed hex values	-$8000_0000	$7FFF_FFFF
Binary Output, unsigned *	**Details**	**Min Output**	**Max Output**
UBIN(value)	Output auto-size unsigned binary value	%0	%11111111_11111111_11111111_11111111
UBIN_BYTE(value)	Output byte-size unsigned binary value	%00000000	%11111111
UBIN_WORD(value)	Output word-size unsigned binary value	%00000000_00000000	%11111111_11111111
UBIN_LONG(value)	Output long-size unsigned binary value	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
UBIN_REG_ARRAY(reg_pointer,size)	Output register array as unsigned binary values	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
UBIN_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as unsigned binary values	%00000000	%11111111
UBIN_WORD_ARRAY(hub_pointer,size)	Output hub word array as unsigned binary values	%00000000_00000000	%11111111_11111111
UBIN_LONG_ARRAY(hub_pointer,size)	Output hub long array as unsigned binary values	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
Binary Output, signed *	**Details**	**Min Output**	**Max Output**
SBIN(value)	Output auto-size signed binary value	-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111
SBIN_BYTE(value)	Output byte-size signed binary value	-%10000000	%01111111
SBIN_WORD(value)	Output word-size signed binary value	-%10000000_00000000	%01111111_11111111
SBIN_LONG(value)	Output long-size signed binary value	-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111
SBIN_REG_ARRAY(reg_pointer,size)	Output register array as signed binary values	-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111
SBIN_BYTE_ARRAY(hub_pointer,size)	Output hub byte array as signed binary values	-%10000000	%01111111
SBIN_WORD_ARRAY(hub_pointer,size)	Output hub word array as signed binary values	-%10000000_00000000	%01111111_11111111
SBIN_LONG_ARRAY(hub_pointer,size)	Output hub long array as signed binary values	-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111

Delay to Pace Messages	Details
DLY(milliseconds)	Delay for some milliseconds to slow down continuous message outputs for this cog. DLY is only allowed as the last command in a DEBUG statement, since it releases LOCK[15] before the delay, permitting other cogs to capture LOCK[15] so that they may take control of the DEBUG serial-transmit pin and output their own DEBUG messages.

* These commands accept multiple parameters, or multiple sets of parameters. Alternate commands with the same names, but ending in "_", are also available for value-only output (i.e. ZSTR_, LSTR_, UDEC_).

Symbols you can define to modify DEBUG behavior

CON Symbol	Default	Purpose
DOWNLOAD_BAUD	2_000_000	Sets the download baud rate.
DEBUG_COGS	%11111111	Selects which cogs have debug interrupts enabled. Bits 7..0 enable debugging interrupts in cogs 7..0.
DEBUG_DELAY	0	Sets a delay in milliseconds before your application runs and begins transmitting DEBUG messages.
DEBUG_PIN	62	Sets the DEBUG serial output pin. For DEBUG windows to open, DEBUG_PIN must be 62.
DEBUG_BAUD	DOWNLOAD_BAUD	Sets the DEBUG baud rate. May be necessary to add DEBUG_DELAY if DEBUG_BAUD is less than DOWNLOAD_BAUD.
DEBUG_TIMESTAMP	undefined	By declaring this symbol, each DEBUG message will be time-stamped with the 64-bit CT value.
DEBUG_LOG_SIZE	0	Sets the maximum size of the 'DEBUG.log' file which will collect DEBUG messages. A value of 0 will inhibit log file generation.
DEBUG_LEFT	(dynamic)	Sets the left screen coordinate where the DEBUG message window will appear.
DEBUG_TOP	(dynamic)	Sets the top screen coordinate where the DEBUG message window will appear.
DEBUG_WIDTH	(dynamic)	Sets the width of the DEBUG message window.
DEBUG_HEIGHT	(dynamic)	Sets the height of the DEBUG message window.
DEBUG_DISPLAY_LEFT	0	Sets the overall left screen offset where any DEBUG displays will appear (adds to 'POS' x coordinate in each DEBUG display).
DEBUG_DISPLAY_TOP	0	Sets the overall top screen offset where any DEBUG displays will appear (adds to 'POS' y coordinate in each DEBUG display).
DEBUG_WINDOWS_OFF	0	Disables any DEBUG windows from opening after downloading, if set to a non-zero value.

Simple DEBUG example in Spin2

```
CON _clkfreq = 10_000_000      'set 10 MHz clock (assumes 20 MHz crystal)

PUB go() | i
  REPEAT i FROM 0 TO 9         'count from 0 to 9
    DEBUG(UDEC(i))             'debug, output i
```

When run with Ctrl-F10, the Debug window opens and this is what appears:

```
Cog0  INIT $0000_0000 $0000_0000 load
Cog0  INIT $0000_0D6C $0000_10BC jump
Cog0  i = 0
Cog0  i = 1
Cog0  i = 2
Cog0  i = 3
Cog0  i = 4
Cog0  i = 5
Cog0  i = 6
Cog0  i = 7
Cog0  i = 8
Cog0  i = 9
```

In the first line of the report, you see Cog0 loading the Spin2 set-up code from $00000. In the second line, the Spin2 interpreter is launched from $00D58 with its stack space starting at $0101C. After that, the Spin2 program is running and you see 'i' iterating from 0 to 9.

If you change the "9" to "99" in the REPEAT, data will scroll too fast to read, but by adding a DLY command at the end of the DEBUG statement, you can slow down the output:

```
    debug(udec(i), dly(250))     'debug, output i with a 250ms delay after each report
```

Let's say you want to limit the messages being output, so that only odd values of 'i' are shown. You could use an IF at the start of your DEBUG statement to check the least-significant bit of 'i'. When the IF is false, no message will be output, causing only the odd values of i to be shown:

```
    debug(if(i & 1), udec(i), dly(250))     'debug, output only odd i values with a 250ms delay after each report
```

Simple DEBUG example in PASM

```
CON _clkfreq = 10_000_000       'set 10 MHz clock (assumes 20 MHz crystal)
```

```
DAT     ORG

        MOV     i,#9            'set i to 9
loop    DEBUG   (UHEX_LONG(i))  'debug, output i in hex
        DJNF    i,#loop         'decrement i and loop if not -1
        JMP     #$              'don't go wandering off, stay here

i       RES     1               'reserve one register as 'i'
```

When run with Ctrl-F10, the Debug window opens and this is what appears:

```
Cog0  INIT $0000_0000 $0000_0000 load
Cog0  i = $0000_0009
Cog0  i = $0000_0008
Cog0  i = $0000_0007
Cog0  i = $0000_0006
Cog0  i = $0000_0005
Cog0  i = $0000_0004
Cog0  i = $0000_0003
Cog0  i = $0000_0002
Cog0  i = $0000_0001
Cog0  i = $0000_0000
```

In the first line of the report, you see Cog0 loading our PASM program from $00000. After that, the program runs and you see 'i' iterating from 9 down to 0.

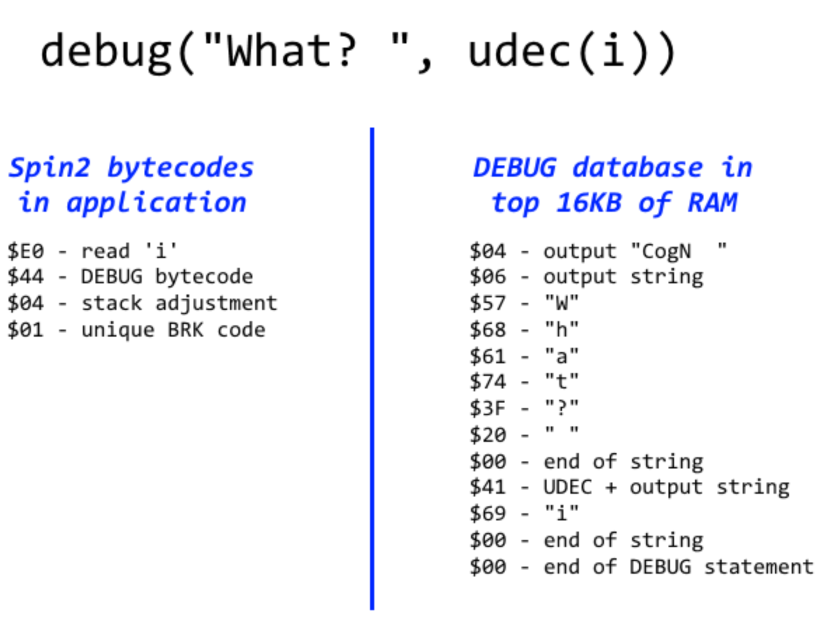
If you change the "9" to "99" in the MOV instruction and you'd like to slow things down, add a DLY command to the DEBUG statement and be sure to express the milliseconds as #250, since a plain 250 would be understood as register 250:

```
    debug   (uhex_long(i), dly(#250))      'debug, output i in hex and delay for 250ms after each report
```

DEBUG memory utilization

Here is what the memory utilization looks like for a Spin2 DEBUG statement. You can see, on the Spin2 side, that a bytecode is needed to read the variable 'i', and then three obligatory bytecodes make up the actual DEBUG instruction.

The 'stack adjustment' byte tells the interpreter how far to drop the stack to effectively 'pop' all the expressions that were pushed in preparation for the DEBUG event. In this case of 'i', only, the stack needs to drop by four bytes (one long). When the debugger is invoked, the values it needs will be ordered right above the current Spin2 stack pointer.



The 'unique BRK code' byte is used as an index to look up the specific record in the DEBUG database at the top of memory, from which the debugger reads its commands.

In the case where DEBUG is active, but a cog has had its debug interrupt disabled via the DEBUG_ENABLE symbol, Spin2 DEBUG instructions will not trigger a debug interrupt, but they do still pop any DEBUG-intended values from the stack, so these are harmless events.

For PASM DEBUG statements, a 'BRK #code' instruction is inserted where the DEBUG command was placed, and all related data resides in the DEBUG database. If the cog's debug interrupt is disabled, the 'BRK #code' instruction does nothing, taking two clocks.

DEBUG and interrupts

Interrupt requests received during a DEBUG statement will execute after the DEBUG completes, but the response time may be so skewed that the retrigger setup for the interrupt won't happen properly. High-frequency cyclical smart pin interrupts are especially prone to this problem. Imagine you do an AKPIN instruction within your normal ISR (interrupt service routine) to drop the INA/INB signal so that the smart pin can make it go high again, triggering a new interrupt. Meanwhile, after the AKPIN and before the RETIx, the smart pin triggers, raising INA/INB high. This is only happening because your cycle-frame timing has become skewed from the DEBUG statement. This interrupt won't be seen since it happened when the ISR was busy. This will cause the interrupt to cease cycling. CT interrupts are not prone to this problem, though, since they have $8000_0000 clock cycles in which to be recognized. To remedy the smart-pin retrigger problem, you could trigger on INA/INB-high, as opposed to INA/INB-rise, but this could cause performance problems with your smart pin configurations.

One fail-safe way to get around this DEBUG/interrupt dilemma is to only do DEBUG statements from cogs that are not executing ISRs in the background. If the ISRs can tolerate timing skew and there is no risk of hanging interrupt cycling, you can do DEBUG statements with some understood interrupt timing degradations.

Graphical DEBUG Displays

DEBUG messages can invoke special graphical DEBUG displays which are built into the tool. These graphical displays each take the form of a unique window. Once instantiated, displays can be continuously fed data to generate animated visualizations. These displays are very handy for development and debugging, as various data types can be viewed in their native contexts. Up to 32 graphical displays can be running simultaneously.

When a DEBUG message contains a backtick (`) character (ASCII $60), a string, containing everything from the backtick to the end of the message, is sent to the graphical DEBUG display parser. The parser looks for several different element types, treating any commas as whitespace:

Element Type	Example	Description
display_type	LOGIC, SCOPE, PLOT, BITMAP	This is the formal name of the graphical DEBUG display type you wish to instantiate.
unknown_symbol	MyLogicDisplay	Each graphical DEBUG display Instance must be given a unique symbolic name.
instance_name	MyLogicDisplay	Once instantiated, a graphical DEBUG display instance is referenced by its symbolic name.
keyword	TITLE, POS, SIZE, SAMPLES	Keywords are used to configure displays. They might be followed by numbers, strings, and other keywords.
number	1024, $FF, %1010	Numbers can be expressed in decimal, hex ($), and binary (%).
string	'Here is a string'	Strings are expressed within single-quotes.

Before getting into how all this fits together, we need to go over some special DEBUG-display syntax that can be used for displays. This syntax is invoked when the first character in the DEBUG statement is the backtick. This causes everything in the DEBUG statement to be viewed as a string, except when subsequent backticks act as 'escape' characters to allow normal or shorthand DEBUG commands.

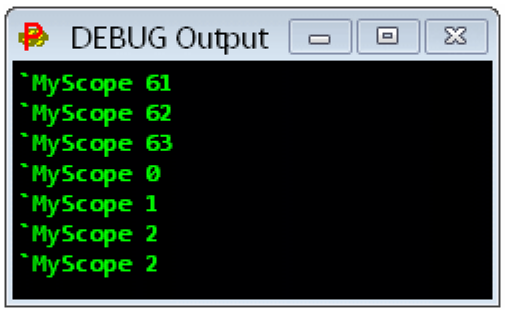
DEBUG Statement (v=100, BYTE[a]=1,2,3,4,5)	DEBUG Message Output	Note
DEBUG("`LOGIC MyDisplay SAMPLES ", SDEC_(v))	Cog0 `LOGIC MyDisplay SAMPLES 100	Regular DEBUG syntax can drive DEBUG displays, but it's not optimal.
DEBUG(`LOGIC MyDisplay SAMPLES 100)	`LOGIC MyDisplay SAMPLES 100	DEBUG-display syntax is simpler and 'CogN' is omitted in the output.
DEBUG(`LOGIC MyDisplay SAMPLES `(v))	`LOGIC MyDisplay SAMPLES 100	Decimal numbers are output using `(value) notation. Short for SDEC_.
DEBUG(`LOGIC MyDisplay SAMPLES `$(v))	`LOGIC MyDisplay SAMPLES $64	Hex numbers are output using `$(value) notation. Short for UHEX_.
DEBUG(`LOGIC MyDisplay SAMPLES `%(v))	`LOGIC MyDisplay SAMPLES %1100100	Binary numbers are output using `%(value) notation. Short for UBIN_.
DEBUG(`LOGIC MyDisplay TITLE '`#(v)')	`LOGIC MyDisplay TITLE 'd'	Characters are output using `#(value) notation.
DEBUG(`MyDisplay `UDEC_BYTE_ARRAY_(@a,5))	`MyDisplay 1, 2, 3, 4, 5	Regular DEBUG commands can follow the backtick, as well.

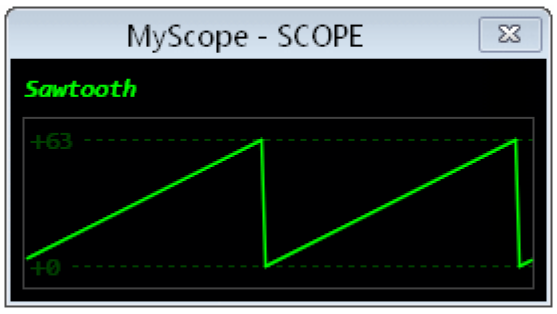
There are two steps to using graphical DEBUG displays. First, they must be instantiated and, second, they must be fed:

To Use a Display:	1st	2nd	3rd	4th	Note
First, instantiate it.	`	display_type	unknown_symbol	keyword(s), number(s), string(s)	Unknown_symbol becomes instance_name.
Then, feed it.	`	instance_name(s)	keyword(s), number(s), string(s)		Multiple displays can be fed the same data.

To bring this all together, let's show a sawtooth wave on a SCOPE display:

```
CON _clkfreq = 10_000_000

PUB go() | i

  debug(`SCOPE MyScope SIZE 254 84 SAMPLES 128)
  debug(`MyScope 'Sawtooth' 0 63 64 10 %1111)

  repeat
    debug(`MyScope `(i & 63))
    i++
    waitms(50)
```





In the example above, a SCOPE is instantiated called MyScope that is 254 x 84 pixels and shows 128 samples. A width of 254 was chosen since samples are numbered 0..127 and I wanted them to be spaced at a constant two-pixel pitch (127 * 2 = 254). A height of 84 was chosen so that there would be 10 pixels above and below the waveform, which will have a height of 64 pixels. A channel called "Sawtooth" is defined which, for the purpose of display, has a bottom value of 0 and a top value of 63, is 64 pixels tall within that range, and is elevated 10 pixels off the bottom of the scope window. The %1111 enables top and bottom legend values and top and bottom lines. Within the REPEAT block, the SCOPE is fed a repeating pattern of 0..63 which forms the sawtooth wave. The SCOPE updates its display each time it receives a value. If there were eight channels defined, instead of just one, it would update the display on every eighth value received, drawing all eight channels.

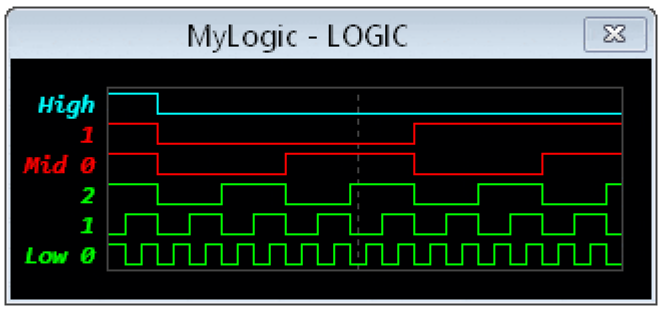
Currently, the following graphical DEBUG displays are implemented, but more will be added in the future:

Display Types	Descriptions
LOGIC	Logic analyzer with single and multi-bit labels, 1..32 channels, can trigger on pattern
SCOPE	Oscilloscope with 1..8 channels, can trigger on level with hysteresis
SCOPE_XY	XY oscilloscope with 1..8 channels, persistence of 0..512 samples, polar mode, log scale mode
FFT	Fast Fourier Transform with 1..8 channels, 4..2048 points, windowed results, log scale mode
SPECTRO	Spectrograph with 4..2048-point FFT, windowed results, phase-coloring, and log scale mode
PLOT	General-purpose plotter with cartesian and polar modes
TERM	Text terminal with up to 300 x 200 characters, 6..200 point font size, 4 simultaneous color schemes
BITMAP	Bitmap, 1..2048 x 1..2048 pixels, 1/2/4/8/16/32-bit pixels with 19 color systems, 15 direction/autoscroll modes, independent X and Y pixel size of 1..256
MIDI	Piano keyboard with 1..128 keys, velocity depiction, variable screen scale

Following are elaborations of each DEBUG display type.

LOGIC Display   Logic analyzer with single and multi-bit labels, 1..32 channels, can trigger on pattern

```
CON _clkfreq   = 10_000_000

PUB go() | i

  debug(`LOGIC MyLogic SAMPLES 32 'Low' 3 'Mid' 2 'High')
  debug(`MyLogic TRIGGER $07 $04 HOLDOFF 2)

  repeat
    debug(`MyLogic `(i & 63))
    i++
    waitms(25)
```

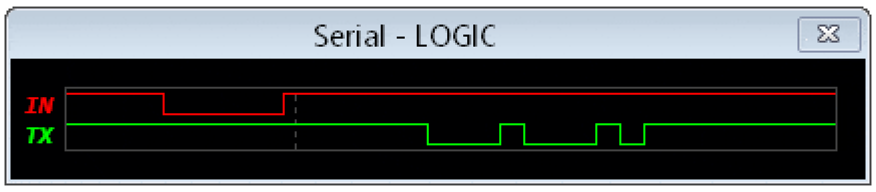


LOGIC Instantiation	Description	Default
`TITLE 'string'`	Set the window caption to 'string'.	<none>
`POS left top`	Set the window position.	0, 0
`SAMPLES 4_to_2048`	Set the number of samples to track and display.	32
`SPACING 2_to_32`	Set the sample spacing. The width of the display will be SAMPLES * SPACING.	8
`RATE 1_to_2048`	Set the number of samples (or triggers, if enabled) before each display update.	1
`LINESIZE 1_to_7`	Set the line size.	1
`TEXTSIZE 6_to_200`	Set the legend text size. Height of text determines height of logic levels.	editor text size
`COLOR back_color {grid_color}`	Set the background and grid colors *.	BLACK, GREY 4
`'name' {1_to_32 {color}}`	Set the first/next channel or group name, optional bit count, optional color *.	1, default color
`packed_data_mode`	Enable packed-data mode. See description at end of this section.	<none>
LOGIC Feeding	**Description**	**Default**
`TRIGGER mask match sample_offset`	Trigger on (data & mask) = match. If mask = 0, trigger is disabled.	0, 1, SAMPLES / 2
`HOLDOFF 2_to_2048`	Set the minimum number of samples required from trigger to trigger.	SAMPLES
`data`	Numerical data is applied LSB-first to the channels.	
`CLEAR`	Clear the sample buffer and display, wait for new data.	
`SAVE {WINDOW} 'filename'`	Save a bitmap file (.bmp) of either the entire window or just the display area.	
`CLOSE`	Close the window.	

* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

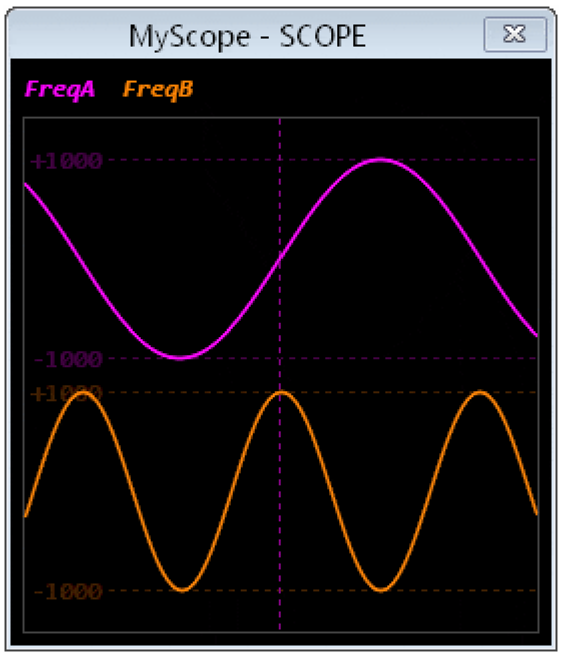
The LOGIC display can be used to display data that was captured at high speed. In the example below, the P2 is generating 8-N-1 serial at 333 Mbaud using a smart pin. This bit stream can be captured by the streamer. On every clock, the streamer will record the smart pin's IN signal and its output state, as read from an adjacent pin. Every time it gets four two-bit sample sets, it does an RFBYTE to save them to hub RAM, forming contiguous bytes, words, and longs. By invoking the LONGS_2BIT packed-data mode, we can have the LOGIC display unpack the two-bit sample sets from longs, yielding 16 sets per long.

```
CON _clkfreq   = 333_333_333  'go really fast, 3ns clock period
    rxpin      = 24           'even pin
    txpin      = rxpin+1      'odd pin
    samps      = 32           'multiple of 16 samples
    bufflongs  = samps / 16   'each long holds 16 2-bit samples
    xmode      = $D0800000 + rxpin << 17 + samps   'streamer mode

VAR buff[bufflongs]

PUB go() | i, buffaddr

  debug(`logic Serial samples `(samps) spacing 12 'TX' 'IN' longs_2bit)
  debug(`Serial trigger %10 %10 22)
  buffaddr := @buff

  repeat
    org
    wrpin     ##+1<<28,#rxpin       'rxpin inputs txpin at rxpin+1

    wrpin     #%01_11110_0,#txpin   'set async tx mode for txpin
    wxpin     ##1<<16+8-1,#txpin    'set baud=sysclock/1 and size=8
    dirh      #txpin                'enable smart pin

    wrfast    #0,buffaddr           'set write-fast at buff
    xinit     ##xmode,#0            'start capturing 2-bit samples

    wypin     i,#txpin              'transmit serial byte

    waitxfi                          'wait for streamer capture done
    end

    debug(`Serial `uhex_long_array_(@buff, bufflongs))
    i++
    waitms(20)
```



## SCOPE Display	Oscilloscope with 1..8 channels, can trigger on level with hysteresis

```
CON _clkfreq   = 100_000_000

PUB go() | a, af, b, bf

  debug(`SCOPE MyScope)
  debug(`MyScope 'FreqA' -1000 1000 100 136 15 MAGENTA)
  debug(`MyScope 'FreqB' -1000 1000 100 20 15 ORANGE)
  debug(`MyScope TRIGGER 0 HOLDOFF 2)

  repeat
    a := qsin(1000, af++, 200)
    b := qsin(1000, bf++, 99)
    debug(`MyScope `(a,b))
    waitus(200)
```

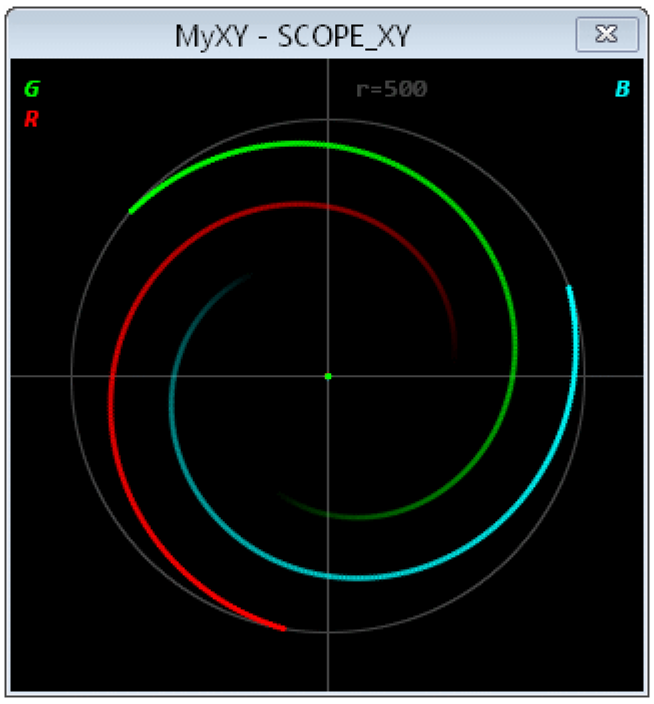


SCOPE Instantiation	Description	Default
`TITLE 'string'`	Set the window caption to 'string'.	<none>
`POS left top`	Set the window position.	0, 0
`SIZE width height`	Set the display size (32..2048 x 32..2048)	255, 256
`SAMPLES 16_to_2048`	Set the number of samples to track and display.	256
`RATE 1_to_2048`	Set the number of samples (or triggers, if enabled) before each display update.	1
`DOTSIZE 0_to_32`	Set the dot size in pixels for showing exact sample points.	0
`LINESIZE 0_to_32`	Set the line size in half-pixels for connecting sample points.	3
`TEXTSIZE 6_to_200`	Set the legend text size.	editor text size
`COLOR back_color {grid_color}`	Set the background and grid colors *.	BLACK, GREY 4
`packed_data_mode`	Enable packed-data mode. See description at end of this section.	<none>
SCOPE Feeding	**Description**	**Default**
`'name' {min {max {y_size {y_base {legend {color}}}}}}`	Set first/next channel name, min value, max value, y size, y base, legend, and color *. Legend is %abcd, where %a to %d enable max legend, min legend, max line, min line.	full, no legend, default color
`TRIGGER channel {arm_level {trigger_level {offset}}}`	Set the trigger channel, arm level, trigger level, and right offset. If channel=-1, disabled.	-1, -1, 0, width / 2
`HOLDOFF 2_to_2048`	Set the minimum number of samples required from trigger to trigger.	SAMPLES
`data`	Numerical data is applied to the channels in ascending order.	
`CLEAR`	Clear the sample buffer and display, wait for new data.	
`SAVE {WINDOW} 'filename'`	Save a bitmap file (.bmp) of either the entire window or just the display area.	
`CLOSE`	Close the window.	

* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

SCOPE_XY Display XY oscilloscope with 1..8 channels, persistence of 1..512 samples, polar mode, log scale mode

```
CON _clkfreq = 100_000_000

PUB go() | i

  debug(`SCOPE_XY MyXY RANGE 500 POLAR 360 'G' 'R' 'B')

  repeat
    repeat i from 0 to 500
      debug(`MyXY `(i, i, i, i+120, i, i+240))
      waitms(5)
```

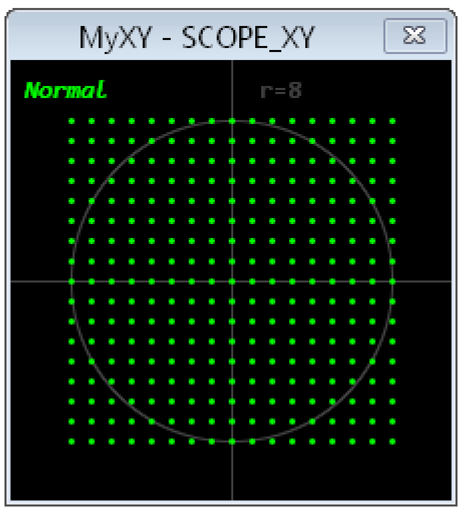


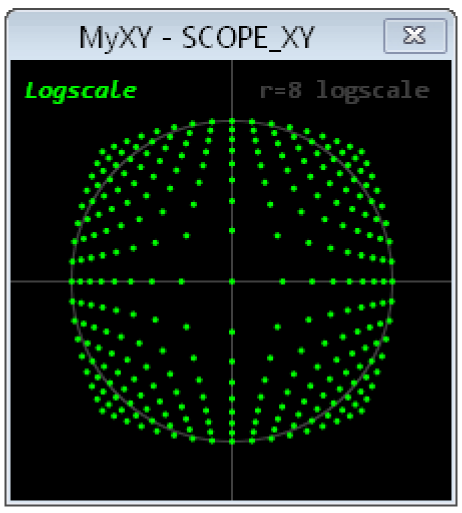
SCOPE_XY Instantiation	Description	Default
TITLE 'string'	Set the window caption to 'string'.	<none>
POS left top	Set the window position.	0, 0
SIZE radius	Set the display radius in pixels.	128
RANGE 1_to_7FFFFFFF	Set the unit circle radius for incoming data	$7FFFFFFF
SAMPLES 0_to_512	Set the number of samples to track and display with persistence. Use 0 for infinite persistence.	256
RATE 1_to_512	Set the number of samples before each display update.	1
DOTSIZE 2_to_20	Set the dot size in half-pixels for showing sample points.	6
TEXTSIZE 6_to_200	Set the legend text size.	editor text size
COLOR back_color {grid_color}	Set the background and grid colors *.	BLACK, GREY 4
POLAR {twopi {offset}}	Set polar mode, twopi value, and offset. For a twopi value of $100000000 or -$100000000, use 0 or -1.	$100000000, 0
LOGSCALE	Set log-scale mode to magnify points within the unit circle.	<off>
'name' {color}	Set the first/next channel name and optionally assign it a color *.	default color
packed_data_mode	Enable packed-data mode. See description at end of this section.	<none>
SCOPE_XY Feeding	**Description**	**Default**
x y	X-Y data pairs are applied to the channels in ascending order. In polar mode, x=length and y=angle.	
CLEAR	Clear the sample buffer and display, wait for new data.	
SAVE {WINDOW} 'filename'	Save a bitmap file (.bmp) of either the entire window or just the display area.	
CLOSE	Close the window.	

* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

```
CON _clkfreq = 10_000_000     'Normal mode

PUB go() | x, y
  debug(`SCOPE_XY MyXY SIZE 80 RANGE 8 SAMPLES 0 'Normal')
  repeat x from -8 to 8
    repeat y from -8 to 8
      debug(`MyXY `(x,y))
```

```
CON _clkfreq = 10_000_000     'LOGSCALE mode magnifies low-level details

PUB go() | x, y
  debug(`SCOPE_XY MyXY SIZE 80 RANGE 8 SAMPLES 0 LOGSCALE 'Logscale')
  repeat x from -8 to 8
    repeat y from -8 to 8
      debug(`MyXY `(x,y))
```

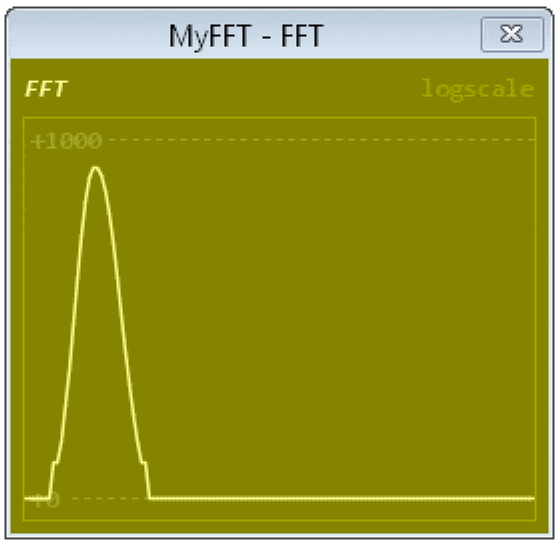


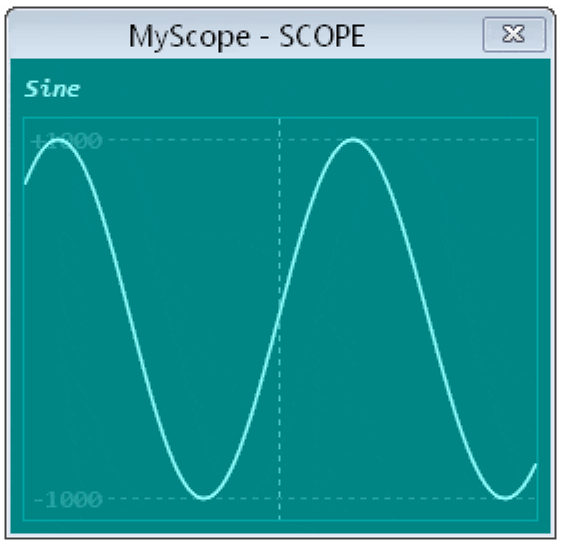


FFT Display

Fast Fourier Transform with 1..8 channels, 4..2048 points, windowed results, log scale mode

```
CON _clkfreq = 100_000_000

PUB go() | i, j, k

  ' Set up FFT
  debug(`FFT MyFFT SIZE 250 200 SAMPLES 2048 0 127 RATE 256 LOGSCALE COLOR YELLOW 4 YELLOW 5)
  debug(`MyFFT 'FFT' 0 1000 180 10 15 YELLOW 12)

  ' Set up SCOPE
  debug(`scope MyScope POS 300 0 SIZE 255 200 COLOR CYAN 4 CYAN 5)
  debug(`MyScope 'Sine' -1000 1000 180 10 15 CYAN 12)
  debug(`MyScope TRIGGER 0)

  repeat
    j += 1550 + qsin(1300, i++, 31_000)
    k := qsin(1000, j, 50_000)
    debug(`MyFFT MyScope `(k))
    waitus(100)
```



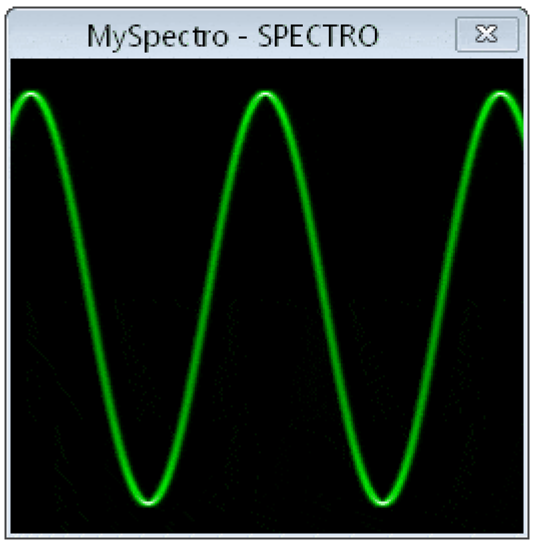


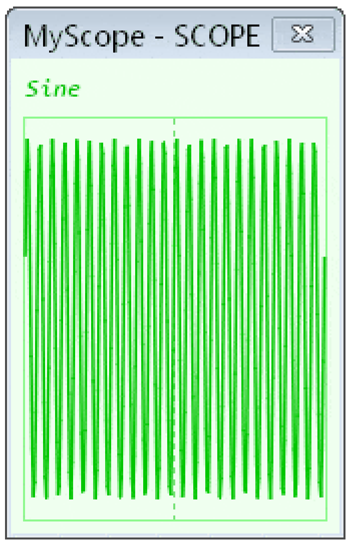
FFT Instantiation	Description	Default
`TITLE 'string'`	Set the window caption to 'string'.	<none>
`POS left top`	Set the window position.	0, 0
`SIZE width height`	Set the display size (32..2048 x 32..2048)	256, 256
`SAMPLES 4_to_2048 {first {last}}`	Set the 2^n number of FFT inputs points, plus the first and last result values to display.	512, 0, 255
`RATE 1_to_2048`	Set the number of samples before each display update.	SAMPLES
`DOTSIZE 0_to_32`	Set the dot size in pixels for showing exact sample points.	0
`LINESIZE neg32_to_32`	Set the line size in half-pixels for connecting sample points. A negative line size will make isolated vertical lines.	3
`TEXTSIZE 6_to_200`	Set the legend text size.	editor text size
`COLOR back_color {grid_color}`	Set the background and grid colors *.	BLACK, GREY 4
`LOGSCALE`	Set log-scale mode to magnify low-level results.	<off>
`packed_data_mode`	Enable packed-data mode. See description at end of this section.	<none>
FFT Feeding	**Description**	**Default**
`'name' {mag {max {y_size {y_base {legend {color}}}}}}`	Set the first/next channel name, magnification factor (2^n, n = 0..11), max amplitude, y size, y base, legend, and color *. Legend is %abcd, where %a to %d enable max legend, min legend, max line, min line.	full, no legend, default color
`data`	Numerical data is fed into the channels' sliding Hanning windows from which the FFT computes power levels.	
`CLEAR`	Clear the sample buffer and display, wait for new data.	
`SAVE {WINDOW} 'filename'`	Save a bitmap file (.bmp) of either the entire window or just the display area.	
`CLOSE`	Close the window.	

* Color is rgb24 value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

SPECTRO Display Spectrograph with 4..2048-point FFT, phase-coloring, and log scale mode

```
CON _clkfreq = 100_000_000

PUB go() | i, j, k

  ' Set up SPECTRO
  debug(`SPECTRO MySpectro SAMPLES 2048 0 236 RANGE 1000 LUMA8X GREEN)

  ' Set up SCOPE
  debug(`SCOPE MyScope POS 280 SIZE 150 200 COLOR GREEN 15 GREEN 12)
  debug(`MyScope 'Sine' -1000 1000 180 10 0 GREEN 6)
  debug(`MyScope TRIGGER 0)

  repeat
    j += 2850 + qsin(2500, i++, 30_000)
    k := qsin(1000, j, 50_000)
    debug(`MySpectro MyScope `(k))
    waitus(100)
```

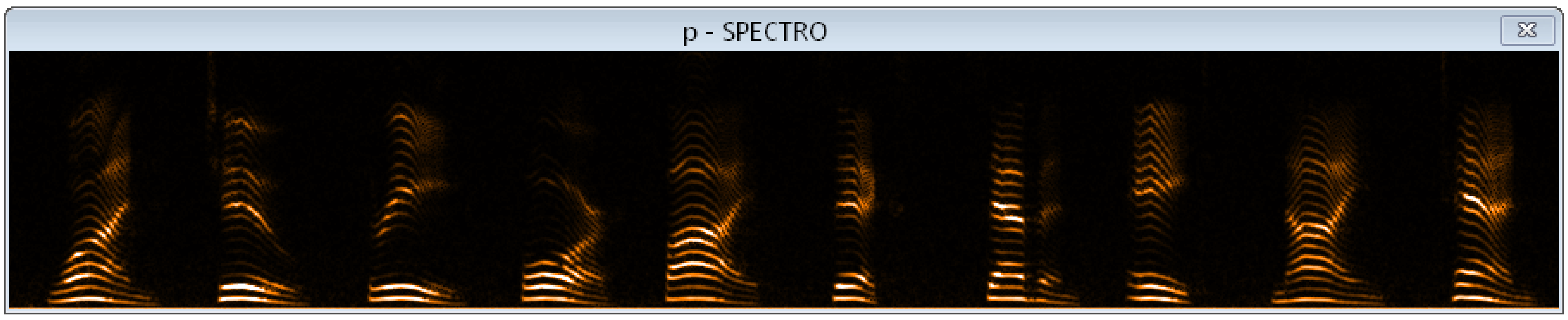




SPECTRO Instantiation	Description	Default
`TITLE 'string'`	Set the window caption to 'string'.	<none>
`POS left top`	Set the window position.	0, 0
`SAMPLES 4_to_2048 {first {last}}`	Set the 2^n number of FFT input points, plus the first and last result values to display (defines display height).	512, 0, 255
`DEPTH 1_to_2048`	Set the number of vertical-line FFT results to display (defines the display width).	256
`MAG 0_to_11`	Set the magnification factor (2^n, n = 0..11).	0
`RANGE saturation_power`	Set the power level at which pixel brightness saturates.	$7FFFFFFF
`RATE 1_to_2048`	Set the number of samples before each display update.	SAMPLES / 8
`TRACE 0_to_15`	Set the trace pattern (see TRACE animation in BITMAP Display).	15 (right, up, scroll)
`DOTSIZE width_and_height {height}`	Set the spectrograph pixel-width and pixel-height together, or set them independently.	1, 1
`luma_or_hsv {color_or_phase}`	Set the color scheme to LUMA8(W)(X) with color *, or HSV16(W)(X) with 0..255 phase-coloring offset.	LUMA8X ORANGE
`LOGSCALE`	Set log-scale mode to magnify low-level results.	<off>
`packed_data_mode`	Enable packed-data mode. See description at end of this section.	<none>
SPECTRO Feeding	**Description**	**Default**
`data`	Numerical data is fed into a sliding Hanning window from which the FFT computes power and phase.	
`CLEAR`	Clear the sample buffer and display, wait for new data.	
`SAVE {WINDOW} 'filename'`	Save a bitmap file (.bmp) of either the entire window or just the display area.	
`CLOSE`	Close the window.	

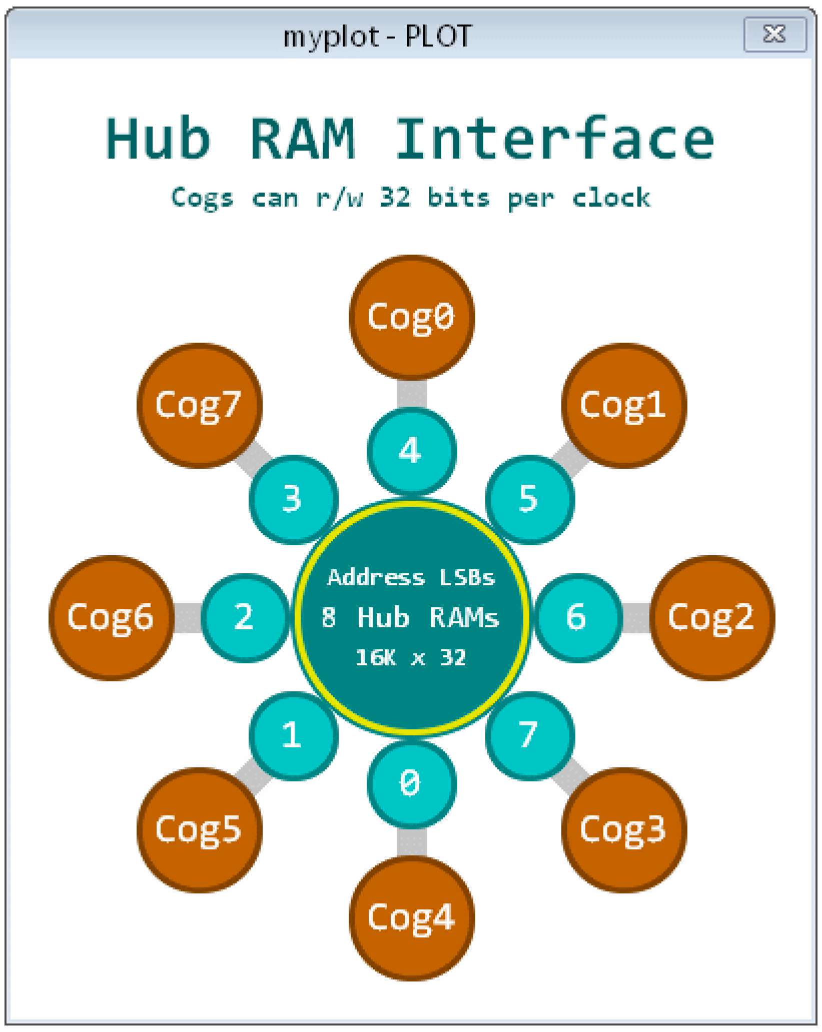
* Color is ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY.

Below, a SPECTRO display was fed ADC samples from a pin attached to a microphone. This is what verbally counting from "1" to "10" looks like, spectrally. The "1" is on the left and the "10" is on the right. The vertical distance between horizontal trend lines is glottal pitch. The larger brightness trends are vocal formants. This gives some idea of how our ears perceive speech:



PLOT Display General-purpose plotter with cartesian and polar modes

```
CON _clkfreq = 10_000_000

PUB go(): i, j, k

  debug(`plot myplot size 400 480 backcolor white update)
  debug(`myplot origin 200 200 polar -64 -16)
  k~
  repeat
    debug(`myplot clear)
    debug(`myplot set 240 0 cyan 3 text 24 3 'Hub RAM Interface')
    debug(`myplot set 210 0 text 11 3 'Cogs can r/w 32 bits per clock')

    if k & 8    'move RAMs or draw spokes?
      j++
    else
      repeat i from 0 to 7
        debug(`myplot grey 12 set 83 `(i*8) line 150 `(i*8) 15)

    debug(`myplot set 0 0 cyan 4 circle 121 yellow 7 circle 117 3)
    debug(`myplot set 20 0 white text 10 'Address LSBs')
    debug(`myplot set 0 0 text 12 1 '8 Hub RAMs')
    debug(`myplot set 20 32 text 10 '16K x 32' )

    repeat i from 0 to 7    'draw RAMs and cogs
      debug(`myplot cyan 6 set 83 `(i*8-j) circle 43 text 14 '`(i)')
      debug(`myplot cyan 4 set 83 `(i*8-j) circle 45 3)
      debug(`myplot orange 6 set 150 `(i*8) circle 61 text 13 'Cog`(i)')
      debug(`myplot orange 4 set 150 `(i*8) circle 63 3)

    debug(`myplot update `dly(30))
    k++
```



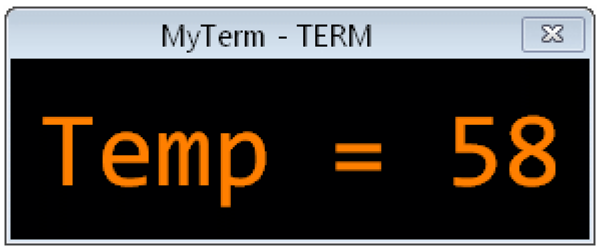
PLOT Instantiation	Description	Default
`TITLE 'string'`	Set the window caption to 'string'.	<none>
`POS left top`	Set the window position.	0, 0
`SIZE width height`	Set the display width (32..2048) and height (32..2048).	256, 256
`DOTSIZE width_and_height {height}`	Set the display pixel-width and pixel-height together, or set them independently.	1, 1
`lut1_to_rgb24`	Set the color mode.	RGB24
`LUTCOLORS rgb24 rgb24 ...`	For LUT1..LUT8 color modes, load the LUT with rgb24 colors. Use HEX_LONG_ARRAY_ to load colors.	default colors 0..7
`BACKCOLOR color`	Set the background color according to the current color mode. *	BLACK
`UPDATE`	Set UPDATE mode. The display will only be updated when fed an 'UPDATE' command.	automatic update
PLOT Feeding	**Description**	**Default**
`lut1_to_rgb24`	Set color mode.	rgb24
`LUTCOLORS rgb24 rgb24 ...`	For LUT1..LUT8 color modes, load the LUT with rgb24 colors. Use HEX_LONG_ARRAY_ to load values.	default colors 0..7
`BACKCOLOR color`	Set the background color according to the current color mode. *	BLACK
`COLOR color`	Set the drawing color according to the current color mode. Use just before TEXT to change text color. *	CYAN
`BLACK/WHITE or ORANGE/BLUE/GREEN/CYAN/ RED/MAGENTA/YELLOW/GREY {brightness}`	Set the drawing color and optional 0..15 brightness for ORANGE..GREY colors (default is 8).	CYAN
`OPACITY level`	Set the opacity level for DOT, LINE, CIRCLE, OVAL, BOX, and OBOX drawing. 0..255 = clear..opaque.	255
`PRECISE`	Toggle precise mode, where line size and (x,y) for DOT and LINE are expressed in 256ths of a pixel.	disabled
`LINESIZE size`	Set the line size in pixels for DOT and LINE drawing.	1
`ORIGIN {x_pos y_pos}`	Set the origin point to cartesian (x_pos, y_pos) or to the current (x, y) if no values are specified.	0, 0
`SET x y`	Set the drawing position to (x, y). After LINE, the endpoint becomes the new drawing position.	
`DOT {linesize {opacity}}`	Draw a dot at the current position with optional LINESIZE and OPACITY overrides.	
`LINE x y {linesize {opacity}}`	Draw a line from the current position to (x,y) with optional LINESIZE and OPACITY overrides.	
`CIRCLE diameter {linesize {opacity}}`	Draw a circle around the current position with optional line size (none/0 = solid) and OPACITY override.	
`OVAL width height {linesize {opacity}}`	Draw an oval around the current position with optional line size (none/0 = solid) and OPACITY override.	
`BOX width height {linesize {opacity}}`	Draw a box around the current position with optional line size (none/0 = solid) and OPACITY override..	
`OBOX width height x_radius y_radius {linesize {opacity}}`	Draw a rounded box around the current position with width, height, x and y radii, and optional line size (none/0 = solid) and OPACITY override.	
`TEXTSIZE size`	Set the text size (6..200).	10
`TEXTSTYLE style_YYXXUIWW`	Set the text style to %YYXXUIWW: %YY is vertical justification: %00 = middle, %10 = bottom, %11 = top.	%00000001

	%XX is horizontal justification: %00 = middle, %10 = right, %11 = left. %U is underline: %1 = underline. %I is italic: %1 = italic. %WW is weight: %00 = light, %01 = normal, %10 = bold, and %11 = heavy.	
`TEXTANGLE angle`	Set the text angle. In cartesian mode, the angle is in degrees.	0
`TEXT {size {style {angle}}} 'text'`	Draw text with overrides for size, style, and angle. To change text color, declare a color just before TEXT.	
`POLAR {twopi {offset}}`	Set polar mode, twopi value, and offset. For example, POLAR -12 -3 would be like a clock face. For a twopi value of $100000000 or -$100000000, use 0 or -1. In polar mode, (x, y) coordinates are interpreted as (length, angle).	$100000000, 0
`CARTESIAN`	Set cartesian mode. This is the default mode.	
`CLEAR`	Clear the plot to the background color.	
`UPDATE`	Update the window with the current plot. Used in UPDATE mode.	
`SAVE {WINDOW} 'filename'`	Save a bitmap file (.bmp) of either the entire window or just the display area.	
`CLOSE`	Close the window.	

* Color is a modal value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

TERM Display   Terminal for displaying text

```
CON _clkfreq = 10_000_000

PUB go() | i

  debug(`TERM MyTerm SIZE 9 1 TEXTSIZE 40)
  repeat
    repeat i from 50 to 60
      debug(`MyTerm 1 'Temp = `(i)')
      waitms(500)
```

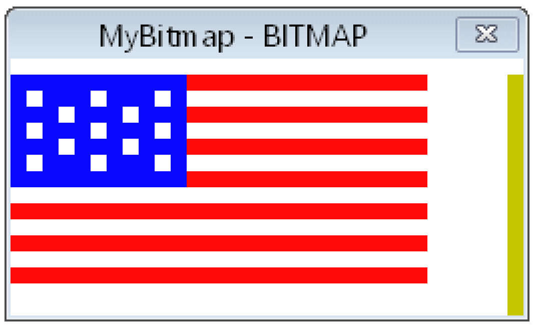


TERM Instantiation	Description	Default
`TITLE 'string'`	Set the window caption to 'string'.	<none>
`POS left top`	Set the window position.	0, 0
`SIZE columns rows`	Set the number of terminal columns (1..256) and terminal rows (1..256).	40, 20
`TEXTSIZE size`	Set the terminal text size (6..200).	editor text size
`COLOR text_color back_color ...`	Set text-color and background-color combos #0..#3. *	default colors
`BACKCOLOR color`	Set the display background color. *	BLACK
`UPDATE`	Set UPDATE mode. The display will only be updated when fed an 'UPDATE' command.	automatic update
TERM Feeding	**Description**	**Default**
`character`	0 = Clear terminal display and home cursor. 1 = Home cursor. 2 = Set column to next character value. 3 = Set row to next character value. 4 = Select color combo #0. 5 = Select color combo #1. 6 = Select color combo #2. 7 = Select color combo #3. 8 = Backspace. 9 = Tab to next 8th column. 13+10 or 13 or 10 = New line. 32..255 = Printable character.	
`'string'`	Print string.	
`CLEAR`	Clear the display to the background color.	
`UPDATE`	Update the window with the current text screen. Used in UPDATE mode.	
`SAVE {WINDOW} 'filename'`	Save a bitmap file (.bmp) of either the entire window or just the display area.	
`CLOSE`	Close the window.	

* Color is a modal value, else BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

BITMAP Display Pixel-driven bitmap

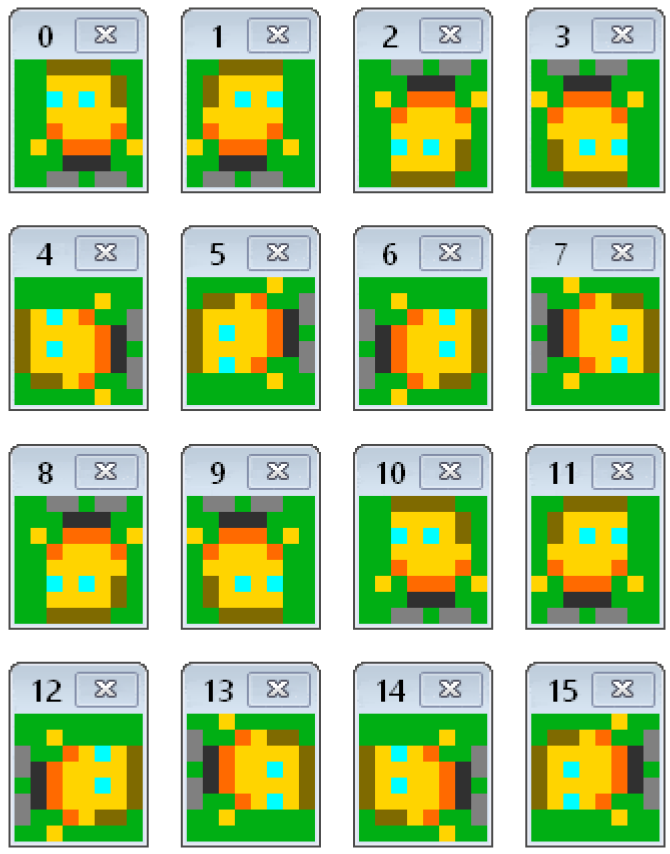
```
CON _clkfreq = 10_000_000

PUB go() | i

  debug(`bitmap MyBitmap SIZE 32 16 DOTSIZE 8 LUT2 LONGS_2BIT)
  debug(`MyBitmap TRACE 14 LUTCOLORS WHITE RED BLUE YELLOW 6)
  repeat
    debug(`MyBitmap `uhex_(flag[i++ & $1F]) `dly(100))

DAT

flag    long   %%3333333333333330
        long   %%0010101022222220
        long   %%0010101020202020
        long   %%0010101022222220
        long   %%0010101022020220
        long   %%0010101022222220
        long   %%0010101020202020
        long   %%0010101022222220
        long   %%0010101022020220
        long   %%0010101022222220
        long   %%0010101020202020
        long   %%0010101022222220
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0010101010101010
        long   %%0000000000000000
        long   %%0000000000000000
        long   %%0000000000000000
        long   %%0000000000000000
        long   %%0000000000000000
```



BITMAP Instantiation	Description	Default
`TITLE 'string'`	Set the window caption to 'string'.	<none>
`POS left top`	Set the window position.	0, 0
`SIZE x_pixels y_pixels`	Set the number of pixels in the bitmap (1..2048 for both x and y).	256, 256
`DOTSIZE x_and_y_size {y_size}`	Set the size of the displayed pixels (1..256). 'DOTSIZE 16' makes each pixel 16x16 on the final display.	1, 1
`lut1_to_rgb24`	Set the color mode. See images below.	RGB24
`LUTCOLORS rgb24 rgb24 ...`	For LUT1..LUT8 color modes, load the LUT with RGB24 colors. Use HEX_LONG_ARRAY_ to load.	default colors 0..7
`TRACE 0_to_15`	Set the pixel loading direction and whether to scroll after each line is filled. See animation below.	0
`RATE pixels_per_update`	Set the number of pixels before each display update. 'RATE -1' sets the rate to the bitmap size.	line size
`packed_data_mode`	Enable packed-data mode. See description at end of this section.	<none>
`UPDATE`	Set UPDATE mode. The display will only be updated when fed an 'UPDATE' command.	automatic update
BITMAP Feeding	**Description**	**Default**
`lut1_to_rgb24`	Change the color mode.	RGB24
`LUTCOLORS rgb24 rgb24 ...`	For LUT1..LUT8 color modes, load the LUT with rgb24 colors. Use HEX_LONG_ARRAY_ to load colors.	default colors 0..7
`TRACE 0_to_15`	Change the direction in which pixels are loaded into the bitmap. Sets the rate to the line size.	0
`RATE pixels_per_update`	Set the number of pixels before each display update. 'RATE -1' sets the rate to the bitmap size.	
`SET x_position {y_position}`	Set the current pixel-loading position. Cancels scroll mode by clearing bit 3 of TRACE.	
`SCROLL x_scroll y_scroll`	Scroll the bitmap by some number of pixels. Negative/positive values determine the direction, 0 = none.	
`CLEAR`	Clear the bitmap to zero-value pixels.	
`UPDATE`	Update the window with the current bitmap. Used in UPDATE mode.	
`SAVE {WINDOW} 'filename'`	Save a bitmap file (.bmp) of either the entire window or just the bitmap at 1x scale.	
`CLOSE`	Close the window.	

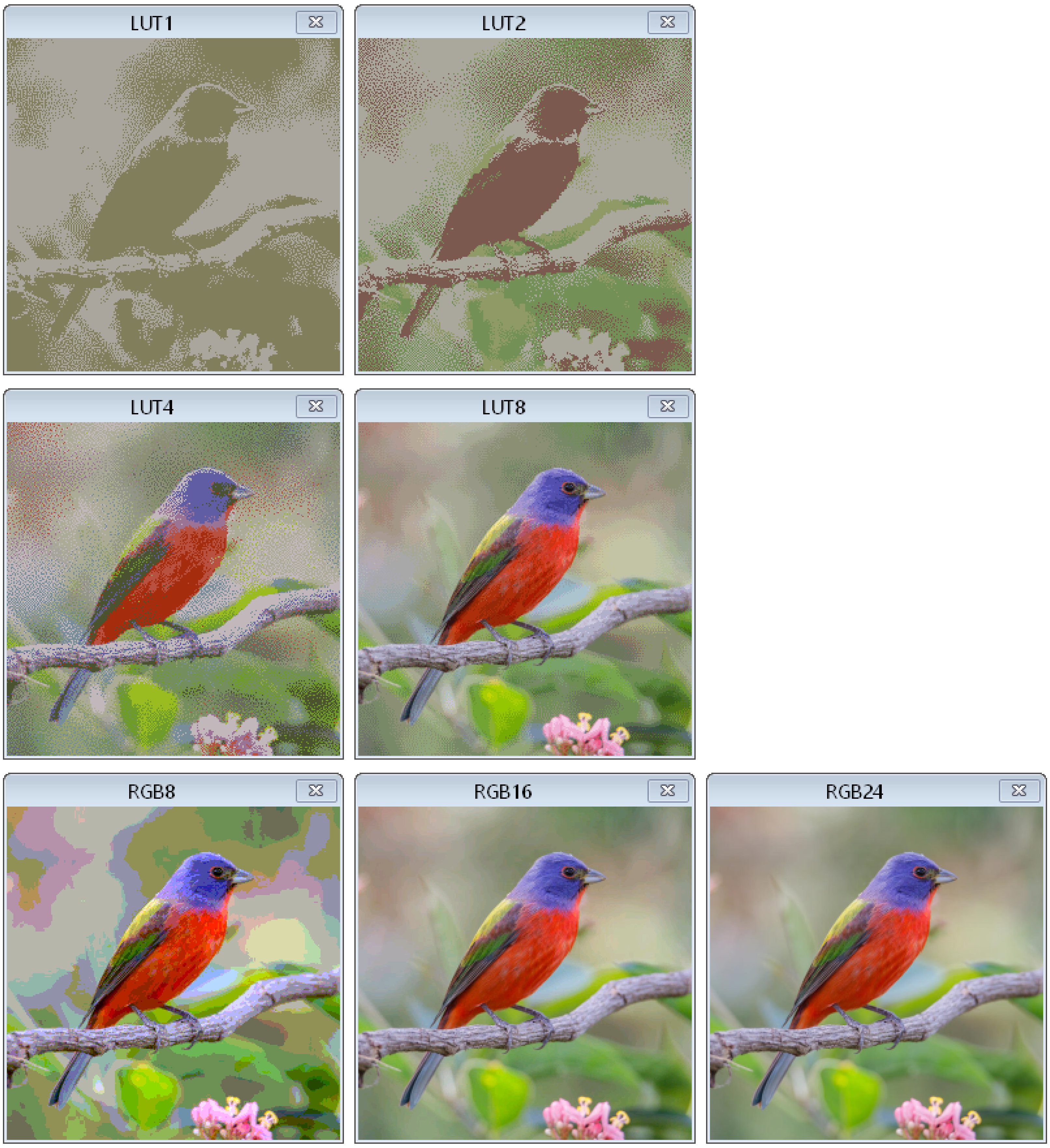
TRACE modes

Rate is set to 1 so that each pixel can be seen as it's loaded.

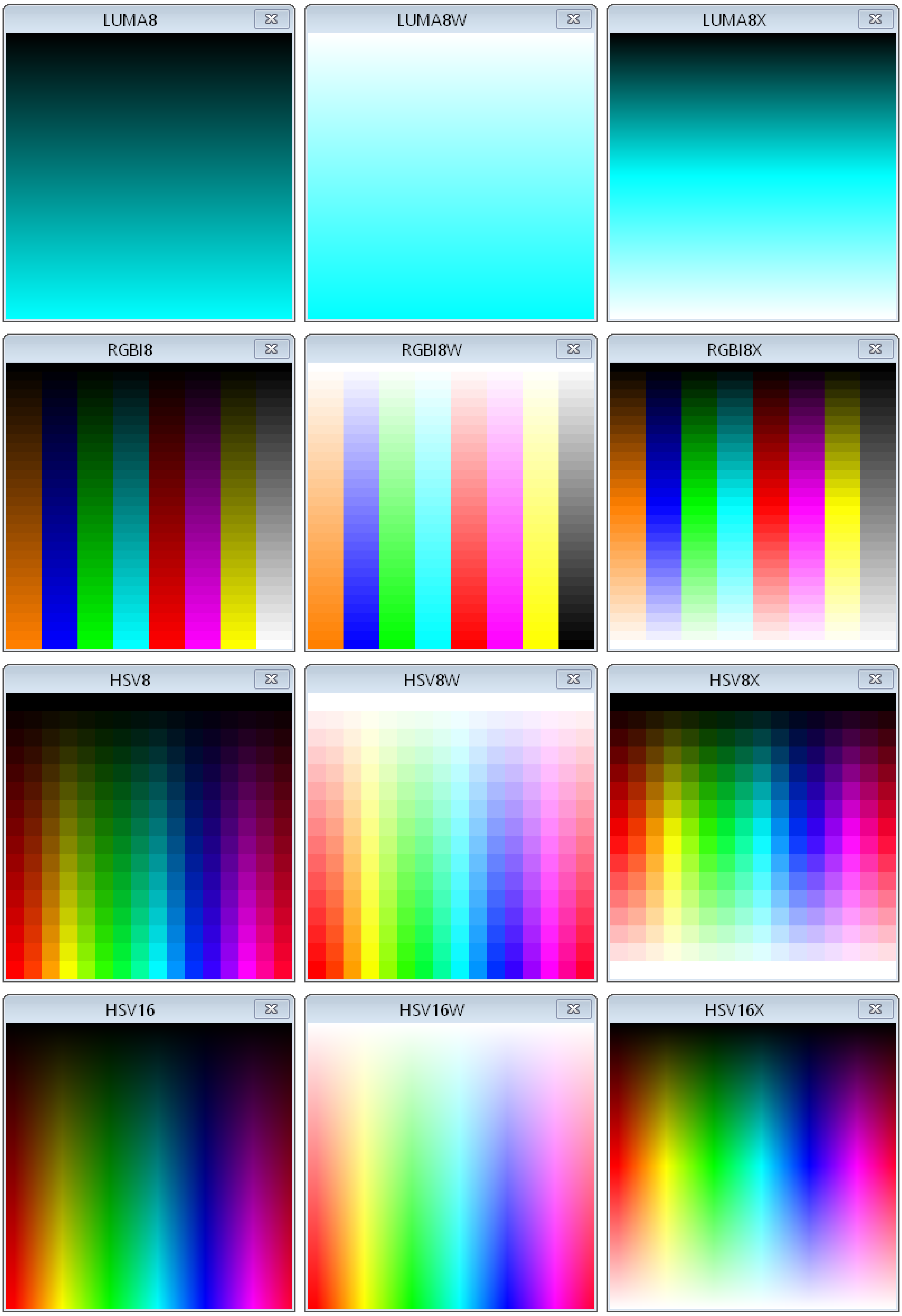


Color Mode	Bits/ Pixel	Description	Intention
`LUT1`	1	Pixel indexes LUT colors 0..1	Memory-efficient 2-color-palette graphics
`LUT2`	2	Pixel indexes LUT colors 0..3	Memory-efficient 4-color-palette graphics
`LUT4`	4	Pixel indexes LUT colors 0..15	Memory-efficient 16-color-palette graphics
`LUT8`	8	Pixel indexes LUT colors 0..255	Memory-efficient 256-color-palette graphics.
`LUMA8`	8	From black to color *	Instrumentation where luminance indicates level
`LUMA8W`	8	From white to color *	Instrumentation where saturation indicates level
`LUMA8X`	8	From black to color * to white	Instrumentation where luminance indicates level, peaking in white
`HSV8`	8	From black to color: %HHHHSSSS	16 hues with 16 luminance levels
`HSV8W`	8	From white to color: %HHHHSSSS	16 hues with 16 saturation levels, coming from white
`HSV8X`	8	From black to color to white: %HHHHSSSS	16 hues with 16 luminance levels, peaking in white
`RGBI8`	8	From black to color: %RGBIIIII	8 basic colors with 32 luminance levels
`RGBI8W`	8	From white to color: %RGBIIIII	8 basic colors with 32 saturation levels, coming from white
`RGBI8X`	8	From black to color to white: %RGBIIIII	8 basic colors with 32 luminance levels, peaking in white
`RGB8`	8	%RRRGGGBB	Byte-level RGB with 8 red, 8 green, and 4 blue levels
`HSV16`	16	From black to color: %HHHHHHHH_SSSSSSSS	256 hues with 256 luminance levels
`HSV16W`	16	From white to color: %HHHHHHHH_SSSSSSSS	256 hues with 256 saturation levels, coming from white
`HSV16X`	16	From black to color to white: %HHHHHHHH_SSSSSSSS	256 hues with 256 luminance levels, peaking in white
`RGB16`	16	%RRRRRGGG_GGGBBBBB	Word-level RGB with 32 red levels, 64 green levels, and 32 blue levels
`RGB24`	24	%RRRRRRRR_GGGGGGGG_BBBBBBBB	Full RGB with 256 levels for red, green, and blue

* Color is ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY.



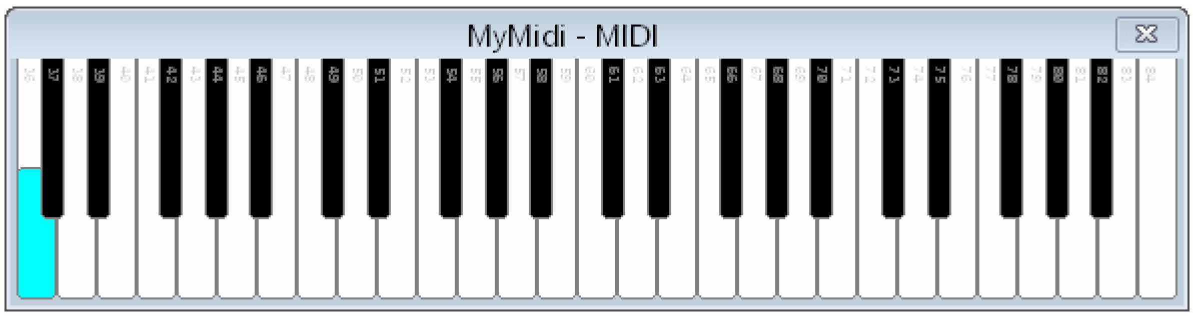

```
CON _clkfreq = 100_000_000

PUB go() | i
  debug(`bitmap a title 'LUT1'  pos 100 100 trace 2 lut1 longs_1bit alt)
  debug(`bitmap b title 'LUT2'  pos 370 100 trace 2 lut2 longs_2bit alt)
  debug(`bitmap c title 'LUT4'  pos 100 395 trace 2 lut4 longs_4bit alt)
  debug(`bitmap d title 'LUT8'  pos 370 395 trace 2 lut8 longs_8bit)
  debug(`bitmap e title 'RGB8'  pos 100 690 trace 2 rgb8)
  debug(`bitmap f title 'RGB16' pos 370 690 trace 2 rgb16)
  debug(`bitmap g title 'RGB24' pos 640 690 trace 2 rgb24)
  waitms(1000)

  showbmp("a", @image1, $8A,   2,  $800)    'send LUT1 image
  showbmp("b", @image2, $36,   4, $1000)    'send LUT2 image
  showbmp("c", @image3, $8A,  16, $2000)    'send LUT4 image
  showbmp("d", @image4, $36, 256, $4000)    'send LUT8 image

  i := @image5 + $36  'send RGB8/RGB16/RGB24 images from the same 24-bpp file
  repeat $10000
    debug(`e `uhex_(byte[i+0] >> 6 + byte[i+1] >> 5 << 2 + byte[i+2] >> 5 << 5 ))
    debug(`f `uhex_(byte[i+0] >> 3 + byte[i+1] >> 2 << 5 + byte[i+2] >> 3 << 11))
    debug(`g `uhex_(byte[i+0]      + byte[i+1] << 8      + byte[i+2] << 16     ))
    i += 3

PRI showbmp(letter, image_address, lut_offset, lut_size, image_longs) | i
  image_address += lut_offset
  debug(``#(letter) lutcolors `uhex_long_array_(image_address, lut_size))
  image_address += lut_size << 2 - 4
  repeat image_longs
    debug(``#(letter) `uhex_(long[image_address += 4]))

DAT
image1  file "bird_lut1.bmp"
image2  file "bird_lut2.bmp"
image3  file "bird_lut4.bmp"
image4  file "bird_lut8.bmp"
image5  file "bird_rgb24.bmp"
```




```
CON _clkfreq = 100_000_000

PUB go() | i
  debug(`bitmap a title 'LUMA8'  pos 100 100 size 1 256 dotsize 256 1 luma8  cyan)
  debug(`bitmap b title 'LUMA8W' pos 370 100 size 1 256 dotsize 256 1 luma8w cyan)
  debug(`bitmap c title 'LUMA8X' pos 640 100 size 1 256 dotsize 256 1 luma8x cyan)
  debug(`bitmap d title 'RGBI8'  pos 100 395 size 8 32 dotsize 32 8 trace 4 rgbi8)
  debug(`bitmap e title 'RGBI8W' pos 370 395 size 8 32 dotsize 32 8 trace 4 rgbi8w)
  debug(`bitmap f title 'RGBI8X' pos 640 395 size 8 32 dotsize 32 8 trace 4 rgbi8x)
  debug(`bitmap g title 'HSV8'   pos 100 690 size 16 16 trace 4 dotsize 16 hsv8)
  debug(`bitmap h title 'HSV8W'  pos 370 690 size 16 16 trace 4 dotsize 16 hsv8w)
  debug(`bitmap i title 'HSV8X'  pos 640 690 size 16 16 trace 4 dotsize 16 hsv8x)
  debug(`bitmap j title 'HSV16'  pos 100 985 size 256 256 trace 4 hsv16)
  debug(`bitmap k title 'HSV16W' pos 370 985 size 256 256 trace 4 hsv16w)
  debug(`bitmap l title 'HSV16X' pos 640 985 size 256 256 trace 4 hsv16x)
  waitms(1000)
  repeat i from 0 to 255                    'feed 8-bit displays
    debug(`a b c d e f g h i `uhex_(i))
  repeat i from 0 to 65535                  'feed 16-bit displays
    debug(`j k l `uhex_(i))
```

MIDI Display   MIDI keyboard for viewing note-on/off status with velocity

```
CON _clkfreq = 10_000_000

PUB go() | i

  debug(`midi MyMidi size 3 range 36 84)
  repeat
    repeat i from 36 to 84
      debug(`MyMidi $90 `(i, getrnd() & $7F))
      waitms(150)
      debug(`MyMidi $80 `(i, 0))
```

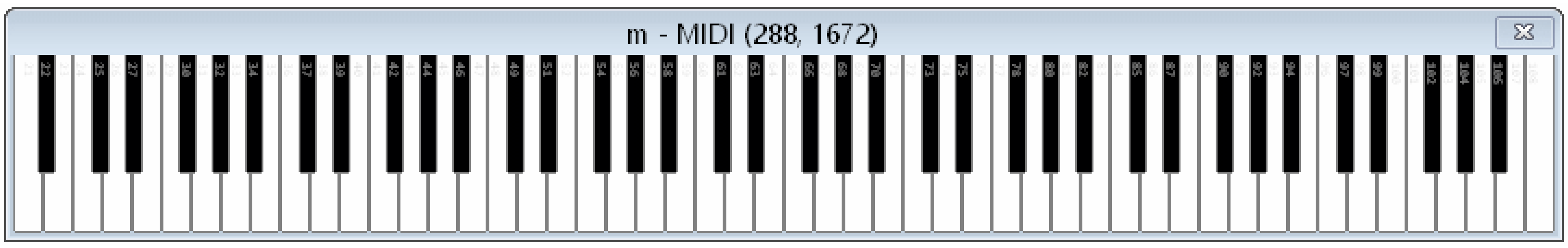


MIDI Instantiation	Description	Default
TITLE 'string'	Set the window caption to 'string'.	<none>
POS left top	Set the window position.	0, 0
SIZE keyboard_size	Set the size of the MIDI keyboard display (1..50).	4
RANGE first_key last_key	Set the first and last MIDI key numbers (0..127).	21, 108 (88 keys)
CHANNEL channel_number	Set the MIDI channel number to observe (0..15).	0
COLOR white_key black_key	Set the 'ON' colors for white and black keys. *	CYAN, MAGENTA
MIDI Feeding	**Description**	**Default**
byte	If $90 + channel then NOTE_ON mode, else if $80 + channel then NOTE_OFF mode. If NOTE_ON mode then receive a key ($00..$7F) and then its velocity ($00..$7F), update display. If NOTE_OFF mode then receive a key ($00..$7F) and then its velocity ($00..$7F), update display.	
CLEAR	Clear all notes.	
SAVE {WINDOW} 'filename'	Save a bitmap file (.bmp) of either the entire window or just the display area.	
CLOSE	Close the window.	

* Color is BLACK / WHITE or ORANGE / BLUE / GREEN / CYAN / RED / MAGENTA / YELLOW / GREY followed by an optional 0..15 for brightness (default is 8).

Here is a PASM program which receives MIDI serial on P16 and sends it to the MIDI display:

```
CON     _clkfreq        = 10_000_000
        rxpin           = 16

DAT     org

        asmclk

        debug   (`midi m size 2)

        wrpin   #%11111_0,#rxpin
        wxpin   ##(clkfreq_/31250) << 16 + 8-1, #rxpin
        drvl    #rxpin

.wait   testp   #rxpin  wc
  if_nc jmp     #.wait

        rdpin   x,#rxpin
        shr     x,#32-8

        debug   ("`m ", uhex_byte_(x))

        jmp     #.wait

x       res     1
```



Packed-Data Modes

Packed-data modes are used to efficiently convey sub-byte data types, by having the host side unpack them from bytes, words, or longs it receives. As well, bytes can be sent within words and longs, and words can be sent within longs for some efficiency improvement.

To establish packed-data operation, you must specify one of the modes listed below, followed by optional 'ALT' and 'SIGNED' keywords:

```
packed_mode {ALT} {SIGNED}
```

The **ALT** keyword will cause bits, double-bits, or nibbles, within each byte sent, to be reordered on the host side, within each byte. This simplifies cases where the raw data you are sending has its bitfields out-of-order with respect to the DEBUG display you are using. This is most-likely to be needed for bitmap data that was composed in standard formats.

The **SIGNED** keyword will cause all unpacked data values to be sign-extended on the host side.

Packed-Data Modes	Descriptions	Final Values	Final Values if SIGNED
`LONGS_1BIT`	Each value received is translated into 32 separate 1-bit values, starting from the LSB of the received value.	0..1	-1..0
`LONGS_2BIT`	Each value received is translated into 16 separate 2-bit values, starting from the LSBs of the received value.	0..3	-2..1
`LONGS_4BIT`	Each value received is translated into 8 separate 4-bit values, starting from the LSBs of the received value.	0..15	-8..7
`LONGS_8BIT`	Each value received is translated into 4 separate 8-bit values, starting from the LSBs of the received value.	0..255	-128..127
`LONGS_16BIT`	Each value received is translated into 2 separate 16-bit values, starting from the LSBs of the received value.	0..65,535	-32,768..32,767
`WORDS_1BIT`	Each value received is translated into 16 separate 1-bit values, starting from the LSB of the received value.	0..1	-1..0
`WORDS_2BIT`	Each value received is translated into 8 separate 2-bit values, starting from the LSBs of the received value.	0..3	-2..1
`WORDS_4BIT`	Each value received is translated into 4 separate 4-bit values, starting from the LSBs of the received value.	0..15	-8..7
`WORDS_8BIT`	Each value received is translated into 2 separate 8-bit values, starting from the LSBs of the received value.	0..255	-128..127
`BYTES_1BIT`	Each value received is translated into 8 separate 1-bit values, starting from the LSB of the received value.	0..1	-1..0
`BYTES_2BIT`	Each value received is translated into 4 separate 2-bit values, starting from the LSBs of the received value.	0..3	-2..1
`BYTES_4BIT`	Each value received is translated into 2 separate 4-bit values, starting from the LSBs of the received value.	0..15	-8..7

Built-In Symbols for Smart Pin Configuration

Smart Pin Symbol Value	Symbol Name	Details
A Input Polarity	**(pick one)**	
%0000_0000_000_0000000000000_00_00000_0	P_TRUE_A (default)	True A input
%1000_0000_000_0000000000000_00_00000_0	P_INVERT_A	Invert A input
A Input Selection	**(pick one)**	
%0000_0000_000_0000000000000_00_00000_0	P_LOCAL_A (default)	Select local pin for A input
%0001_0000_000_0000000000000_00_00000_0	P_PLUS1_A	Select pin+1 for A input
%0010_0000_000_0000000000000_00_00000_0	P_PLUS2_A	Select pin+2 for A input
%0011_0000_000_0000000000000_00_00000_0	P_PLUS3_A	Select pin+3 for A input
%0100_0000_000_0000000000000_00_00000_0	P_OUTBIT_A	Select OUT bit for A input
%0101_0000_000_0000000000000_00_00000_0	P_MINUS3_A	Select pin-3 for A input
%0110_0000_000_0000000000000_00_00000_0	P_MINUS2_A	Select pin-2 for A input
%0111_0000_000_0000000000000_00_00000_0	P_MINUS1_A	Select pin-1 for A input
B Input Polarity	**(pick one)**	
%0000_0000_000_0000000000000_00_00000_0	P_TRUE_B (default)	True B input
%0000_1000_000_0000000000000_00_00000_0	P_INVERT_B	Invert B input
B Input Selection	**(pick one)**	
%0000_0000_000_0000000000000_00_00000_0	P_LOCAL_B (default)	Select local pin for B input
%0000_0001_000_0000000000000_00_00000_0	P_PLUS1_B	Select pin+1 for B input
%0000_0010_000_0000000000000_00_00000_0	P_PLUS2_B	Select pin+2 for B input
%0000_0011_000_0000000000000_00_00000_0	P_PLUS3_B	Select pin+3 for B input
%0000_0100_000_0000000000000_00_00000_0	P_OUTBIT_B	Select OUT bit for B input
%0000_0101_000_0000000000000_00_00000_0	P_MINUS3_B	Select pin-3 for B input
%0000_0110_000_0000000000000_00_00000_0	P_MINUS2_B	Select pin-2 for B input
%0000_0111_000_0000000000000_00_00000_0	P_MINUS1_B	Select pin-1 for B input
A, B Input Logic	**(pick one)**	
%0000_0000_000_0000000000000_00_00000_0	P_PASS_AB (default)	Select A, B
%0000_0000_001_0000000000000_00_00000_0	P_AND_AB	Select A & B, B
%0000_0000_010_0000000000000_00_00000_0	P_OR_AB	Select A | B, B
%0000_0000_011_0000000000000_00_00000_0	P_XOR_AB	Select A ^ B, B
%0000_0000_100_0000000000000_00_00000_0	P_FILT0_AB	Select FILT0 settings for A, B
%0000_0000_101_0000000000000_00_00000_0	P_FILT1_AB	Select FILT1 settings for A, B
%0000_0000_110_0000000000000_00_00000_0	P_FILT2_AB	Select FILT2 settings for A, B
%0000_0000_111_0000000000000_00_00000_0	P_FILT3_AB	Select FILT3 settings for A, B
Low-Level Pin Modes	**(pick one)**	
Logic/Schmitt/Comparator Input Modes		
%0000_0000_000_0000000000000_00_00000_0	P_LOGIC_A (default)	Logic level A → IN, output OUT
%0000_0000_000_0001000000000_00_00000_0	P_LOGIC_A_FB	Logic level A → IN, output feedback
%0000_0000_000_0010000000000_00_00000_0	P_LOGIC_B_FB	Logic level B → IN, output feedback
%0000_0000_000_0011000000000_00_00000_0	P_SCHMITT_A	Schmitt trigger A → IN, output OUT
%0000_0000_000_0100000000000_00_00000_0	P_SCHMITT_A_FB	Schmitt trigger A → IN, output feedback
%0000_0000_000_0101000000000_00_00000_0	P_SCHMITT_B_FB	Schmitt trigger B → IN, output feedback
%0000_0000_000_0110000000000_00_00000_0	P_COMPARE_AB	A > B → IN, output OUT
%0000_0000_000_0111000000000_00_00000_0	P_COMPARE_AB_FB	A > B → IN, output feedback
%xxxx_xxxx_xxx_xxxxSIOHHHLLL_xx_xxxxx_x		Sync mode, IN/output polarity, high/low drive
ADC Input Modes		
%0000_0000_000_1000000000000_00_00000_0	P_ADC_GIO	ADC GIO → IN, output OUT
%0000_0000_000_1000010000000_00_00000_0	P_ADC_VIO	ADC VIO → IN, output OUT

%0000_0000_000_1000100000000_00_00000_0	P_ADC_FLOAT	ADC FLOAT → IN, output OUT
%0000_0000_000_1000110000000_00_00000_0	P_ADC_1X	ADC 1x → IN, output OUT
%0000_0000_000_1001000000000_00_00000_0	P_ADC_3X	ADC 3.16x → IN, output OUT
%0000_0000_000_1001010000000_00_00000_0	P_ADC_10X	ADC 10x → IN, output OUT
%0000_0000_000_1001100000000_00_00000_0	P_ADC_30X	ADC 31.6x → IN, output OUT
%0000_0000_000_1001110000000_00_00000_0	P_ADC_100X	ADC 100x → IN, output OUT
%xxxx_xxxx_xxx_xxxxxxOHHHLLL_xx_xxxxx_x		O = output polarity, HHH/LLL = high/low drive
DAC Output Modes		**DIR enables output, OUT enables ADC**
%0000_0000_000_1010000000000_00_00000_0	P_DAC_990R_3V	DAC 990Ω, 3.3V peak, ADC 1x → IN
%0000_0000_000_1010100000000_00_00000_0	P_DAC_600R_2V	DAC 600Ω, 2.0V peak, ADC 1x → IN
%0000_0000_000_1011000000000_00_00000_0	P_DAC_124R_3V	DAC 123.75Ω, 3.3V peak, ADC 1x → IN
%0000_0000_000_1011100000000_00_00000_0	P_DAC_75R_2V	DAC 75Ω, 2.0V peak, ADC 1x → IN
%xxxx_xxxx_xxx_xxxxxDDDDDDDD_xx_xxxxx_x		DDDDDDDD = 8-bit DAC value
Level-Comparison Modes		**DIR enables output (1.5kΩ drive)**
%0000_0000_000_1100000000000_00_00000_0	P_LEVEL_A	A > Level → IN, output OUT
%0000_0000_000_1101000000000_00_00000_0	P_LEVEL_A_FBN	A > Level → IN, output negative feedback
%0000_0000_000_1110000000000_00_00000_0	P_LEVEL_B_FBP	B > Level → IN, output positive feedback
%0000_0000_000_1111000000000_00_00000_0	P_LEVEL_B_FBN	B > Level → IN, output negative feedback
%xxxx_xxxx_xxx_xxxxSLLLLLLLL_xx_xxxxx_x		S = Synchronous, LLLLLLLL = 8-bit Level
Low-Level Pin Sub-Modes		
Sync Mode	(pick one)	(for Logic/Schmitt/Comparator/Level modes)
%xxxx_xxxx_xxx_xxxxSxxxxxxxx_xx_xxxxx_x		Sync mode bit
%0000_0000_000_0000000000000_00_00000_0	P_ASYNC_IO (default)	Select asynchronous I/O
%0000_0000_000_0000100000000_00_00000_0	P_SYNC_IO	Select synchronous I/O
IN Polarity	(pick one)	(for Logic/Schmitt/Comparator modes)
%xxxx_xxxx_xxx_xxxxxIxxxxxxx_xx_xxxxx_x		IN polarity bit
%0000_0000_000_0000000000000_00_00000_0	P_TRUE_IN (default)	True IN bit
%0000_0000_000_0000010000000_00_00000_0	P_INVERT_IN	Invert IN bit
Output Polarity	(pick one)	(for Logic/Schmitt/Comparator/ADC modes)
%xxxx_xxxx_xxx_xxxxxxOxxxxxx_xx_xxxxx_x		Output polarity bit
%0000_0000_000_0000000000000_00_00000_0	P_TRUE_OUTPUT (default)	Select true output
%0000_0000_000_0000001000000_00_00000_0	P_INVERT_OUTPUT	Select inverted output
Drive-High Strength	(pick one)	(for Logic/Schmitt/Comparator/ADC modes)
%xxxx_xxxx_xxx_xxxxxxxHHHxxx_xx_xxxxx_x		Drive-high selector bits
%0000_0000_000_0000000000000_00_00000_0	P_HIGH_FAST (default)	Drive high fast (30mA)
%0000_0000_000_0000000001000_00_00000_0	P_HIGH_1K5	Drive high 1.5kΩ
%0000_0000_000_0000000010000_00_00000_0	P_HIGH_15K	Drive high 15kΩ
%0000_0000_000_0000000011000_00_00000_0	P_HIGH_150K	Drive high 150kΩ
%0000_0000_000_0000000100000_00_00000_0	P_HIGH_1MA	Drive high 1mA
%0000_0000_000_0000000101000_00_00000_0	P_HIGH_100UA	Drive high 100µA
%0000_0000_000_0000000110000_00_00000_0	P_HIGH_10UA	Drive high 10µA
%0000_0000_000_0000000111000_00_00000_0	P_HIGH_FLOAT	Float high
Drive-Low Strength	(pick one)	(for Logic/Schmitt/Comparator/ADC modes)
%xxxx_xxxx_xxx_xxxxxxxxxxLLL_xx_xxxxx_x		Drive-low selector bits
%0000_0000_000_0000000000000_00_00000_0	P_LOW_FAST (default)	Drive low fast (30mA)
%0000_0000_000_0000000000001_00_00000_0	P_LOW_1K5	Drive low 1.5kΩ
%0000_0000_000_0000000000010_00_00000_0	P_LOW_15K	Drive low 15kΩ
%0000_0000_000_0000000000011_00_00000_0	P_LOW_150K	Drive low 150kΩ
%0000_0000_000_0000000000100_00_00000_0	P_LOW_1MA	Drive low 1mA
%0000_0000_000_0000000000101_00_00000_0	P_LOW_100UA	Drive low 100µA
%0000_0000_000_0000000000110_00_00000_0	P_LOW_10UA	Drive low 10µA

%0000_0000_000_0000000000111_00_00000_0	P_LOW_FLOAT	Float low
DIR/OUT Control	**(pick one)**	
%0000_0000_000_0000000000000_00_00000_0	P_TT_00 (default)	TT = %00
%0000_0000_000_0000000000000_01_00000_0	P_TT_01	TT = %01
%0000_0000_000_0000000000000_10_00000_0	P_TT_10	TT = %10
%0000_0000_000_0000000000000_11_00000_0	P_TT_11	TT = %11
%0000_0000_000_0000000000000_01_00000_0	P_OE	Enable output in smart pin mode
%0000_0000_000_0000000000000_01_00000_0	P_CHANNEL	Enable DAC channel in non-smart pin DAC mode
%0000_0000_000_0000000000000_10_00000_0	P_BITDAC	Enable BITDAC for non-smart pin DAC mode
Smart Pin Modes	**(pick one)**	
%0000_0000_000_0000000000000_00_00000_0	P_NORMAL (default)	Normal mode (not smart pin mode)
%0000_0000_000_0000000000000_00_00001_0	P_REPOSITORY	Long repository (non-DAC mode)
%0000_0000_000_0000000000000_00_00001_0	P_DAC_NOISE	DAC Noise (DAC mode)
%0000_0000_000_0000000000000_00_00010_0	P_DAC_DITHER_RND	DAC 16-bit random dither (DAC mode)
%0000_0000_000_0000000000000_00_00011_0	P_DAC_DITHER_PWM	DAC 16-bit PWM dither (DAC mode)
%0000_0000_000_0000000000000_00_00100_0	P_PULSE	Pulse/cycle output
%0000_0000_000_0000000000000_00_00101_0	P_TRANSITION	Transition output
%0000_0000_000_0000000000000_00_00110_0	P_NCO_FREQ	NCO frequency output
%0000_0000_000_0000000000000_00_00111_0	P_NCO_DUTY	NCO duty output
%0000_0000_000_0000000000000_00_01000_0	P_PWM_TRIANGLE	PWM triangle output
%0000_0000_000_0000000000000_00_01001_0	P_PWM_SAWTOOTH	PWM sawtooth output
%0000_0000_000_0000000000000_00_01010_0	P_PWM_SMPS	PWM switch-mode power supply I/O
%0000_0000_000_0000000000000_00_01011_0	P_QUADRATURE	A-B quadrature encoder input
%0000_0000_000_0000000000000_00_01100_0	P_REG_UP	Inc on A-rise when B-high
%0000_0000_000_0000000000000_00_01101_0	P_REG_UP_DOWN	Inc on A-rise when B-high, dec on A-rise when B-low
%0000_0000_000_0000000000000_00_01110_0	P_COUNT_RISES	Inc on A-rise, optionally dec on B-rise
%0000_0000_000_0000000000000_00_01111_0	P_COUNT_HIGHS	Inc on A-high, optionally dec on B-high
%0000_0000_000_0000000000000_00_10000_0	P_STATE_TICKS	For A-low and A-high states, count ticks
%0000_0000_000_0000000000000_00_10001_0	P_HIGH_TICKS	For A-high states, count ticks
%0000_0000_000_0000000000000_00_10010_0	P_EVENTS_TICKS	For X A-highs/rises/edges, count ticks / Timeout on X ticks of no A-high/rise/edge
%0000_0000_000_0000000000000_00_10011_0	P_PERIODS_TICKS	For X periods of A, count ticks
%0000_0000_000_0000000000000_00_10100_0	P_PERIODS_HIGHS	For X periods of A, count highs
%0000_0000_000_0000000000000_00_10101_0	P_COUNTER_TICKS	For periods of A in X+ ticks, count ticks
%0000_0000_000_0000000000000_00_10110_0	P_COUNTER_HIGHS	For periods of A in X+ ticks, count highs
%0000_0000_000_0000000000000_00_10111_0	P_COUNTER_PERIODS	For periods of A in X+ ticks, count periods
%0000_0000_000_0000000000000_00_11000_0	P_ADC	ADC sample/filter/capture, internally clocked
%0000_0000_000_0000000000000_00_11001_0	P_ADC_EXT	ADC sample/filter/capture, externally clocked
%0000_0000_000_0000000000000_00_11010_0	P_ADC_SCOPE	ADC scope with trigger
%0000_0000_000_0000000000000_00_11011_0	P_USB_PAIR	USB pin pair
%0000_0000_000_0000000000000_00_11100_0	P_SYNC_TX	Synchronous serial transmit
%0000_0000_000_0000000000000_00_11101_0	P_SYNC_RX	Synchronous serial receive
%0000_0000_000_0000000000000_00_11110_0	P_ASYNC_TX	Asynchronous serial transmit
%0000_0000_000_0000000000000_00_11111_0	P_ASYNC_RX	Asynchronous serial receive

Built-In Symbols for Streamer Modes

Streamer Symbol Value	Symbol Name
#Immediate → LUT → Pins / DACs	
%0000_0000_0000_0000 << 16 %0000_DDDD_EPPP_BBBB << 16	X_IMM_32X1_LUT
%0001_0000_0000_0000 << 16 %0001_DDDD_EPPP_BBBB << 16	X_IMM_16X2_LUT
%0010_0000_0000_0000 << 16 %0010_DDDD_EPPP_BBBB << 16	X_IMM_8X4_LUT
%0011_0000_0000_0000 << 16 %0011_DDDD_EPPP_BBBB << 16	X_IMM_4X8_LUT
#Immediate → Pins / DACs	
%0100_0000_0000_0000 << 16 %0100_DDDD_EPPP_PPPA << 16	X_IMM_32X1_1DAC1
%0101_0000_0000_0000 << 16 %0101_DDDD_EPPP_PP0A << 16	X_IMM_16X2_2DAC1
%0101_0000_0000_0010 << 16 %0101_DDDD_EPPP_PP1A << 16	X_IMM_16X2_1DAC2
%0110_0000_0000_0000 << 16 %0110_DDDD_EPPP_P00A << 16	X_IMM_8X4_4DAC1
%0110_0000_0000_0010 << 16 %0110_DDDD_EPPP_P01A << 16	X_IMM_8X4_2DAC2
%0110_0000_0000_0100 << 16 %0110_DDDD_EPPP_P10A << 16	X_IMM_8X4_1DAC4
%0110_0000_0000_0110 << 16 %0110_DDDD_EPPP_0110 << 16	X_IMM_4X8_4DAC2
%0110_0000_0000_0111 << 16 %0110_DDDD_EPPP_0111 << 16	X_IMM_4X8_2DAC4
%0110_0000_0000_1110 << 16 %0110_DDDD_EPPP_1110 << 16	X_IMM_4X8_1DAC8
%0110_0000_0000_1111 << 16 %0110_DDDD_EPPP_1111 << 16	X_IMM_2X16_4DAC4
%0111_0000_0000_0000 << 16 %0111_DDDD_EPPP_0000 << 16	X_IMM_2X16_2DAC8
%0111_0000_0000_0001 << 16 %0111_DDDD_EPPP_0001 << 16	X_IMM_1X32_4DAC8
RDFAST → LUT → Pins / DACs	
%0111_0000_0000_0010 << 16 %0111_DDDD_EPPP_001A << 16	X_RFLONG_32X1_LUT
%0111_0000_0000_0100 << 16 %0111_DDDD_EPPP_010A << 16	X_RFLONG_16X2_LUT
%0111_0000_0000_0110 << 16 %0111_DDDD_EPPP_011A << 16	X_RFLONG_8X4_LUT
%0111_0000_0000_1000 << 16 %0111_DDDD_EPPP_1000 << 16	X_RFLONG_4X8_LUT
RDFAST → Pins / DACs	
%1000_0000_0000_0000 << 16 %1000_DDDD_EPPP_PPPA << 16	X_RFBYTE_1P_1DAC1
%1001_0000_0000_0000 << 16 %1001_DDDD_EPPP_PP0A << 16	X_RFBYTE_2P_2DAC1
%1001_0000_0000_0010 << 16 %1001_DDDD_EPPP_PP1A << 16	X_RFBYTE_2P_1DAC2
%1010_0000_0000_0000 << 16 %1010_DDDD_EPPP_P00A << 16	X_RFBYTE_4P_4DAC1
%1010_0000_0000_0010 << 16 %1010_DDDD_EPPP_P01A << 16	X_RFBYTE_4P_2DAC2
%1010_0000_0000_0100 << 16 %1010_DDDD_EPPP_P10A << 16	X_RFBYTE_4P_1DAC4
%1010_0000_0000_0110 << 16 %1010_DDDD_EPPP_0110 << 16	X_RFBYTE_8P_4DAC2
%1010_0000_0000_0111 << 16 %1010_DDDD_EPPP_0111 << 16	X_RFBYTE_8P_2DAC4
%1010_0000_0000_1110 << 16 %1010_DDDD_EPPP_1110 << 16	X_RFBYTE_8P_1DAC8
%1010_0000_0000_1111 << 16	X_RFWORD_16P_4DAC4

%1010_DDDD_EPPP_1111 << 16	
%1011_0000_0000_0000 << 16 %1011_DDDD_EPPP_0000 << 16	X_RFWORD_16P_2DAC8
%1011_0000_0000_0001 << 16 %1011_DDDD_EPPP_0001 << 16	X_RFLONG_32P_4DAC8
RDFAST → RGB → Pins / DACs	
%1011_0000_0000_0010 << 16 %1011_DDDD_EPPP_0010 << 16	X_RFBYTE_LUMA8
%1011_0000_0000_0011 << 16 %1011_DDDD_EPPP_0011 << 16	X_RFBYTE_RGBI8
%1011_0000_0000_0100 << 16 %1011_DDDD_EPPP_0100 << 16	X_RFBYTE_RGB8
%1011_0000_0000_0101 << 16 %1011_DDDD_EPPP_0101 << 16	X_RFWORD_RGB16
%1011_0000_0000_0110 << 16 %1011_DDDD_EPPP_0110 << 16	X_RFLONG_RGB24
Pins → DACs / WRFAST	
%1100_0000_0000_0000 << 16 %1100_DDDD_WPPP_PPPA << 16	X_1P_1DAC1_WFBYTE
%1101_0000_0000_0000 << 16 %1101_DDDD_WPPP_PP0A << 16	X_2P_2DAC1_WFBYTE
%1101_0000_0000_0010 << 16 %1101_DDDD_WPPP_PP1A << 16	X_2P_1DAC2_WFBYTE
%1110_0000_0000_0000 << 16 %1110_DDDD_WPPP_P00A << 16	X_4P_4DAC1_WFBYTE
%1110_0000_0000_0010 << 16 %1110_DDDD_WPPP_P01A << 16	X_4P_2DAC2_WFBYTE
%1110_0000_0000_0100 << 16 %1110_DDDD_WPPP_P10A << 16	X_4P_1DAC4_WFBYTE
%1110_0000_0000_0110 << 16 %1110_DDDD_WPPP_0110 << 16	X_8P_4DAC2_WFBYTE
%1110_0000_0000_0111 << 16 %1110_DDDD_WPPP_0111 << 16	X_8P_2DAC4_WFBYTE
%1110_0000_0000_1110 << 16 %1110_DDDD_WPPP_1110 << 16	X_8P_1DAC8_WFBYTE
%1110_0000_0000_1111 << 16 %1110_DDDD_WPPP_1111 << 16	X_16P_4DAC4_WFWORD
%1111_0000_0000_0000 << 16 %1111_DDDD_WPPP_0000 << 16	X_16P_2DAC8_WFWORD
%1111_0000_0000_0001 << 16 %1111_DDDD_WPPP_0001 << 16	X_32P_4DAC8_WFLONG
ADCs / Pins → DACs / WRFAST	
%1111_0000_0000_0010 << 16 %1111_DDDD_W000_0010 << 16	X_1ADC8_0P_1DAC8_WFBYTE
%1111_0000_0000_0011 << 16 %1111_DDDD_WPPP_0011 << 16	X_1ADC8_8P_2DAC8_WFWORD
%1111_0000_0000_0100 << 16 %1111_DDDD_W000_0100 << 16	X_2ADC8_0P_2DAC8_WFWORD
%1111_0000_0000_0101 << 16 %1111_DDDD_WPPP_0101 << 16	X_2ADC8_16P_4DAC8_WFLONG
%1111_0000_0000_0110 << 16 %1111_DDDD_W000_0110 << 16	X_4ADC8_0P_4DAC8_WFLONG
DDS / Goertzel	
%1111_0000_0000_0111 << 16 %1111_DDDD_0PPP_P111 << 16	X_DDS_GOERTZEL_SINC1
%1111_0000_1000_0111 << 16 %1111_DDDD_1PPP_P111 << 16	X_DDS_GOERTZEL_SINC2
Sub-Fields	
DAC Channel Outputs	
%xxxx_DDDD_xxxx_xxxx << 16 %0000_0000_0000_0000 << 16 %0000_0001_0000_0000 << 16 %0000_0010_0000_0000 << 16 %0000_0011_0000_0000 << 16 %0000_0100_0000_0000 << 16 %0000_0101_0000_0000 << 16 %0000_0110_0000_0000 << 16 %0000_0111_0000_0000 << 16 %0000_1000_0000_0000 << 16 %0000_1001_0000_0000 << 16	 X_DACS_OFF (default) X_DACS_0_0_0_0 X_DACS_X_X_0_0 X_DACS_0_0_X_X X_DACS_X_X_X_0 X_DACS_X_X_0_X X_DACS_X_0_X_X X_DACS_0_X_X_X X_DACS_0N0_0N0 X_DACS_X_X_0N0

%0000_1010_0000_0000 << 16 %0000_1011_0000_0000 << 16 %0000_1100_0000_0000 << 16 %0000_1101_0000_0000 << 16 %0000_1110_0000_0000 << 16 %0000_1111_0000_0000 << 16	X_DACS_0N0_X_X X_DACS_1_0_1_0 X_DACS_X_X_1_0 X_DACS_1_0_X_X X_DACS_1N1_0N0 X_DACS_3_2_1_0
Pin Output Control	
%xxxx_xxxx_Exxx_xxxx << 16 %0000_0000_0000_0000 << 16 %0000_0000_1000_0000 << 16	 X_PINS_OFF (default) X_PINS_ON
Write Control	
%xxxx_xxxx_Wxxx_xxxx << 16 %0000_0000_0000_0000 << 16 %0000_0000_1000_0000 << 16	 X_WRITE_OFF (default) X_WRITE_ON
Alternate Order for 1/2/4 bits	
%xxxx_xxxx_xxxx_xxxA << 16 %0000_0000_0000_0000 << 16 %0000_0000_0000_0001 << 16	 X_ALT_OFF (default) X_ALT_ON

Built-In Symbols for Events and Interrupt Sources

Symbol Value	Symbol Name
0	EVENT_INT / INT_OFF
1	EVENT_CT1
2	EVENT_CT2
3	EVENT_CT3
4	EVENT_SE1
5	EVENT_SE2
6	EVENT_SE3
7	EVENT_SE4
8	EVENT_PAT
9	EVENT_FBW
10	EVENT_XMT
11	EVENT_XFI
12	EVENT_XRO
13	EVENT_XRL
14	EVENT_ATN
15	EVENT_QMT

Built-In Symbols for COGINIT Usage

COGINIT Symbol Value	Symbol Name	Details
%00_0000	COGEXEC (default)	Use "COGEXEC + CogNumber" to start a cog in cogexec mode
%10_0000	HUBEXEC	Use "HUBEXEC + CogNumber" to start a cog in hubexec mode
%01_0000	COGEXEC_NEW	Starts an available cog in cogexec mode
%11_0000	HUBEXEC_NEW	Starts an available cog in hubexec mode
%01_0001	COGEXEC_NEW_PAIR	Starts an available eve/odd pair of cogs in cogexec mode, useful for LUT sharing
%11_0001	HUBEXEC_NEW_PAIR	Starts an available eve/odd pair of cogs in hubexec mode, useful for LUT sharing

Built-In Symbol for COGSPIN Usage

COGINIT Symbol Value	Symbol Name	Details
%01_0000	NEWCOG	Starts an available cog

Built-In Numeric Symbols

Symbol Value	Symbol Name	Details
$0000_0000	FALSE	Same as 0
$FFFF_FFFF	TRUE	Same as -1
$8000_0000	NEGX	Negative-extreme integer, -2_147_483_648 ($8000_0000)
$7FFF_FFFF	POSX	Positive-extreme integer, +2_147_483_647 ($7FFF_FFFF)
$4049_0FDB	PI	Single-precision floating-point value of Pi, 3.14159265

Command Line options for PNut.exe

Command	Action	ERROR.TXT file afterwards (file will contain one of these lines)
`pnut`	Start PNut.exe.	`okay`
`pnut filename`	Load *filename* (.spin2 extension is assumed, but not enforced).	`okay`
`pnut filename -c`	Load and compile *filename*, then exit.	`okay` <filename_path>`:`<line_number>`:error:`<error_message>
`pnut filename -r`	Load, compile, and run *filename*, then exit.	okay <filename_path>`:`<line_number>`:error:`<error_message> `serial_error`

`pnut filename -rd`	Load, compile, run, and debug *filename*, then exit when the Debug window gets closed.	`okay`
		<filename_path>`:`<line_number>`:error:`<error_message>
		`serial_error`

`pnut filename -f`	Load, compile, and program flash *filename*, then exit.	`okay`
		<filename_path>`:`<line_number>`:error:`<error_message>
		`serial_error`

`pnut filename -fd`	Load, compile, program flash, and debug *filename*, then exit when the Debug window gets closed.	`okay`
		<filename_path>`:`<line_number>`:error:`<error_message>
		`serial_error`

`pnut -debug {CommPort} {BaudRate}`	Open CommPort (default = 1) at BaudRate (default = 2_000_000) and present incoming DEBUG data and displays.	`okay` `serial_error`

Included Batch File to invoke PNut.exe and return status to STDOUT, STDERR, and ERRORLEVEL

PNUT_SHELL.BAT File	Batch File Line Descriptions
`@echo off`	Cancel echo to console.
`set ERROR_FILE=error.txt`	Set ERROR.TXT filename.
`if exist %ERROR_FILE% del /q /f %ERROR_FILE%`	If ERROR.TXT exists, delete it.
`if exist %1 set GOOD_SRC=1`	Check first parameter for a valid source file.
`if exist %1.spin2 set GOOD_SRC=1`	Check first parameter for a valid .spin2 source file.
`if defined GOOD_SRC (`	IF source file exists
`    pnut_v35L %1 %2 %3`	...Invoke PNut with passed parameters. Example: pnut_shell filename -r
`    set pnuterror = %ERRORLEVEL%`	...Capture ERRORLEVEL from PNut (0 = okay, 1 = error).
`    for /f "tokens=*" %%i in (%ERROR_FILE%) do echo %%i 1>&2`	...Copy ERROR.TXT file to STDOUT and STDERR.
`) else (`	ELSE
`    set pnuterror=-1`	...Set file-not-found error.
`    echo "Error: File NOT found - %1" 1>&2`	...Return file-not-found error message to STDOUT and STDERR.
`)`	
`exit %pnuterror%`	Return ERRORLEVEL. Change to 'exit /b %pnuterror%' to maintain the console window.

Clock Setup

To establish the initial clock setup for your program, you can declare certain symbols which the compiler will look for to determine your setup. These symbols must be defined in one of the following combinations:

CON declarations (numbers are for example, can vary)	Effect	HUBSET %CC_SS **
`CON _clkfreq = 250_000_000` `_errfreq = 0`	Selects XI/XO-crystal-plus-PLL mode, assumes 20MHz crystal. The optimal PLL setting will be computed to achieve _clkfreq. Compilation fails if _clkfreq ± _errfreq is unachievable. *	`10_11`

`CON _xtlfreq = 12_000_000`
`_clkfreq = 148_500_000`
`_errfreq = 150_000`
	Selects XI/XO-crystal-plus-PLL mode, along with frequencies.
	The optimal PLL setting will be computed to achieve _clkfreq.
	Compilation fails if _clkfreq ± _errfreq is unachievable. *
		`1x_11`

`CON _xinfreq = 32_000_000`
`_clkfreq = 297_500_000`
`_errfreq = 100_000`
	Selects XI-input-plus-PLL mode, along with frequencies.
	The optimal PLL setting will be computed to achieve _clkfreq.
	Compilation fails if _clkfreq ± _errfreq is unachievable. *
		`01_10`

`CON _xtlfreq = 16_000_000`	Selects XI/XO-crystal mode and frequency.	`1x_10`
`CON _xinfreq = 100_000_000`	Selects XI-input mode and frequency.	`01_10`
`CON _rcslow`	Selects internal RCSLOW oscillator which runs at ~20KHz.	`00_01`
`CON _rcfast`	Selects internal RCFAST oscillator which runs at 20MHz+. This is the default mode, in case nothing is specified.	`00_00`

* The _errfreq declaration is optional, since _errfreq defaults to 1_000_000.
** If _xtlfreq >= 16_000_000 then x=0 for 15pF per XI/XO, else x=1 for 30pF per XI/XO.

During compilation, two constant symbols are defined by the compiler, whose values reflect the compiled clock setup:

<table>
<tr><th>Symbol</th><th>Description</th></tr>
<tr><td><code>clkmode_</code></td><td>The compiled clock mode, settable via HUBSET.

For Spin2 programs, HUBSET will be invoked with 'clkmode_' before your program starts, in order to set the compiled clock mode. The 'clkmode_' value will also be stored in the hub variable 'clkmode'.
For pure PASM programs, 'clkmode_' can be used to set the clock mode away from its initial RCFAST setting to any crystal/PLL compiled setting, as follows:
<pre>
HUBSET ##clkmode_ & !3   'start crystal/PLL, stay in RCFAST
WAITX   ##20_000_000/100   'wait 10ms
HUBSET ##clkmode_   'switch to crystal/PLL
</pre>
The 'clkmode_' value may differ in each file of the application hierarchy. Files below the top-level file do not inherit the top-level file's value.
</td></tr>
<tr><td><code>clkfreq_</code></td><td>The compiled clock frequency.

For Spin2 programs, the 'clkfreq_' value will be stored in the hub variable 'clkfreq'.
For pure PASM programs, 'clkfreq_' may be referenced only as a constant.
The 'clkfreq_' value may differ in each file of the application hierarchy. Files below the top-level file do not inherit the top-level file's value.
</td></tr>
</table>

For Spin2 programs, two hub variables are maintained which reflect the current clock setup:

<table>
<tr><th>Spin2 Variables</th><th>Description</th></tr>
<tr><td><code>clkmode</code></td><td>The current clock mode, located at LONG[$40]. Initialized with the 'clkmode_' value.</td></tr>
<tr><td><code>clkfreq</code></td><td>The current clock frequency, located at LONG[$44]. Initialized with the 'clkfreq_' value.</td></tr>
<tr><td></td><td>

For Spin2 methods, these variables can be read and written as 'clkmode' and 'clkfreq'.

Rather than write these variables directly, it's much safer to use:

CLKSET(new_clkmode, new_clkfreq)

This way, all other code sees a quick, parallel update to both 'clkmode' and 'clkfreq', and the clock mode transition is done safely, employing the prior values, in order to avoid a potential clock glitch.
For PASM code running under Spin2, these variables can be read and written as follows:
<pre>
RDLONG x,#@clkmode   'read clkmode into x
WRLONG x,#@clkmode   'write x to clkmode

RDLONG x,#@clkfreq   'read clkfreq into x
WRLONG x,#@clkfreq   'write x to clkfreq

SETQ   #2-1   'read clkmode and clkfreq into x and x+1
RDLONG x,#@clkmode

SETQ   #2-1   'write x and x+1 to clkmode and clkfreq
WRLONG x,#@clkmode
</pre>
</td></tr>
</table>

For PASM-only programs, there is a special instruction named ASMCLK which will set the clock mode specified by the clock setup symbols. ASMCLK has no operands, but may be used with a conditional prefix. ASMCLK will assemble to one or six PASM instructions, depending upon the clock mode. This instruction is meant to be used once at the start of a PASM-only program, with the assumption that the RCFAST mode inherited from boot-up is currently selected:

CON declarations (numbers are for example, can vary)	HUBSET %CC_SS	ASMCLK assembles to:

`CON _clkfreq = 250_000_000`
`_errfreq = 0`
	10_11	`HUBSET ##clkmode_ & !%11 'start external clock, stay in RCFAST mode`
		`WAITX ##20_000_000/100 'allow 10ms for external clock to stabilize`
		`HUBSET ##clkmode_ 'switch to external clock mode`

`CON _xtlfreq = 12_000_000`
`_clkfreq = 148_500_000`
`_errfreq = 150_000`
	1x_11	

`CON _xinfreq = 32_000_000`
`_clkfreq = 297_500_000`
`_errfreq = 100_000`
	01_10	

`CON _xtlfreq = 16_000_000`	1x_10	
`CON _xinfreq = 100_000_000`	01_10	
`CON _rcslow`	00_01	`HUBSET #1 'switch to RCSLOW mode`
`CON _rcfast`	00_00	`HUBSET #0 'stay in RCFAST mode`