

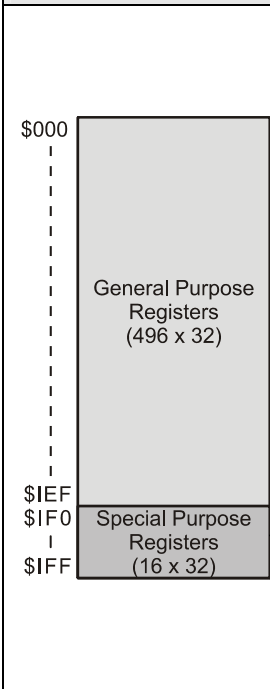
Supplement and Errata for *Propeller Manual v1.0* (#122-32000)

(Items added/changed/deleted are marked in **blue**.)

Supplemental Information

Page 23:

Table 1-3:

Table 0-1: Cog RAM Special Purpose Registers				
Cog RAM Map	Address	Name	Type	Description
 <p>General Purpose Registers (496 x 32)</p> <p>Special Purpose Registers (16 x 32)</p>	\$1F0	PAR	Read-Only ¹	Boot Parameter
	\$1F1	CNT	Read-Only ¹	System Counter
	\$1F2	INA	Read-Only ¹	Input States for P31 - P0
	\$1F3	INB	Read-Only ¹	Input States for P63- P32³
	\$1F4	OUTA	Read/Write	Output States for P31 - P0
	\$1F5	OUTB	Read/Write	Output States for P63 - P32³
	\$1F6	DIRA	Read/Write	Direction States for P31 - P0
	\$1F7	DIRB	Read/Write	Direction States for P63 - P32³
	\$1F8	CTRA	Read/Write	Counter A Control
	\$1F9	CTRB	Read/Write	Counter B Control
	\$1FA	FRQA	Read/Write	Counter A Frequency
	\$1FB	FRQB	Read/Write	Counter B Frequency
	\$1FC	PHSA	Read/Write²	Counter A Phase
	\$1FD	PHSB	Read/Write²	Counter B Phase
	\$1FE	VCFG	Read/Write	Video Configuration
	\$1FF	VSCL	Read/Write	Video Scale

Note 1: Only accessible as a source register (i.e. MOV *Dest, Source*).

Note 2: Only readable as a source register (i.e. MOV *Dest, Source*); read-modify-write not possible as a destination register.

Note 3: Reserved for future use.

Page 28:

Modified last sentence of second paragraph in **CLK Register** section:

When possible, it is recommended to use **Spin's** CLKSET command (page 183), since it automatically updates all the above-mentioned locations with the proper information.

Page 30:

Table 1-10 RCSLOW row:

0	0	1	RCSLOW	~20 kHz	Internal	Very low power. No external parts. May range from 13 kHz to 33 kHz.
---	---	---	--------	---------	----------	---

Page 136:

OBJ block of Blinker2.spin code:

```
OBJ  
LED[MAXLEDS] : "Output"
```

Page 165:

BYTE syntax should be the following:

```
VAR  
  BYTE Symbol <[Count]>  
-----  
DAT  
  <Symbol> BYTE Data <[Count]>  
-----  
(PUB | PRI)  
  BYTE [BaseAddress] <[Offset]>  
-----  
(PUB | PRI)  
  Symbol. BYTE <[Offset]>
```

- *Symbol* is the desired name for the variable (Syntax 1) or data block (Syntax 2) or is the existing name of the variable (Syntax 4).
- *Count* is an optional expression indicating the number of byte-sized elements for *Symbol* (Syntax 1), or the number of byte-sized entries of *Data* (Syntax 2) to store in a data table.
- *Data* is a constant expression or comma-separated list of constant expressions. Quoted strings of characters are also allowed; they are treated as a comma-separated list of characters.
- *BaseAddress* is an expression describing the address of main memory to read or write. If *Offset* is omitted, *BaseAddress* is the actual address to operate on. If *Offset* is specified, *BaseAddress* + *Offset* is the actual address to operate on.
- *Offset* is an optional expression indicating the offset from *BaseAddress* to operate on, or the offset from byte 0 of *Symbol*.

New paragraphs at end of **Byte Data Declaration (Syntax 2)** section, page 167:

Data items may be repeated by using the optional *Count* field. For example:

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

The above example declares a byte-aligned, byte-sized data table, called `MyData`, consisting of the following ten values: 64, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, 55. There were eight occurrences of \$AA due to the [8] in the declaration immediately after it.

Page 175:

Modified last paragraph:

The value that `CLKFREQ` returns is actually read from long 0 (the first location in RAM) and that value can change whenever the application changes the clock mode, either manually or via the `CLKSET` command. Objects that are time-sensitive should check `CLKFREQ` at strategic points in order to adjust to new settings automatically.

Page 194:

CON (Constant Block) syntax should be the following:

```
CON
  Symbol = Expression <((, | ↵)) Symbol = Expression>...
CON
  #Expression ((, | ↵) Symbol <[Offset] <((, | ↵)) Symbol <[Offset] >...
CON
  Symbol <[Offset] <((, | ↵) Symbol <[Offset] >...
```

- *Symbol* is the desired name for the constant.
- *Expression* is any valid integer, or floating-point, constant algebraic expression. Expression can include other constant symbols as long as they were defined previously.
- *Offset* is an optional expression by which to adjust the enumeration value for the *Symbol* following this one. If *Offset* is not specified, the default offset of 1 is applied. Use *Offset* to influence the next *Symbol*'s enumerated value to something other than this *Symbol*'s value plus 1.

Page 199:

New paragraphs below paragraph 1:

A more recommended way to achieve the previous example's result is to include the optional *Offset* field. The previous code could have been written as follows:

```
CON
'Declare modes of operation
#1, RunTest, RunVerbose[3], RunBrief, RunFull
```

Just as before, `RunTest` and `RunVerbose` are 1 and 2, respectively. The `[3]` immediately following `RunVerbose` causes the current enumeration value (2) to be incremented by 3 before the next enumerated symbol. The effect of this is also like before, `RunBrief` and `RunFull` are 5 and 6, respectively. The advantage of this technique, however, is that the enumerated symbols are all set relative to each other. Changing the line's starting value causes them all to change relatively. For example, changing the `#1`, to `#4` causes `RunTest` and `RunVerbose` to be 4 and 5, respectively, and `RunBrief` and `RunFull` to be 8 and 9, respectively. In contrast, if the original example's `#1` were changed to `#4`, both `RunVerbose` and `RunBrief` would be set to 5, possibly causing the code that relies on those symbols to misbehave.

The *Offset* value may be any signed value, but only affects the value immediately following it; the enumerated value is always incremented by 1 after *Symbol*'s that don't specify *Offset*. If overlapping values are desired, specifying an *Offset* of 0 or less can achieve that effect.

Modified sentence within paragraph 3:

Anything defined this way will always start with the first symbol equal to either 0 (for new `CON` blocks) or to the next enumerated value relative to the previous one (within the same `CON` block).

Page 203:

New sentences to add at end of **RCFAST through PLL16X** paragraph:

Note that they are enumerated constants and are not equivalent to the corresponding `CLK` register value. See `CLK Register` on page 28 for information regarding how each constant relates to the `CLK` register bits.

Page 208:

DAT syntax should be the following:

DAT

*<Symbol> Alignment <Size> <Data> [*Count*] <, <Size> Data>...*

DAT

<Symbol> <Condition> Instruction <Effect(s)>

- *Symbol* is an optional name for the data, reserved space, or instruction that follows.
- Alignment is the desired alignment and default size (BYTE, WORD, or LONG) of the data elements that follow.
- *Size* is the desired size (BYTE, WORD, or LONG) of the following data element immediately following it; alignment is unchanged.
- *Data* is a constant expression or comma-separated list of constant expressions. Quoted strings of characters are also allowed; they are treated as a comma-separated list of characters.
- *Count* is an optional expression indicating the number of byte-, word-, or long-sized entries of *Data* to store in the data table.
- *Condition* is an assembly language condition, IF_C, IF_NC, IF_Z, etc.
- *Instruction* is an assembly language instruction, ADD, SUB, MOV, etc., and all its operands.
- *Effect(s)* is/are one, two or three assembly language effects that cause the result of the instruction to be written or not, NR, WR, WC, or WZ.

Page 211:

New paragraph above the **Writing Propeller Assembly Code (Syntax 2)** section:

Declaring Repeating Data (Syntax 1)

Data items may be repeated by using the optional *Count* field. For example:

DAT

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

The above example declares a byte-aligned, byte-sized data table, called *MyData*, consisting of the following ten values: 64, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, \$AA, 55. There were eight occurrences of \$AA due to the [8] in the declaration immediately after it.

Page 212:

Clarification in third paragraph of the **Explanation** section:

DIRA is used to both set and get the current direction states of one or more I/O pins in Port A. A low (0) bit sets the corresponding I/O pin to an input direction. A high (1) bit sets the corresponding I/O pin to an output direction. [All the DIRA](#)

register's bits default to zero (0) upon cog startup; all I/O pins are specified as inputs by that cog until the code instructs otherwise.

Pages 236 - 237:

LONG syntax should be the following:

```
VAR
  LONG Symbol <[Count]>
DAT
  <Symbol> LONG Data <[Count]>
((PUB | PRI))
  LONG [BaseAddress] <[Offset]>
```

- *Symbol* is the desired name for the variable (Syntax 1) or data block (Syntax 2).
- *Count* is an optional expression indicating the number of long-sized elements for *Symbol* (Syntax 1), or the number of long-sized entries of *Data* (Syntax 2) to store in a data table.
- *Data* is a constant expression or comma-separated list of constant expressions.
- *BaseAddress* is an expression describing the address of main memory to read or write. If *Offset* is omitted, *BaseAddress* is the actual address to operate on. If *Offset* is specified, *BaseAddress* + *Offset* is the actual address to operate on.
- *Offset* is an optional expression indicating the offset from *BaseAddress* to operate on.

New paragraphs to add at end of **Long Data Declaration (Syntax 2)** section, page 237:

Data items may be repeated by using the optional *Count* field. For example:

```
DAT
  MyData      long  640_000, $BB50[3]
```

The above example declares a long-aligned, long-sized data table, called *MyData*, consisting of the following four values: 640000, \$BB50, \$BB50, \$BB50. There were three occurrences of \$BB50 due to the [3] in the declaration immediately after it.

Page 280:

Clarification in third paragraph of the **Explanation** section:

OUTA is used to both set and get the current output states of one or more I/O pins in Port A. A low (0) bit sets the corresponding I/O pin to ground. A high (1) bit sets the corresponding I/O pin VDD (3.3 volts). All the OUTA register's bits default to zero (0) upon cog startup.

Page 316:

Additional section after the **Scope of Variables** section:

Organization of Variables

During compilation of an object, all declarations in its collective Variable Blocks are group together by type. The variables in RAM are arranged with all the longs first, followed by all words, and finally by all bytes. This is done so that RAM space is allocated efficiently without unnecessary gaps. Keep this in mind when writing code that accesses variables indirectly based on relative positions to each other.

Page 331:

WORD syntax should be the following:

```
VAR
  WORD Symbol <[Count]>
DAT
  <[Symbol]> WORD Data <[Count]>
((PUB | PRI))
  WORD [BaseAddress] <[Offset]>
((PUB | PRI))
  Symbol. WORD <[Offset]>
```

- *Symbol* is the desired name for the variable (Syntax 1) or data block (Syntax 2) or is the existing name of the variable (Syntax 4).
- *Count* is an optional expression indicating the number of word-sized elements for *Symbol* (Syntax 1), or the number of word-sized entries of *Data* (Syntax 2) to store in a data table.
- *Data* is a constant expression or comma-separated list of constant expressions.
- *BaseAddress* is an expression describing the address of main memory to read or write. If *Offset* is omitted, *BaseAddress* is the actual address to operate on. If *Offset* is specified, *BaseAddress* + *Offset* is the actual address to operate on.
- *Offset* is an optional expression indicating the offset from *BaseAddress* to operate on, or the offset from byte 0 of *Symbol*.

Page 333:

New paragraphs at end of **Word Data Declaration (Syntax 2)** section:

Data items may be repeated by using the optional *Count* field. For example:

```
DAT
  MyData      word  640, $AAAA[4], 5_500
```

The above example declares a word-aligned, word-sized data table, called *MyData*, consisting of the following six values: 640, \$AAAA, \$AAAA, \$AAAA, \$AAAA, 5500.

There were four occurrences of \$AAAA due to the [4] in the declaration immediately after it.

Page 345:

Instruction added between TEST and MOV instructions:

TESTN Bitwise AND a value with the NOT of another to affect flags only; p 410.

Page 350 - 351:

ADDABS row:

ADDABS	D, S	100010 001i 1111 dddddddd ssssssss	Result = 0	Unsigned Carry ¹	Written	4
--------	------	------------------------------------	------------	-----------------------------	---------	---

SUBABS row:

SUBABS	D, S	100011 001i 1111 dddddddd ssssssss	Result = 0	Unsigned Borrow ²	Written	4
--------	------	------------------------------------	------------	------------------------------	---------	---

Instruction added between TEST and TJNZ rows:

TESTN	D, S	011001 000i 1111 dddddddd ssssssss	Result = 0	Parity of Result	Not Written	4
-------	------	------------------------------------	------------	------------------	-------------	---

Footnotes added to table:

Note 1: ADDABS C out: If S is negative, C = the inverse of unsigned borrow (for D-S).

Note 2: SUBABS C out: If S is negative, C = the inverse of unsigned carry (for D+S).

Page 353:

ABS opcode table:

-INSTR-	ZCRI	-CON-	-DEST-	-SRC-	Z Result	C Result	Result	Clocks
101010	001i	1111	ddddddd	sssssss	Result = 0	S[31]	Written	4

Page 355:

ADDABS opcode table:

-INSTR-	ZCRI	-CON-	-DEST-	-SRC-	Z Result	C Result	Result	Clocks
100010	001i	1111	ddddddd	sssssss	Result = 0	Unsigned Carry ¹	Written	4

1: If SValue is negative, C Result is the inverse of unsigned borrow (for Value - SValue).

Page 398:

New paragraph at bottom of page:

Notice that we defined our reserved space after our explicitly-defined data, rather than before it. It is important to only use the **RES** directive after all explicitly defined data intended for the cog at run-time; that is, after code, bytes, words, and longs. This is because the **RES** directive simply increments the compiler's cog memory pointer, not the object memory pointer. If we had mistakenly defined `Time` (a user-reserved symbol) before `Delay` (an explicitly-defined long symbol), at run-time `Time` and `Delay` would both end up using the same cog register, instead of two different registers as was intended.

Page 404:

SUBABS opcode table:

-INSTR-	ZCRI	-CON-	-DEST-	-SRC-	Z Result	C Result	Result	Clocks
100010	001i	1111	ddddddddd	sssssssss	Result = 0	Unsigned Borrow ¹	Written	4

¹: If *SValue* is negative, C Result is the inverse of unsigned carry (for Value + SValue).

Pages 381 - 383:

The **MOVD** instruction; new paragraph inserted after **Explanation's** first paragraph:

For self-modifying code, however, ensure that the **MOVD** instruction executes at least 2 instruction-cycles prior to the instruction that it modifies. This gives the cog time to write the result before it fetches that instruction for execution; otherwise, the unmodified instruction will be fetched and executed.

The **MOVI** instruction; new paragraph inserted after **Explanation's** first paragraph:

For self-modifying code, however, ensure that the **MOVI** instruction executes at least 2 instruction-cycles prior to the instruction that it modifies. This gives the cog time to write the result before it fetches that instruction for execution; otherwise, the unmodified instruction will be fetched and executed.

The **MOVS** instruction; new paragraph inserted after **Explanation's** first paragraph:

For self-modifying code, however, ensure that the **MOVS** instruction executes at least 2 instruction-cycles prior to the instruction that it modifies. This gives the cog time to write the result before it fetches that instruction for execution; otherwise, the unmodified instruction will be fetched and executed.

Page 389:

The NOP instruction:

Additional sentences for the end of the Explanation paragraph:

Because of this, the NOP instruction can never be preceded by a *Condition*, such as IF_Z or IF_C_AND_Z, since it can never be conditionally executed.

Page 397:

Table 5-5:

Table 0-2: Registers	
Register(s)	Description
DIRA, DIRB	Direction Registers for 32-bit port A and 32-bit port B. See the Explanation section of DIRA, DIRB on page 212. The optional “[Pin(s)]” parameter does not apply to Propeller Assembly; all bits of the entire register are read/written at once, unless using the MUXx instructions.
INA ¹ , INB ¹	Input Registers for 32-bit port A and 32-bit port B. (Read-Only). See the Explanation section of INA, INB on page 226. The optional “[Pin(s)]” parameter does not apply to Propeller Assembly; all bits of the entire register are read at once.
OUTA, OUTB	Output Registers for 32-bit port A and 32-bit port B. See the Explanation section of OUTA, OUTB on page 280. The optional “[Pin(s)]” parameter does not apply to Propeller Assembly; all bits of the entire register are read/written at once, unless using the MUXx instructions.
CNT ¹	32-bit System Counter Register. (Read-Only). See the Explanation section of CNT on page 184.
CTRA, CTB	Counter A and Counter B Control Registers. See CTRA, CTB on page 204.
FRQA, FRQB	Counter A and Counter B Frequency Registers. See FRQA, FRQB on page 219.
PHSA ² , PHSB ²	Counter A and Counter B Phase Lock Loop Registers. See PHSA, PHSB on page 285.
VCFG	Video Configuration Register. See VCFG on page 317.
VSCL	Video Scale Register. See VSCL on page 320.
PAR ¹	Cog Boot Parameter Register. See PAR on page 283.

Note 1: For Propeller Assembly, only accessible as a source register (i.e. MOV Dest, Source).

Note 2: For Propeller Assembly, only readable as a source register (i.e. MOV Dest, Source); read-modify-write not possible as a destination register.

Page 402:

The SHL instruction’s explanation:

SHL (Shift Left) shifts *Value* left by *Bits* places **and sets the new LSBs to 0.**

Page 403:

The SHR instruction’s explanation:

SHR (Shift Right) shifts *Value* right by *Bits* places **and sets the new MSBs to 0.**

Page 410:

Added TESTN instruction:

TESTN

Instruction: Bitwise AND a value with the NOT of another to affect flags only.

TESTN *Value1*, <#> *Value2*

Result: Optionally, zero-result and parity of result is written to the Z and C flags.

- *Value1* (d-field) is the register containing the value to bitwise AND with !*Value2*.
- *Value2* (s-field) is a register or a 9-bit literal whose value is inverted (bitwise NOT) and bitwise ANDed with *Value1*.

-INSTR-	ZCRI	-CON-	-DEST-	-SRC-	Z Result	C Result	Result	Clocks
0111001	000i	1111	ddddddddd	sssssssss	Result = 0	Parity of Result	Not Written	4

Explanation

TESTN is similar to ANDN except it doesn't write a result to *Value1*; it performs a bitwise AND NOT of *Value2* into *Value1* and optionally stores the zero-result and parity of the result in the Z and C flags.

If the WZ effect is specified, the Z flag is set (1) if *Value1* AND NOT *Value2* equals zero. If the WC effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits.

Page 419:

In Reserved Word List, added TESTN instruction between TEST and TJNZ:

TEST^a

Amended the following instructions with '#' footnotes:

DIRB^{d#}
INB^{d#}
OUTB^{d#}
MUL^{a#}
MULS^{a#}
ONES^{a#}

And changed the footnote to:

a = Assembly element; s = Spin element; d = dual (available in both languages); # = reserved for future use

Errata Items

Page 18:

Boot Up Procedure, item 2c:

- c. If no EEPROM was detected, the boot loader stops, Cog 0 is terminated, the Propeller chip goes into shutdown mode, and all I/O pins **are** set to inputs.

Page 29:

Table 1-9 XINPUT row:

0	0	XINPUT	Infinite	6 pF (pad only)	DC to 128 MHz Input
---	---	--------	----------	-----------------	----------------------------

...should read:

0	0	XINPUT	Infinite	6 pF (pad only)	DC to 80 MHz Input
---	---	--------	----------	-----------------	---------------------------

Page 181:

Table 4-4, column 1, row 3:

XINPUT	0_0_0_00_010
--------	--------------

...should read:

XINPUT	0_0_1_00_010
--------	--------------

Page 188:

Example code in the **Propeller Assembly Code (Syntax 2)** section:

```
coginit(2, @Update, Pos)
```

...should read:

```
coginit(2, @Update, @Pos)
```

Page 207:

Table 4-7, rows 5 and 6:

Table 0-3: Counter Modes (CTRMODE Field Values)				
CTRMODE	Description	Accumulate FRQx to PHSx	APIN Output*	BPIN Output*
%00100	NCO/PWM single-ended	1	PHSx[31]	0
%00101	NCO/PWM differential	1	PHSx[31]	!PHSx[31]

...should read:

Table 0-4: Counter Modes (CTRMODE Field Values)				
CTRMODE	Description	Accumulate FRQx to PHSx	APIN Output*	BPIN Output*
%00100	NCO single-ended	1	PHSx[31]	0
%00101	NCO differential	1	PHSx[31]	!PHSx[31]

And Table 4-7, row 26:

Table 0-5: Counter Modes (CTRMODE Field Values)				
CTRMODE	Description	Accumulate FRQx to PHSx	APIN Output*	BPIN Output*
%11001	LOGIC A == B	IA ¹ == B ¹	0	0

...should read:

Table 0-6: Counter Modes (CTRMODE Field Values)				
CTRMODE	Description	Accumulate FRQx to PHSx	APIN Output*	BPIN Output*
%11001	LOGIC A == B	A ¹ == B ¹	0	0

Page 209:

Table 4-8 column headings:

Table 4-8: Example Data in Memory																								
L	0				2				3				4				5				6			
W	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
D	40	41	53	74	72	69	6E	67	00	00	C2	FF	F8	24	00	00	11	22	33	44	20	00	00	00

...should read:

Table 0-7: Example Data in Memory																								
L	0				1				2				3				4				5			
W	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
D	40	41	53	74	72	69	6E	67	00	00	C2	FF	F8	24	00	00	11	22	33	44	20	00	00	00

Page 271:

Example:

X := %00101100 | %00001111

...should read:

Example:

```
X := %001011100 ^ %00001111
```

Page 350:

The **CMPSUB** row of the Propeller Assembly Instruction Master Table should read:

CMPSUB	D, S	111000 001i 1111 dddddddd ssssssss	D = S	Unsigned (D => S)	Written	4
--------	------	------------------------------------	-------	-------------------	---------	---

Page 360:

Second paragraph of **Explanation** section:

The Propeller does not use a call stack, so the return address is stored in a different manner; it is recorded at the location of the routine's **RET** command itself. For the **CALL** instruction, the assembler searches for a label that is *Address* with “_ret” appended to it. **It then encodes the address of the label *Address_ret* into the **CALL** instruction as well as the *Address* you specified to jump to.** At run time, when executing the **CALL** instruction, the cog first stores the return address (PC + 1) into the source field of the “**RET**” instruction at *Address_ret* and then jumps to *Address*.

...should read:

The Propeller does not use a call stack, so the return address is stored in a different manner; it is recorded at the location of the routine's **RET** command itself. For the **CALL** instruction, the assembler searches for a label that is *Address* with “_ret” appended to it. **It then encodes the address of that label into the **CALL** instruction's destination (d) field and encodes the *Address* you specified into the source (s) field.** At run time, when executing the **CALL** instruction, the cog first stores the return address (PC + 1) into the source field of the “**RET**” instruction at *Address_ret* and then jumps to *Address*.

And, in the code example:

```
call    Routine
<other code here>
```

...should read:

```
call    #Routine
<other code here>
```

Page 361:

Third paragraph:

The return address is written **to** the *Address_ret* register unless the NR effect is specified.

...should read:

The return address is written **into the source of** the *Address_ret* register unless the NR effect is specified.

Page 362:

In the CMP and CMPS instruction sections:

Z Result
Result = 0

...should be:

Z Result
D = S

Page 363:

Last sentence of CMPS instruction's **Explanation** section:

If the WC effect is specified, the C flag is set (1) if *SValue1* is less than *SValue2*.

In the CMPSUB instruction section:

-INSTR-	ZCRI	-CON-	-DEST-	-SRC-	Z Result	C Result	Result	Clocks
111000	000i	1111	ddddddddd	sssssssss	D = S	Unsigned (D => S)	Not Written	4

...should be:

-INSTR-	ZCRI	-CON-	-DEST-	-SRC-	Z Result	C Result	Result	Clocks
111000	001i	1111	ddddddddd	sssssssss	D = S	Unsigned (D => S)	Written	4

In the CMPSUB Explanation section, the first sentence should read:

CMPSUB compares the unsigned values of *Value1* and *Value2*, and if *Value2* is equal to or **less** than *Value1* then it is subtracted from *Value1* (if the WR effect is specified).

The last two sentences of the last paragraph should be changed to the following:

If the **WC** effect is specified, the C flag is set (1) if **a subtraction is possible (Value1 is equal to or greater than Value2)**. **The result, if any, is written to Value1 unless the NR effect is specified.**

Pages 364 - 365:

In the **CMPX** and **CMPX** instruction sections:

Z Result
Z & (Result = 0)

...should be:

Z Result
Z & (D = S+C)

Page 374:

JMP sets the Program Counter (PC) to *Address* causing execution to **jump that** location in Cog RAM.

...should read:

JMP sets the Program Counter (PC) to *Address* causing execution to **jump to that** location in Cog RAM.

Page 375:

Second paragraph:

The return address is written **to** the *RetInstAddr* register unless the **NR** effect is specified.

...should read:

The return address is written **into the source of** the *RetInstAddr* register unless the **NR** effect is specified.

Page 412 - 413:

In the **WAITPEQ** and **WAITPNE** instruction sections:

Z Result
Result = 0

...should be:

Z Result

And the last sentence of each section, referring to the Z flag, should be removed.