Propeller Chip Quick Reference

Spin Language

Command	Returns Value	Description
ABORT *Value*	✓	Exit from **PUB**/**PRI** method using abort status with optional return value.
BYTE *Symbol* [*Count*]		Declare byte-sized symbol in **VAR** block.
BYTE *Data*		Declare byte-aligned and/or byte-sized data in **DAT** block.
BYTE [*BaseAddress*] [*Offset*]	✓	Read/write byte of main memory.
Symbol.**BYTE** [*Offset*]	✓	Read/write byte-sized component of word/long-sized variable.
BYTEFILL (*StartAddress*, *Value*, *Count*)		Fill bytes of main memory with a value.
BYTEMOVE (*DestAddress*, *SrcAddress*, *Count*)		Copy bytes from one region to another in main memory.
CASE *CaseExpression* →| *MatchExpression* : →| *Statement(s)* →| *MatchExpression* : →| *Statement(s)* →| **OTHER** : →| *Statement(s)*		Compare expression against matching expression(s), execute code block if match found. *MatchExpression* can contain a single expression or multiple comma-delimited expressions. Expressions can be a single value (ex: 10) or a range of values (ex: 10..15).
CHIPVER	✓	Version number of the Propeller chip.
CLKFREQ	✓	Current System Clock frequency, in Hz.
CLKMODE	✓	Current clock mode setting.
CLKSET (*Mode*, *Frequency*)		Set both clock mode and System Clock frequency at run time.
CNT	✓	Current 32-bit System Counter value.
COGID	✓	Current cog's ID number; 0-7.
COGINIT (*CogID*, *SpinMethod* (*ParameterList*) , *StackPointer*)		Start or restart cog by ID to run Spin code.
COGINIT (*CogID*, *AsmAddress*, *Parameter*)		Start or restart cog by ID to run Propeller Assembly code.
COGNEW (*SpinMethod* (*ParameterList*) , *StackPointer*)	✓	Start new cog for Spin code and get cog ID; 0-7 = succeeded, -1 = failed.
COGNEW (*AsmAddress*, *Parameter*)	✓	Start new cog for Propeller Assembly code and get cog ID; 0-7 = succeeded, -1 = failed.
COGSTOP (*CogID*)		Stop cog by its ID.
CON *Symbol* = *Expr* ((, ┆ ↳)) *Symbol* = *Expr* …		Declare symbolic, global constants.
CON #*Expr* ((, ┆ ↳)) *Symbol* ((, ┆ ↳)) #*Expr* ((, ┆ ↳)) *Symbol* …		Declare global enumerations (incrementing symbolic constants).
CONSTANT (*ConstantExpression*)	✓	Declare in-line constant expression to be completely resolved at compile time.
CTRA	✓	Counter A Control register.
CTRB	✓	Counter B Control register.
DAT *Symbol* *Alignment* *Size* *Data* , *Size* *Data* …		Declare table of data, aligned and sized as specified.
DAT *Symbol* *Condition* *Instruction* *Effect(s)*		Denote Propeller Assembly instruction.
DIRA [*Pin(s)*]	✓	Direction register for 32-bit port A.
FILE "*FileName*"		Import external file as data in **DAT** block.
FLOAT (*IntegerConstant*)	✓	Convert integer constant expression to compile-time floating-point value in any block.
FRQA	✓	Counter A Frequency register.
FRQB	✓	Counter B Frequency register.
((**IF** ┆ **IFNOT**)) *Condition(s)* →| *IfStatement(s)* **ELSEIF** *Condition(s)* →| *ElseIfStatement(s)* … **ELSEIFNOT** *Condition(s)* →| *ElseIfStatement(s)* … **ELSE** →| *ElseStatement(s)*		Test condition(s) and execute block of code if valid. **IF** and **ELSEIF** each test for **TRUE**. **IFNOT** and **ELSEIFNOT** each test for **FALSE**.
INA [*Pin(s)*]	✓	Input register for 32-bit ports A.
LOCKCLR (*ID*)	✓	Clear semaphore to false and get its previous state; **TRUE** or **FALSE**.
LOCKNEW	✓	Check out new semaphore and get its ID; 0-7, or -1 if none were available.
LOCKRET (*ID*)		Return semaphore back to semaphore pool, releasing it for future **LOCKNEW** requests.
LOCKSET (*ID*)	✓	Set semaphore to true and get its previous state; **TRUE** or **FALSE**.
LONG *Symbol* [*Count*]		Declare long-sized symbol in **VAR** block.
LONG *Data*		Declare long-aligned and/or long-sized data in **DAT** block.
LONG [*BaseAddress*] [*Offset*]	✓	Read/write long of main memory.
LONGFILL (*StartAddress*, *Value*, *Count*)		Fill longs of main memory with a value.
LONGMOVE (*DestAddress*, *SrcAddress*, *Count*)		Copy longs from one region to another in main memory.
LOOKDOWN (*Value*:*ExpressionList*)	✓	Get the one-based index of a value in a list.
LOOKDOWNZ (*Value*:*ExpressionList*)	✓	Get the zero-based index of a value in a list.
LOOKUP (*Index*:*ExpressionList*)	✓	Get value from a one-based index position of a list.
LOOKUPZ (*Index*:*ExpressionList*)	✓	Get value from a zero-based index position of a list.
NEXT		Skip remaining statements of **REPEAT** loop and continue with the next loop iteration.

Spin Language (continued…)

Command	Returns Value	Description
OBJ *Symbol* [*Count*] : "*Object*" ↳ *Symbol* [*Count*] : "*Object*" …		Declare symbol object references.
OUTA [*Pin(s)*]	✓	Output register for 32-bit port A.
PAR	✓	Cog Boot Parameter register.
PHSA	✓	Counter A Phase Lock Loop (PLL) register.
PHSB	✓	Counter B Phase Lock Loop (PLL) register.
PRI *Name* (*Par* ,*Par* …) : *RVal* | *LVar* [*Cnt*] ,*LVar* [*Cnt*] … *SourceCodeStatements*		Declare private method with optional parameters, return value and local variables.
PUB *Name* (*Par* ,*Par* …) : *RVal* | *LVar* [*Cnt*] ,*LVar* [*Cnt*] … *SourceCodeStatements*		Declare public method with optional parameters, return value and local variables.
QUIT		Exit from **REPEAT** loop immediately.
REBOOT		Reset the Propeller chip.
REPEAT *Count* → *Statement(s)*		Execute code block repetitively, either infinitely, or for a finite number of iterations.
REPEAT *Variable* **FROM** *Start* **TO** *Finish* **STEP** *Delta* → *Statement(s)*		Execute code block repetitively, for finite, counted iterations.
REPEAT ((**UNTIL** | **WHILE**)) *Condition(s)* → *Statement(s)*		Execute code block repetitively, zero-to-many conditional iterations.
REPEAT → *Statement(s)* ((**UNTIL** | **WHILE**)) *Condition(s)*		Execute code block repetitively, one-to-many conditional iterations.
RESULT	✓	Return value variable for **PUB**/**PRI** methods.
RETURN *Value*	✓	Exit from **PUB**/**PRI** method with optional return *Value*.
ROUND (*FloatConstant*)	✓	Round floating-point constant to the nearest integer at compile-time, in any block.
SPR [*Index*]	✓	Special Purpose Register array.
STRCOMP (*StringAddress1*, *StringAddress2*)	✓	Compare two strings for equality.
STRING (*StringExpression*)	✓	Declare in-line string constant and get its address.
STRSIZE (*StringAddress*)	✓	Get size, in bytes, of zero-terminate string.
TRUNC (*FloatConstant*)	✓	Remove fractional portion from floating-point constant at compile-time, in any block.
VAR *Size Symbol* [*Count*] ((, | ↳ *Size*)) *Symbol* [*Count*] …		Declare symbolic global variables.
VCFG	✓	Video Configuration register.
VSCL	✓	Video Scale register.
WAITCNT (*Value*)		Pause cog's execution temporarily.
WAITPEQ (*State*, *Mask*, *Port*)		Pause cog's execution until I/O pin(s) match designated state(s).
WAITPNE (*State*, *Mask*, *Port*)		Pause cog's execution until I/O pin(s) do not match designated state(s).
WAITVID (*Colors*, *Pixels*)		Pause cog's execution until its Video Generator is available for pixel data.
WORD *Symbol* [*Count*]		Declare word-sized symbol in **VAR** block.
WORD *Data*		Declare word-aligned and/or word-sized data in **DAT** block.
WORD [*BaseAddress*] [*Offset*]	✓	Read/write word of main memory.
Symbol.**WORD** [*Offset*]	✓	Read/write word-sized component of long-sized variable.
WORDFILL (*StartAddress*, *Value*, *Count*)		Fill words of main memory with a value.
WORDMOVE (*DestAddress*, *SrcAddress*, *Count*)		Copy words from one region to another in main memory.

Propeller Assembly Language

Instruction			Description	Z Result	C Result	Result	Clocks
ABS	*AValue*,	# *SValue*	Get absolute value of a number.	Result = 0	S[31]	Written	4
ABSNEG	*NValue*,	# *SValue*	Get the negative of a number's absolute value.	Result = 0	S[31]	Written	4
ADD	*Value1*,	# *Value2*	Add unsigned values.	Result = 0	Unsigned Carry	Written	4
ADDABS	*Value*,	# *SValue*	Add absolute value to another value.	Result = 0	Unsigned Carry	Written	4
ADDS	*SValue1*,	# *SValue2*	Add signed values.	Result = 0	Signed Overflow	Written	4
ADDSX	*SValue1*,	# *SValue2*	Add signed values plus C.	Z & (Result = 0)	Signed Overflow	Written	4
ADDX	*Value1*,	# *Value2*	Add unsigned values plus C.	Z & (Result = 0)	Unsigned Carry	Written	4
AND	*Value1*,	# *Value2*	Bitwise AND values.	Result = 0	Parity of Result	Written	4
ANDN	*Value1*,	# *Value2*	Bitwise AND value with NOT of another.	Result = 0	Parity of Result	Written	4
CALL	#*Address*		Jump to address with intention to return to next instruction.	Result = 0	---	Written	4
CLKSET	*Mode*		Set clock mode at run time.	---	---	Not Written	7..22 *
CMP	*Value1*,	# *Value2*	Compare unsigned values.	D = S	Unsigned Borrow	Not Written	4
CMPS	*SValue1*,	# *SValue2*	Compare signed values.	D = S	Signed Borrow	Not Written	4
CMPSUB	*Value1*,	# *Value2*	Compare unsigned values, subtract second if it is lesser or equal.	D = S	Unsigned (D => S)	Written	4
CMPSX	*SValue1*,	# *SValue2*	Compare signed values plus C.	Z & (D = S+C)	Signed Borrow	Not Written	4
CMPX	*Value1*,	# *Value2*	Compare unsigned values plus C.	Z & (D = S+C)	Unsigned Borrow	Not Written	4
COGID	*Destination*		Get current cog's ID.	Result = 0	---	Written	7..22 *
COGINIT	*Destination*		Re/start cog, ID optional, to run Propeller Assembly or Spin code.	Result = 0	No Cog Free	Not Written	7..22 *
COGSTOP	*CogID*		Start a cog by ID.	---	---	Not Written	7..22 *
DJNZ	*Value*,	# *Address*	Decrement value and jump to address if not zero.	Result = 0	Unsigned Borrow	Written	4 or 8 **

Propeller Assembly Language (continued…)

Instruction		Description	Z Result	C Result	Result	Clocks
HUBOP	*Destination*, # *Operation*	Perform a hub operation.	Result = 0	---	Not Written	7..22 *
JMP	# *Address*	Jump to address unconditionally.	Result = 0	---	Not Written	4
JMPRET	*RetInstAddr*, # *DestAddr*	Jump to address with intention to "return" to another address.	Result = 0	---	Written	4
LOCKCLR	*ID*	Clear semaphore to False and get its previous state.	---	Prior Lock State	Not Written	7..22 *
LOCKNEW	*NewID*	Check out new semaphore and get its ID.	Result = 0	No Lock Free	Written	7..22 *
LOCKRET	*ID*	Return semaphore back for future "new semaphore" requests.	---	---	Not Written	7..22 *
LOCKSET	*ID*	Set semaphore to true and get its previous state.	---	Prior Lock State	Not Written	7..22 *
MAX	*Value1*, # *Value2*	Limit maximum of unsigned value to another unsigned value.	D = S	Unsigned (D < S)	Written	4
MAXS	*SValue1*, # *SValue2*	Limit maximum of signed value to another signed value.	D = S	Signed (D < S)	Written	4
MIN	*Value1*, # *Value2*	Limit minimum of unsigned value to another unsigned value.	D = S	Unsigned (D < S)	Written	4
MINS	*SValue1*, # *SValue2*	Limit minimum of signed value to another signed value.	D = S	Signed (D < S)	Written	4
MOV	*Destination*, # *Value*	Set register to a value.	Result = 0	S[31]	Written	4
MOVD	*Destination*, # *Value*	Set register's destination field to a value.	Result = 0	---	Written	4
MOVI	*Destination*, # *Value*	Set register's instruction field to a value.	Result = 0	---	Written	4
MOVS	*Destination*, # *Value*	Set register's source field to a value.	Result = 0	---	Written	4
MUXC	*Destination*, # *Mask*	Set discrete bits of value to state of C.	Result = 0	Parity of Result	Written	4
MUXNC	*Destination*, # *Mask*	Set discrete bits of value to state of !C.	Result = 0	Parity of Result	Written	4
MUXNZ	*Destination*, # *Mask*	Set discrete bits of value to state of !Z.	Result = 0	Parity of Result	Written	4
MUXZ	*Destination*, # *Mask*	Set discrete bits of value to state of Z.	Result = 0	Parity of Result	Written	4
NEG	*NValue*, # *SValue*	Get negative of a number.	Result = 0	S[31]	Written	4
NEGC	*RValue*, # *Value*	Get value, or its additive inverse, based on C.	Result = 0	S[31]	Written	4
NEGNC	*RValue*, # *Value*	Get value, or its additive inverse, based on !C.	Result = 0	S[31]	Written	4
NEGNZ	*RValue*, # *Value*	Get value, or its additive inverse, based on !Z.	Result = 0	S[31]	Written	4
NEGZ	*RValue*, # *Value*	Get value, or its additive inverse, based on Z.	Result = 0	S[31]	Written	4
NOP		No operation, just elapse four clock cycles.	---	---	---	4
OR	*Value1*, # *Value2*	Bitwise OR values.	Result = 0	Parity of Result	Written	4
RCL	*Value*, # *Bits*	Rotate C left into value by specified number of bits.	Result = 0	D[31]	Written	4
RCR	Value, # Bits	Rotate C right into value by specified number of bits.	Result = 0	D[0]	Written	4
RDBYTE	*Value*, # *Address*	Read byte of main memory.	Result = 0	---	Written	7..22 *
RDLONG	*Value*, # *Address*	Read long of main memory.	Result = 0	---	Written	7..22 *
RDWORD	*Value*, # *Address*	Read word of main memory.	Result = 0	---	Written	7..22 *
RET		Return to address.	Result = 0	---	Not Written	4
REV	*Value*, # *Bits*	Reverse LSBs of value and zero-extend.	Result = 0	D[0]	Written	4
ROL	*Value*, # *Bits*	Rotate value left by specified number of bits.	Result = 0	D[31]	Written	4
ROR	*Value*, # *Bits*	Rotate value right by specified number of bits.	Result = 0	D[0]	Written	4
SAR	*Value*, # *Bits*	Shift value arithmetically right by specified number of bits.	Result = 0	D[0]	Written	4
SHL	*Value*, # *Bits*	Shift value left by specified number of bits.	Result = 0	D[31]	Written	4
SHR	*Value*, # *Bits*	Shift value right by specified number of bits.	Result = 0	D[0]	Written	4
SUB	*Value1*, # *Value2*	Subtract unsigned values.	Result = 0	Unsigned Borrow	Written	4
SUBABS	*Value*, # *SValue*	Subtract absolute value from another value.	Result = 0	Unsigned Borrow	Written	4
SUBS	*SValue1*, # *SValue2*	Subtract signed values.	Result = 0	Signed Underflow	Written	4
SUBSX	*SValue1*, # *SValue2*	Subtract signed value plus C from another signed value.	Z & (Result = 0)	Signed Underflow	Written	4
SUBX	*Value1*, # *Value2*	Subtract unsigned value plus C from another unsigned value.	Z & (Result = 0)	Unsigned Borrow	Written	4
SUMC	*SValue1*, # *SValue2*	Sum signed value with another whose sign is inverted based on C.	Result = 0	Signed Overflow	Written	4
SUMNC	*SValue1*, # *SValue2*	Sum signed value with another whose sign is inverted based on !C.	Result = 0	Signed Overflow	Written	4
SUMNZ	*SValue1*, # *SValue2*	Sum signed value with another whose sign is inverted based on !Z.	Result = 0	Signed Overflow	Written	4
SUMZ	*SValue1*, # *SValue2*	Sum signed value with another whose sign is inverted based on Z.	Result = 0	Signed Overflow	Written	4
TEST	*Value1*, # *Value2*	Bitwise AND values to affect flags only.	Result = 0	Parity of Result	Not Written	4
TESTN	*Value1*, # *Value2*	Bitwise AND value with NOT of another to affect flags only.	Result = 0	Parity of Result	Not Written	4
TJNZ	*Value*, # *Address*	Test value and jump to address if not zero.	Result = 0	0	Not Written	4 or 8 **
TJZ	*Value*, # *Address*	Test value and jump to address if zero.	Result = 0	0	Not Written	4 or 8 **
WAITCNT	*Target*, # *Delta*	Pause execution temporarily.	Result = 0	Unsigned Carry	Written	5+
WAITPEQ	*State*, # *Mask*	Pause execution until I/O pin(s) match designated state(s).	Result = 0	---	Not Written	5+
WAITPNE	*State*, # *Mask*	Pause execution until I/O pin(s) don't match designated state(s).	Result = 0	---	Not Written	5+
WAITVID	*Colors*, # *Pixels*	Pause execution until Video Generator can take pixel data.	Result = 0	---	Not Written	5+
WRBYTE	*Value*, # *Address*	Write byte to main memory.	---	---	Not Written	7..22 *
WRLONG	*Value*, # *Address*	Write long to main memory.	---	---	Not Written	7..22 *
WRWORD	*Value*, # *Address*	Write word to main memory.	---	---	Not Written	7..22 *
XOR	*Value1*, # *Value2*	Bitwise XOR values.	Result = 0	Parity of Result	Written	4

* Hub instructions require 7 to 22 clock cycles to execute depending on the relation between its moment of execution and the cog's hub access window. Since cogs run independent of the hub, they must sync to the hub to execute hub instructions. Cogs receive an "access window" every 16 clocks. The first hub instruction in a sequence will take 0 to 15 clocks to sync and 7 clocks afterwards to execute; 0+7 to 15+7 = 7 to 22 clock cycles. After that instruction, there are 9 (16–7) free clocks before the cog's next access window; enough time for two 4-clock instructions. Beware that hub instructions can cause timing to appear indeterminate; particularly the first hub instruction in a sequence.

** Conditional-Jump instructions require extra clock cycles if a jump is not required. These instructions take 4 clock cycles if a jump is required and 8 clock cycles if no jump is required. Since loops utilizing these instructions typically need to be fast, they are optimized in this way for speed.

Math and Logic Operators

Level[1]	Operator: Normal	Operator: Assign[2]	Constant Expressions[3]: Integer	Constant Expressions[3]: Float	Is Unary	Description
Highest (0)	--	always			✓	Pre-decrement (--X) or post-decrement (X--).
	++	always			✓	Pre-increment (++X) or post-increment (X++).
	~	always			✓	Sign-extend bit 7 (~X) or post-clear to 0 (X~).
	~~	always			✓	Sign-extend bit 15 (~~X) or post-set to -1 (X~~).
	?	always			✓	Random number forward (?X) or reverse (X?).
	@	never	✓		✓	Symbol address.
	@@	never			✓	Object address plus symbol.
1	+	never	✓	✓	✓	Positive (+X); unary form of Add.
	-	if solo	✓	✓	✓	Negate (-X); unary form of Subtract.
	^^	if solo	✓	✓	✓	Square root.
	||	if solo	✓	✓	✓	Absolute value.
	|<	if solo	✓		✓	Bitwise: Decode 0 – 31 to long w/single-high-bit.
	>|	if solo	✓		✓	Bitwise: Encode long to 0 – 32; high-bit priority.
	!	if solo	✓		✓	Bitwise: NOT.
2	<-	<-=	✓			Bitwise: Rotate left.
	->	->=	✓			Bitwise: Rotate right.
	<<	<<=	✓			Bitwise: Shift left.
	>>	>>=	✓			Bitwise: Shift right.
	~>	~>=	✓			Shift arithmetic right.
	><	><=	✓			Bitwise: Reverse.
3	&	&=	✓			Bitwise: AND.
4	|	|=	✓			Bitwise: OR.
	^	^=	✓			Bitwise: XOR.
5	*	*=	✓	✓		Multiply and return lower 32 bits (signed).
	**	**=	✓			Multiply and return upper 32 bits (signed).
	/	/=	✓	✓		Divide (signed).
	//	//=	✓			Modulus (signed).
6	+	+=	✓	✓		Add.
	-	-=	✓	✓		Subtract.
7	#>	#>=	✓	✓		Limit minimum (signed).
	<#	<#=	✓	✓		Limit maximum (signed).
8	<	<=	✓	✓		Boolean: Is less than (signed).
	>	>=	✓	✓		Boolean: Is greater than (signed).
	<>	<>=	✓	✓		Boolean: Is not equal.
	==	===	✓	✓		Boolean: Is equal.
	=<	=<=	✓	✓		Boolean: Is equal or less (signed).
	=>	=>=	✓	✓		Boolean: Is equal or greater (signed).
9	NOT	if solo	✓	✓	✓	Boolean: NOT (promotes non-0 to -1).
10	AND	AND=	✓	✓		Boolean: AND (promotes non-0 to -1).
11	OR	OR=	✓	✓		Boolean: OR (promotes non-0 to -1).
Lowest (12)	=	always	n/a[3]	n/a[3]		Constant assignment (CON blocks).
	:=	always	n/a[3]	n/a[3]		Variable assignment (PUB/PRI blocks).

[1] Precedence level: higher-level operators evaluate before lower-level operators. Operators in same level are commutable; evaluation order does not matter.

[2] Assignment forms of binary (non-unary) operators are in the lowest precedence (level 12).

[3] Assignment forms of operators are not allowed in constant expressions.

Assembly Conditions

Condition	Instruction Executes	Condition	Instruction Executes
IF_ALWAYS	always	IF_NC_AND_Z	if C clear and Z set
IF_NEVER	never	IF_NC_AND_NZ	if C clear and Z clear
IF_E	if equal (Z)	IF_C_OR_Z	if C set or Z set
IF_NE	if not equal (!Z)	IF_C_OR_NZ	if C set or Z clear
IF_A	if above (!C & !Z)	IF_NC_OR_Z	if C clear or Z set
IF_B	if below (C)	IF_NC_OR_NZ	if C clear or Z clear
IF_AE	if above/equal (!C)	IF_Z_EQ_C	if Z equal to C
IF_BE	if below/equal (C | Z)	IF_Z_NE_C	if Z not equal to C
IF_C	if C set	IF_Z_AND_C	if Z set and C set
IF_NC	if C clear	IF_Z_AND_NC	if Z set and C clear
IF_Z	if Z set	IF_NZ_AND_C	if Z clear and C set
IF_NZ	if Z clear	IF_NZ_AND_NC	if Z clear and C clear
IF_C_EQ_Z	if C equal to Z	IF_Z_OR_C	if Z set or C set
IF_C_NE_Z	if C not equal to Z	IF_Z_OR_NC	if Z set or C clear
IF_C_AND_Z	if C set and Z set	IF_NZ_OR_C	if Z clear or C set
IF_C_AND_NZ	if C set and Z clear	IF_NZ_OR_NC	if Z clear or C clear

Constants (pre-defined)

Constant[1]	Description	
_CLKFREQ	Settable in Top Object File to specify System Clock frequency.	
_CLKMODE	Settable in Top Object File to specify application's clock mode.	
_XINFREQ	Settable in Top Object File to specify external crystal frequency.	
_FREE	Settable in Top Object File to specify application's free space.	
_STACK	Settable in Top Object File to specify application's stack space.	
TRUE	Logical true:	-1 ($FFFFFFFF)
FALSE	Logical false:	0 ($00000000)
POSX	Max. positive integer:	2,147,483,647 ($7FFFFFFF)
NEGX	Max. negative integer:	-2,147,483,648 ($80000000)
PI	Floating-point PI:	≈ 3.141593 ($40490FDB)
RCFAST	Internal fast oscillator:	$00000001 (%00000000001)
RCSLOW	Internal slow oscillator:	$00000002 (%00000000010)
XINPUT	External clock/oscillator:	$00000004 (%00000000100)
XTAL1	External low-speed crystal:	$00000008 (%00000001000)
XTAL2	External medium-speed crystal:	$00000010 (%00000010000)
XTAL3	External high-speed crystal:	$00000020 (%00000100000)
PLL1X	External frequency times 1:	$00000040 (%00001000000)
PLL2X	External frequency times 2:	$00000080 (%00010000000)
PLL4X	External frequency times 4:	$00000100 (%00100000000)
PLL8X	External frequency times 8:	$00000200 (%01000000000)
PLL16X	External frequency times 16:	$00000400 (%10000000000)

[1] "Settable" constants are defined in Top Object File's CON block. Most expect whole numbers, however _CLKMODE uses Valid Clock Modes, below.

Valid Clock Modes

Valid Expression	CLK Reg. Value	Valid Expression	CLK Reg. Value
RCFAST	0_0_0_00_000	XTAL1 + PLL1X	0_1_1_01_011
		XTAL1 + PLL2X	0_1_1_01_100
RCSLOW	0_0_0_00_001	XTAL1 + PLL4X	0_1_1_01_101
		XTAL1 + PLL8X	0_1_1_01_110
XINPUT	0_0_1_00_010	XTAL1 + PLL16X	0_1_1_01_111
		XTAL2 + PLL1X	0_1_1_10_011
XTAL1	0_0_1_01_010	XTAL2 + PLL2X	0_1_1_10_100
XTAL2	0_0_1_10_010	XTAL2 + PLL4X	0_1_1_10_101
XTAL3	0_0_1_11_010	XTAL2 + PLL8X	0_1_1_10_110
		XTAL2 + PLL16X	0_1_1_10_111
XINPUT + PLL1X	0_1_1_00_011	XTAL3 + PLL1X	0_1_1_11_011
XINPUT + PLL2X	0_1_1_00_100	XTAL3 + PLL2X	0_1_1_11_100
XINPUT + PLL4X	0_1_1_00_101	XTAL3 + PLL4X	0_1_1_11_101
XINPUT + PLL8X	0_1_1_00_110	XTAL3 + PLL8X	0_1_1_11_110
XINPUT + PLL16X	0_1_1_00_111	XTAL3 + PLL16X	0_1_1_11_111

Assembly Directives

Directive	Description
FIT *Address*	Validate previous instr/data fit below an address.
ORG *Address*	Adjust compile-time cog address pointer.
Symbol RES *Count*	Reserve next long(s) for symbol.

Assembly Effects

Effect	Results In	Effect	Results In
WC	C Flag modified	WR	Destination Register modified
WZ	Z Flag modified	NR	Destination Register not modified