PARALLAX INC

Web Site: www.parallax.com
Forums: forums.parallax.com
Sales: sales@parallax.com
Technical: support@parallax.com

Office: (916) 624-8333
Fax: (916) 624-8003
Sales: (888) 512-1024
Tech Support: (888) 997-8267

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

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

Supplemental Information

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 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.

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 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, 333:

`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*.

New paragraphs at end of **Word Data Declaration (Syntax 2)** section, page 333:

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 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 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.**

Errata Items

Page 181:

Table 4-4, column 1, row 3:

XINPUT	0_0_0_00_010

…should read:

XINPUT	0_0_1_00_010

Page 207:

Table 4-7, row 26:

Table 0-1: 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

...should read:

Table 0-2: 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 271:

Example:

```
X := %00101100 | %00001111
```

...should read:

Example:

```
X := %00101100 ^ %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:

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

...should read:

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

Page 363:

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.**