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

Propeller Micro C (PMC) Specification

Public Release 4/15/2010

Table of Contents

1 – Introduction

Parallax is developing a C-based programming language for the Propeller microcontroller. Though the name is subject to change, it is currently called Propeller Micro C (PMC). The language and supporting system will be integrated into the Propeller Tool software giving developers the ability to freely program Propeller Applications in three languages: C, Spin, and Propeller Assembly.

1.1 – Goals and Target Audience

This project meets a number of goals for Parallax's Education department and those of the company in general. The target audience includes professional engineers, college students, high school students, and hobbyists of the electronic and computer sciences.

- Needs of Parallax Education department
  - Easy to use
  - Free and built-in to the Propeller Tool
  - Provides entry point to high school and college classrooms currently unwilling or unable to use Spin
  - Delivers a natural introduction/transition to Spin and Propeller Assembly
  - Creates long-term platform to build next generation educational tools upon
- Needs of existing and future Parallax customer base
  - Easy to use
  - Free and built-in to the Propeller Tool
  - Attractive to wide customer base including professional engineering shops
  - Directly maintained and supported by Parallax staff
  - Compatible with existing Spin and Propeller Assembly objects

1.2 – Feature Highlights

- Supported directly in the Propeller Tool software with moderate enhancements.
- Based upon many ANSI C89 and some C99 standards
  - PMC is a custom implementation of the C language that embraces C standards that do not conflict with the principles of, or the efficient use of, the Propeller microcontroller. It is designed with careful attention to the hardware, existing languages, and development environment.
  - PMC code looks and feels much like standard C code.
- PMC code translates into Spin and Propeller Assembly code at compile-time and is processed as is normal by the Propeller Tool and the Propeller microcontroller.
  - PMC-based applications run just like standard Spin/Asm-based apps.
  - Existing Propeller Objects (Spin/Asm) are easily and naturally supported.
    - Propeller Applications can be constructed and distributed as a mixture of individual and/or community-developed PMC, Spin, and Propeller Assembly-based objects in a tidy Propeller Archive format compatible with the Propeller Tool.
  - Developers can easily view how their PMC code translates to the Propeller's native Spin language.
- PMC allows the inclusion of Propeller Assembly.
  - Supported in a Propeller-compatible fashion that emphasizes the architecture's intent-of-use.

2 – Overview of the Propeller Micro C Language

The goal of Propeller Micro C (PMC) is to provide developers familiar with the C language the ability to program the Propeller without obscuring the architecture or the principles behind its design. PMC achieves this first by supporting compatible core C features and then extending the language by building in native Propeller features.

Certain features of standard C are not supported by PMC either because there is no corresponding Propeller feature, or because they violate Propeller principles of software design. Parallax has taken great effort to support as many standard C features as is practical while staying true to the spirit of the Propeller.

The following illustrates a summary view of elemental features of the PMC language. Additional detail is provided in section 3 – Language Functional Details.

2.1 – Core C Elements and Propeller Micro C Support

Propeller Micro C implements many of the core ANSI C standards (much of C89 and some of C99). Below is the full listing of core C keywords and operators along with notes regarding support and special details in Propeller Micro C.

Table 1 – Core C Keywords

Keyword	Supported?	Details
auto	no	Not necessary
break	yes	
case	yes	
char	yes	
const	yes	
continue	yes	
default	yes	
do	yes	
double	no	Don't have a double accurate float type
else	yes	
enum	yes	
extern	yes and no	Only used in place of Spin's PUB to have the same effect (as defined by C89). This is optional as it's the default method scope.
float	yes	Supported for constants and variables, though variable support will be through the automatic inclusion and use of the floating point objects.
for	yes	
goto	no	Propeller doesn't support
if	yes	
inline	no	
int	yes	
long	yes	
register	no	Should specify variables in desired order for fast access if that's important, otherwise using register may change variable placement in memory in a way that is not expected by user.

restrict	no	
return	yes	
short	yes	
signed	yes	But not on int, long, or float.
sizeof	yes	Provides compile-time constant.s
static	yes and no	Only used in place of PRI to have the same effect (as defined by C89). Was going to consider this for data, but now may choose a different method.
struct	maybe	Developing possible implementations
switch	yes	
typedef	maybe	Will support if struct is supported
union	maybe	Will support if struct is supported/if implementation is not terribly complicated
unsigned	no	Default signed types have no unsigned counterparts.
void	yes	Generates compiler error if function used in expression or argument (since it returns no usable value)
volatile	no	Not necessary because all variables are assumed to be modified during runtime by any means.
while	yes	
_bool	no	Implemented as simply bool
_complex	no	
_imaginary	no	

Table 2 – Preprocessing Directives

Directive	Supported?	Details
#define	yes	
defined	maybe	Under consideration; may use a simplified and flexible form of #if.
#undef	yes	
#include	no	Use class, see Table 6.
#if	yes	
#elif	yes	
#else	yes	
#endif	yes	
#ifdef	no	Use #if defined() instead
#ifndef	no	Use #if !defined() instead
#line	no	
#pragma	no	Not necessary
#error	no	

Table 3 – Operators

Operator	Supported?	Details
(type)	yes	
!	yes	Boolean NOT
++	yes	
--	yes	
+	yes	

-	yes	
*	yes	
/	yes	
%	yes	Modulus
+=	yes	
-=	yes	
*=	yes	
/=	yes	
%=	yes	
<	yes	
<=	yes	
>	yes	
>=	yes	
==	yes	
!=	yes	
&&	yes	Boolean AND
||	yes	Boolean OR
?:	maybe	
=	yes	
~	yes	Bitwise NOT
&	yes	Bitwise AND
|	yes	Bitwise OR
^	yes	Bitwise XOR
<<	yes	
>>	yes	
&=	yes	
|=	yes	
^=	yes	
<<=	yes	
>>=	yes	

Table 4 – Value Indicators

Indicator	Supported?	Details
(none)	yes	Decimal (base-10) value.
. (decimal point) or e	yes	Floating-point value.
0	no	Octal (base-8) is not supported.
0x	yes	Hexadecimal (base-16) value follows.
' ' (single quotes)	yes	Character included.
" " (double quotes)	yes	String of characters included.

Table 5 – Special Characters and Escape Sequences

Directive	Supported?	Details
{ }	yes	
;	yes	
\	yes	
&	yes	Address of
*	yes	Indirection (ex: *p)
[ ]	yes	Array index indicators
.	yes	Member of
->	yes	Member of pointer
()	yes	Function call

\a	yes	(7)
\b	yes	(8)
\f	yes	(12)
\n	yes	(10)
\r	yes	(13)
\t	yes	(9)
\v	yes	(11)
\'	yes	(39)
\"	yes	(34)
\?	maybe	Why is this needed? What special purpose is '?' used for?
\\	yes	(92)
\ooo	yes	Yes, but as decimal, not octal
\xh	yes	

2.2 – Additional Propeller Micro C Elements

In addition to the core C keywords (section 2.1), Propeller Micro C includes natural support for most Propeller-specific features via keywords derived from Spin or otherwise. These features are available automatically without the need to "include" any header files.

Table 6 – Additional Keywords

Directive	Details
class	Provides access to Propeller Objects.

Table 7– Additional Keywords (Spin-Derived)
(~~Stricken~~ keywords have alternate or no support)

Spin Command	Details
abort	Supported, but collides with C function in stdlib.h – may have to resolve this
byte	Feature supported as an additional type
~~{symbol} byte data {[count]}~~	Feature may ??? be supported via byte type syntax in data
byte [baseaddress] {[offset]}	Supported. Direct memory access, section 3.8.5.1
~~*symbol.*byte {[offset]}~~	Not Supported.
~~bytefill~~	Implemented as memset(); see section 3.8.5.2
~~bytemove~~	Implemented as memmove(); see section 3.8.5.3
chipver	See section 3.12
clkfreq	See section 3.12
clkmode	See section 3.12
clkset	See section 3.12
cnt	See section 3.13
coginit	See section 3.12
cognew	See section 3.12
cogstop	See section 3.12
~~constant~~	Not supported; very specialized, rare case
ctra/ctrb	See section 3.13
dat	in some appropriate form
dira	See section 3.13
file “filename”	married with DAT block
float	Supported; merges with C float type

frqa/frqb	See section 3.13
ina	See section 3.13
lockclr	See section 3.12
locknew	See section 3.12
lockret	See section 3.12
lockset	See section 3.12
long	Supported; merges with C long type
~~{symbol} long data {[count]}~~	Feature may ??? be supported via long type syntax in data
long [baseaddress] {[offset]}	Supported. Direct memory access, section 3.8.5.1
~~longfill~~	Implemented as memset(); see section 3.8.5.2
~~longmove~~	Implemented as memmove(); see section 3.8.5.3
lookdown	See section 3.12
lookdownz	See section 3.12
lookup	See section 3.12
lookupz	See section 3.12
outa	See section 3.13
par	See section 3.13
phsa/phsb	See section 3.13
~~pri~~	Feature supported as "static" storage type on method declaration
~~pub~~	Feature supported as "extern" storage type on method declaration
reboot	See section 3.12
result	See section 3.13
return	Supported; merges with C return command
round	through some directive?
spr	See section 3.13
strcomp	See section 3.12
string	support just like method calls - check comparable C syntax
strsize	See section 3.12
trunc	through some directive?
vcfg	See section 3.13
vscl	See section 3.13
waicnt	See section 3.12
waitpeq	See section 3.12
waitpne	See section 3.12
waitvid	See section 3.12
word	Feature supported as an additional type
~~{symbol} word data {[count]}~~	Feature may ??? be supported via word type syntax in data
word [baseaddress] {[offset]}	Supported. Direct memory access, section 3.8.5.1
~~*symbol*.word {[offset]}~~	Not supported.
~~wordfill~~	Implemented as memset(); see section 3.8.5.2
~~Wordmove~~	Implemented as memmove(); see section 3.8.5.3

Table 8– Additional Operators

Operator	Details
**	Multiply, return high
**=	Multiply, return high (assign)
~^	Sign-extend bit 7

~~^	Sign-extend bit 15
~	Post-clear to 0
~~	Post-set to -1
??	Random Number
^^	Square Root
#>	Limit minimum
#>=	Limit minimum (assign)
<#	Limit maximum
<#=	Limit maximum (assign)
<:=	Less than (assign)
>:=	Greater than (assign)
<==	Less than or equal (assign)
>==	Greater than or equal (assign)
===	Equal (assign)
!==	Not equal (assign)
&&=	Boolean AND (assign)
||=	Boolean OR (assign)
<+	Rotate left
<+=	Rotate left (assign)
+>	Rotate right
+>=	Rotate right (assign)
~>	Shift arithmetic right
~>=	Shift arithmetic right (assign)
><	Reverse
><=	Reverse (assign)
|<	Decode
>|	Encode

Table 9 – Additional Value Indicators

Indicator	Details
0b	Binary (base-2) value follows.
0q	Quaternary (base-4) value follows.

2.3 – C Libraries

The standard C libraries provide many functions that have no corresponding feature in the Propeller architecture and many other functions that are already covered by existing Propeller objects. The remaining C library functions that are considered vital to Propeller operation are supported by PMC as built-in extensions; no "standard" libraries need be included in an application just to use them.

Since existing Propeller objects are openly accessible by PMC, other functions normally provided by standard C libraries can be provided by existing Propeller objects instead.

Table 10 – PMC Built-In Functions

Function	Details
memmove( )	Enhanced memory move combines support of Spin's bytemove, wordmove, and longmove. See section 3.8.5.3
memset( )	Enhanced memory set combines support of Spin's bytefill, wordfill, and longfill. See section 3.8.5.3

3 – Language Functional Details

This section provides Propeller Micro C implementation details.

3.1 – Fundamentals

A PMC-based application consists of building block objects (written in PMC, Spin, and Assembly) whose collection of methods and data work together to achieve the application's goal. Objects for common tasks are included in the Propeller Library while others written by individual developers are freely available in the Propeller Object Exchange. The PMC and Spin languages promote object creation and sharing, by design, in their structure and function.

Each PMC object consists of executable C code organized into blocks called methods– functions that logically belong to the object. These methods invoke each other to achieve the desired task and many can be called from outside the object by methods in other objects that specifically include them.

In addition, objects may choose to parallel-process certain code (methods or assembly code) by "launching" them into other cogs (processors) on the Propeller.

The architectural features of the Propeller and its native languages (Spin and Propeller Assembly) are made available in PMC through tightly integrated extensions of the C language. The concepts of such things can be learned from existing Propeller documentation outside of this specification.

3.1.1- PMC Object Concept

PMC, like Spin, is a high-level language that uses objects. Objects are conceptual devices that encapsulate data, and the code meant to operate on that data, into one well-defined entity. Objects embody the essence of an entity– its attributes, behavior, and its interface (input and output) with the environment outside of it.

The PMC and Spin languages embrace the core concept of objects– encapsulation. However, the implementation of encapsulation is different from that of many object languages. Specifically, a Propeller object is a source code file, and a source code file is a Propeller object. Everything inside the file (comments, methods, assembly, and data) is part of that object and describes its nature. Thus, a file is the encapsulation of an object, and the name of the object is, by definition, the file's name.

In contrast, while other object languages may be used in a similar way, the trend is to pack multiple objects, called classes, into a single file. Arguably, there are some advantages to that, but the practice leads to name collisions in higher-level objects– a programming obstacle that PMC and Spin completely eliminates.

3.1.2 – PMC Object Structure

Here is an example of a PMC-based object. As an example, it contains comments, directives, a C method, and an assembly method.

Example 1 – PMC Object

Comments

```
/* This object is just for example purposes
   It really does nothing very useful. */
```

Directives

```
#define baud 115_000

class "Parallax Serial Terminal" pst;
```

C Method

```
void main()
  {
  int counter;                                  //A counter variable

  pst.Start(baud);                              //Start pst object
  pst.Str("Propeller Counting\n");              //Display heading on pst
  for(counter = 0; counter < 1000; counter++)   //Loop 1000 times
    {
    pst.PositionX(0);                           //Move to start of line
    pst.Dec(counter);                           //Display current count
    }
  cognew(toggle, 0);                            //Launch assembly method
  }
```

Assembly Method

```
asm toggle
  {
  long time = 0
  long delay = 100_000

                  mov     dira, # |< 0          //Set I/O 0 to output
                  mov     time, cnt             //time is cnt + delay
                  add     time, delay

  twiddle         xor     outa, # |< 0          //outa[0] = !outa[0]
                  waitcnt time, delay           //wait for delay
                  jmp     #twiddle              //loop
  }
```

The *class* declaration creates a symbolic name, **pst**, for a single instance of the preexisting object "Parallax Serial Terminal." This object can later be referred to as **pst** and it's methods can be accessed via the **pst.**methodname format.

The **main** method contains executable code and is declared using a typical C format. This method is special only in that it is called "main;" every PMC object must have one method called **main** which will become the first method executed if that object is the top object in the application.

The statements inside **main** follow typical C conventions with the addition of the **pst** object references. Read the comments to the right of the code for a description of each line.

The **toggle** method is the most notable difference between standard C and PMC. It is declared as type *asm* which means it contains Propeller Assembly code and data. Asm methods must only contain Propeller Assembly code, variable and data declarations, and comments. Asm methods never have declared parameters; however, since they must always be launched into a cog via a **cognew** or **coginit** they naturally receive a 14-bit value in their cog's **par** register that can be used to convey a memory address to operate on.

Directives and methods may be defined in any order. If this object were compiled as the top object in the application, the **main** method (regardless of its position in the file) would be the first method the application executes.

3.1.3 – PMC Processing

When a PMC object is compiled by the Propeller Tool, the source code is translated to Spin source code and then compiled (tokenized and assembled) by the Propeller Tool's Spin Compiler. This happens as one seamless process, as if the object were already written in the Propeller's native language(s), Spin and Propeller Assembly.

The example in section 3.1.2, when compiled, would first be silently translated into the following Spin/Assembly code that would immediately afterwards be compiled in the normal fashion.

Example 2 – Spin Translation of PMC Object

Comments

```
{ This object is just for example purposes
   It really does nothing very useful. }
```

Constants

```
CON

  Baud = 115_000
```

Objects

```
OBJ

  pst : "Parallax Serial Terminal"
```

Spin Method

```
PUB main | counter

  pst.Start(baud)                                 'Start pst object
  pst.Str(string("Propeller Counting", 13))       'Display heading on pst
  repeat counter from 0 to 999                    'Loop 1000 times
    pst.PositionX(0)                              'Move to start of line
    pst.Dec(counter)                              'Display current count
  cognew(@toggle, 0)                              'Launch assembly method
```

Data and Assembly

```
DAT

  toggle          org

                  mov     dira, # |< 0          'Set I/O 0 to output
                  mov     time, cnt             'time is cnt + delay
                  add     time, delay

  twiddle         xor     outa, # |< 0          'outa[0] = !outa[0]
                  waitcnt time, delay           'wait for delay
                  jmp     #twiddle              'loop

  long  time      0
  long  delay     100_000
```

3.2 – Identifiers

Identifiers are alphanumeric names either created by the compiler (keywords) or by the code developer (user-defined words) representing constants, variables, types, methods, etc.

3.2.1 – Identifier Rules

Identifiers must fit the following rules:

- Begins with a letter (a – z) or an underscore '_'.
- Contains only letters, numbers, and underscores (a – z, 0 – 9, _); no spaces allowed.
- Must be 30 characters or less.
- Is unique within the given scope; not an existing keyword or user-defined identifier. Identifiers are not case-sensitive in PMC.

3.2.2 – Identifier Types

The Propeller is a 32-bit device. This limits the maximum size of natural storage to 4 bytes in length. Various types can be derived from this as either 1-byte, 2-byte, and 4-byte entities.

Identifier types, storage requirements, and ranges supported by PMC are shown in Table 11.

Table 11 – Identifier Types

Type	Size	Range in Decimal (Hexadecimal)
void	n/a	n/a
bool	1 byte	-1 [true] or 0 [false] ($FFFFFFFF[1] to $0)
char	1 byte	0 to 255 ($00 to $FF)
byte	1 byte	0 to 255 ($00 to $FF)
signed char	1 byte	-128 to 127 ($FFFFFF80[2] to $7F)
signed byte	1 byte	-128 to 127 ($FFFFFF80[2] to $FF)
short	2 bytes	0 to 65535 ($0000 to $FFFF)
word	2 bytes	0 to 65535 ($0000 to $FFFF)
signed short	2 bytes	-32768 to 32767 ($FFFF8000[2] to $7FFF)
signed word	2 bytes	-32768 to 32767 ($FFFF8000[2] to $7FFF)
int	4 bytes	-2,147,483,648 to 2,147,483,647 ($80000000 to $7FFFFFFF)
long	4 bytes	-2,147,483,648 to 2,147,483,647 ($80000000 to $7FFFFFFF)
float	4 bytes	-3.4e+38 to 3.4e+38 ($FF7FC99E to $7F7FC99E)

1 Non-zero Boolean values are promoted to -1, which is $FFFFFFFF at time of evaluation.
2 Signed types smaller than long are sign-extended at the time of evaluation.

The types **byte**, **char**, **word**, and **short** are unsigned. By using the prefix **signed**, they become signed types that use the same storage size as their unsigned counterparts.

The types **long**, **int**, and **float** are signed. They can not be made to be unsigned.

The Boolean type, **bool**, is implemented as a signed byte. It has two values, **true** and **false**, that correspond to -1 (or non-zero) and 0, respectively. NOTE: Due to architectural design, **true** in PMC differs in sign from ANSI C **true**; however, in both cases any non-zero value is treated as **true**.

3.2.2.1 – Translation of Void Type

The **void** type doesn't have a corresponding feature in Spin; every method returns a value (defaults to 0) regardless of whether it will be used or not. To effectively translate **void** methods, the PMC translator simply ensures that void methods don't modify the **result** local variable and references to the method don't expect a value.

Void parameter lists and casting are disallowed.

3.2.2.2 – Translation of Bool Type

The **bool** type doesn't have a directly corresponding feature in Spin; any integer value can be treated as a Boolean value where 0 is false and non-zero is true. The PMC Translator treats **bool** types as **byte** values when used with Boolean operators, and as ~**byte** (sign-extended byte value) when used with mathematical and bitwise operators.

3.2.2.3 – Translation of Byte and Char Types (Unsigned/Signed)

The **byte** and **char** types both translate to **byte**. If signed, they are both translated as ~**byte** (sign-extended byte value) wherever evaluated, and truncated to **byte** when necessary.

3.2.2.4 – Translation of Word and Short Types (Unsigned/Signed)

The **word** and **short** types both translate to **word**. If signed, they are both translated as ~~**word** (sign-extended word value) wherever evaluated, and truncated to **word** when necessary.

3.2.2.5 – Translation of Long and Int Types

The **long** and **int** types both translate to **long**. They are always signed.

3.2.2.6 – Translation of Float Type

The **float** type translates differently depending on how it is used. If it is a constant, it is treated naturally by the compiler when used in a constant expression. If it is a variable, or is used in a variable expression, the floating point objects are included and related methods are called to evaluate it.

3.3 – Literals

In PMC code, literal values can be represented in binary (base-2), quaternary (base-4), decimal (base-10), hexadecimal (base-16), a characters, or a string of characters.

Numerical values can also use underscores, '_', as group separators for clarification. The underscore separator can be used in place of commas (such as in decimal values) or to form logical groups of bits, bytes, words, etc.

Table 12 – Forms of Literal Values

Base	Type of Value	Indicator	Examples
2	Binary	0b	0b1010, 0b1110_1001
4	Quaternary	0q	0q2130_3311, 0q3311_1012
10	Decimal (integer)	none	1024, 2_147_483_647
10	Decimal (floating-point)	. (decimal point) or e	0.70712, 1e6
16	Hexadecimal	0x	0x1AF, 0xFFAF_126D
n/a	Character	' ' (single quotes)	'A', '9'
n/a	String	" " (double quotes)	"Testing", "Hello\n"

Except for floating point literals, numerical literals are always treated as a **long** type and are automatically cast to the destination type in expression results. Floating point literals are of type **float**.

String literals are stored internally as an array of **char** with a null terminator, \0.

3.3.1.1 – Translation of Literals

Literal values have simple translations.

Table 13 – Literals

Example	Notes	Translation
0bn	Binary	%n
n / n.n / nen	Decimal (integer/floating-point)	n / n.n / nen
0qn	Quaternary	%%n
0xn	Hexadecimal	$n
'c'	Character	"c"
"s"	String	"s"

3.4 – Variables

Variables are declared in the form: *type identifier* < = *literal* >;

Type is one of the valid types from section 3.2.2 – Identifier Types.

Identifier is a unique name for the variable; see section 3.2.1 – Identifier Rules.

The optional *literal* field initializes the variable to the specified value.

3.4.1.1 – Translation of Variables

The following table demonstrates some variable declarations.

Table 14 – Variables

Example	Translation
int temp;	VAR long temp
short count = 100;	VAR word count count := 100
char smallval	VAR byte smallval

3.5 – Constants

There are two types of constants; normal and enumerated.

3.5.1 – Normal Constants

Normal constants, simply called constants, are declared in a form very similar to variables:

const *type identifier* = *literal*;

All fields are exactly the same as with variables except that the declaration is preceded by the type qualifier **const**, to indicate it is a constant, and the *literal* field is required.

References to constants later in the code are limited to read-only access; constants can not be modified.

3.5.1.1 – Translation of Normal Constants

The following table demonstrates some constant declarations.

Table 15 – Constants

Example	Translation
const int speed = 80_000_000;	CON speed = 80_000_000
const short count = 100;	CON count = 100

const char smallval = 'A';	CON smallval = 'A'

3.5.2 – Enumerated Constants

Enumerated constants are a collection of identifiers with an associated type that have certain discrete integer values. The declared enumeration type of the group can also be used to create variables which can consequently only be set to one of those discrete values.

Enumerated types and constants are declared using the following format:

enum *<type_identifier> { identifier_list };*

The optional *type_identifier* is a user-defined name for this enumerator group.

The *identifier_list* is a comma-delimited list of user-defined identifiers that each represents a discrete value. These identifiers become like normal constants where their individual values are automatically set as incrementing integers starting from 0. The values themselves don't matter as much as the identifier names, however, the values can be manually set as well using the form *identifier* = *value* as a list entry.

3.5.2.1 – Translation of Enumerated Constants

Table 16 – Enumerated Constants

Example	Translation
enum { off, on };	VAR off, on
enum mode { norm, run = 0, alt, suspended };	VAR norm, #0, run, alt, suspended

3.6 – Methods

Methods are functions that belong to an object and contain code that performs a given task. In PMC, there are C methods and Assembly methods. Since most methods are C methods, this text simply refers to them as "methods," and refers to Assembly methods as "ASM methods."

All methods in an object are accessible by other methods within that object and most are accessible from other objects; however, methods may optionally be scope-limited to prevent use from outside the object. Every object contains at least one method, called *main*.

3.6.1 – C Methods

C method declarations follow this syntax:

<storage_class> <type> identifier(*<parameter_list>*) { *declarations_and_statements* }

The optional *storage_class* is either **extern** or **static**. **Extern**, the default, makes the method accessible to other objects. **Static** makes the method scope-limited to the object; it is "private" to the object so it is not accessible to other objects.

Type is one of the valid types from section 3.2.2 – Identifier Types.

Identifier is a unique name for the method; see section 3.2.1 – Identifier Rules.

Parameter_list is a comma-delimited list of parameters the method accepts. Each parameter is listed in the form of variable and array declarations (see section 3.4 – Variables and section 3.8.2 – Arrays), except no initializers are allowed. When executed, the parameters are initialized with the values of the arguments provided by

the calling reference. All arguments in PMC are passed to parameters by value, not by reference; changing the parameter's value within the method has no effect on the source argument.

Declarations_and_statements make up the body of the method and are comprised of local variables, global variables and executable code.

The **return** command can be used to exit the method from other than the end. Additionally, **return** supports an optional parameter, the *returnvalue*, which gets stored in the method's pre-defined **result** variable and returned to the caller.

3.6.1.1 – Translation of C Methods

Table 17 – C Methods

Example	Translation

void main()
{
declarations
statements
}
	PUB main
	declarations
	statements

static int dosomething(x)
{
int i, j;
statements
}
	PRI dosomething(x) | i, j
	{
	statements
	}

3.6.2 – ASM Methods

ASM method declarations follow this syntax:

asm *identifier* { *data_and_instructions* }

Identifier is a unique name for the ASM method; see section 3.2.1 – Identifier Rules.

Data_and_instructions make up the body of the ASM method and are comprised of variables, constants, and executable Propeller Assembly code.

ASM methods never have declared parameters, but they always get launched into a cog via a **cognew** or **coginit** command that delivers a 14-bit value to their cog's **par** register which can be used to convey a memory address to operate on.

ASM methods are always of type **void** because they don't return any value to the caller; except possibly through the memory pointed to by the 14-bit *mempointer* parameter of the **cognew** or **coginit** command that launched it. There is no need to specify the **void** type for ASM methods since they can never be anything else.

3.6.2.1 – Translation of ASM Methods

Table 18 – ASM Methods

Example	Translation

asm toggle
{
declarations
instructions
}
	DAT
	toggle
	org 0
	instructions
	declarations

3.7 – Operations

There are three categories of expression operations: Mathematical, Comparison, and Bitwise. Mathematical operations perform addition, subtraction, etc., and usually involve two **int** values, but can include smaller types (like **byte** or **word**) or the more complex **float** type. Comparison operations are like mathematical operations except that they determine if one of the two values is greater, lesser, or equal to the other. Bitwise operations use any numerical type, except **float**, to perform logical operations with every set of two corresponding bits between two values.

Expressions can include one, two, or all three categories of operations into one statement.

3.7.1 – Mathematical Operations

Except with the **float** type, all mathematical operations are performed using signed 32-bit integer math. Types smaller than an **int** are converted to an **int** at the time of the operation, and results are converted back to smaller types when necessary.

Note that division with integer operands returns an integer result. In contrast, division with one or two floating point operands returns a floating point result. The modulus operator, %, always requires integer operands.

3.7.1.1 – Translation of Mathematical Operators

Most mathematical operators have no need for translation; they are equally represented in both PMC and Spin, or they are completely derived from Spin. Table 19 shows the details.

Table 19 – Mathematical Operators

Operator	Notes	Translation
=	Assign	x = y, x := y
++	Pre-/Post-Increment [1]	++x, x++
--	Pre-/Post-Decrement [1]	--x, x--
+	Unary Positive / Binary Add [2]	+x, x + y
+=	Add (assign) [2]	x += y
-	Unary Negative / Binary Subtract [2]	-x, x - y
-=	Subtract (assign) [2]	x -= y
*	Multiply [2]	x * y
*=	Multiply (assign) [2]	x *= y
**	Multiply, return high [1, 3]	x ** y
**=	Multiply, return high (assign) [1, 3]	x **= y
/	Divide [2]	x / y
/=	Divide (assign) [2]	x /= y
%	Modulus [2]	x // y
%=	Modulus (assign) [2]	x //= y
~^	Sign-extend bit 7 [1, 3]	~x
~~^	Sign-extend bit 15 [1, 3]	~~x
~	Post-clear to 0 [1, 3, 4]	x~
~~	Post-set to -1 [1, 3]	x~~
??	Random Number [1, 3]	?x, x?
^^	Square Root [2, 3]	^^x
abs()	Absolute [2]	||x
#>	Limit minimum [1, 3]	x #> y
#>=	Limit minimum (assign) [1, 3]	x #>= y

<#	Limit maximum [1, 3]	x <# y
<#=	Limit maximum (assign) [1, 3]	x <#= y

[1] Integers only.

[2] Integer operators are used when both operands are one of the integer types. The floating-point library's math methods are used when at least one of the operands is a **float** type.

[3] New operator in C; derived from Spin.

[4] Post-clear (~) differentiates itself from Bitwise NOT (~) by being postfix rather than prefix.

3.7.2 – Comparison Operations

All comparison operations are performed using signed 32-bit integer math. Types smaller than an **int** are converted to an **int** at the time of the operation.

The Boolean operators (&&, ||, and !) promote non-zero values to **true**.

Boolean comparisons using && and || operators always evaluate both operands; no short-circuit evaluation is performed.

The result of a comparison is always a **bool**.

3.7.2.1 – Translations of Comparison Operators

Many comparison operators have no need for translation; they are equally represented in both PMC and Spin, or they are completely derived from Spin. Note that there are some incompatible character patterns between ANSI C and Spin– in those cases, a new character pattern is used for the operator is used to resolve issue. Table 20 shows the details.

Table 20 – Comparison Operators

Operator	Notes	Translation
<	Less than	x < y
<:=	Less than (assign) [1]	x <= y
>	Greater than	x > y
>:=	Greater than (assign) [1]	x >= y
<=	Less than or equal	x =< y
<==	Less than or equal (assign) [1]	x =<= y
>=	Greater than or equal	x => y
>==	Greater than or equal (assign) [1]	x =>= y
==	Equal	x == y
===	Equal (assign) [1]	x === y
!=	Not equal	x <> y
!==	Not equal (assign) [1]	x <>= y
!	Boolean NOT	NOT x
&&	Boolean AND	x AND y
&&=	Boolean AND (assign) [1]	x AND= y
||	Boolean OR	x OR y
||=	Boolean OR (assign) [1]	x OR= y

[1] New operator in C; derived from Spin.

3.7.3 – Bitwise Operations

All bitwise operations are performed using the 32-bit integer type. Types smaller than an **int** are converted to an **int** at the time of the operation, and results are converted back to smaller types when necessary.

3.7.3.1 – Translations of Bitwise Operators

Many bitwise operators have no need for translation; they are equally represented in both PMC and Spin, or they are completely derived from Spin. Table 21 shows the details.

Table 21 – Bitwise Operators

Operator	Notes	Translation
~	Bitwise NOT	!x
&	Bitwise AND	x & y
&=	Bitwise AND (assign)	x &= y
|	Bitwise OR	x | y
|=	Bitwise OR (assign)	x |= y
^	Bitwise XOR	x ^ y
^=	Bitwise XOR (assign)	x ^= y
<<	Shift left	x << y
<<=	Shift left (assign)	x <<= y
>>	Shift right	x >> y
>>=	Shift right (assign)	x >>= y
<+	Rotate left [1]	x <- y
<+=	Rotate left (assign) [1]	x <-= y
+>	Rotate right [1]	x -> y
+>=	Rotate right (assign) [1]	x ->= y
~>	Shift arithmetic right [1]	x ~> y
~>=	Shift arithmetic right (assign) [1]	x ~>= y
><	Reverse [1]	x >< y
><=	Reverse (assign) [1]	x ><= y
|<	Decode [1]	x |< y
>|	Encode [1]	x >| y

[1] New operator in C; derived from Spin.

3.8 – Memory Access

PMC provides access to memory through variables, arrays, pointers, structures, and direct access. Except for pointer and structure syntax, all of these features are naturally available through Spin so very little translation is performed.

3.8.1 – Variables

Variables are similar in both PMC and Spin. See section 3.2 – Identifiers, and section 3.4 – Variables for details and translation information.

3.8.2 – Arrays

Arrays are declared using the syntax:

type identifier [*size*] <[*size*]…> <, *identifier* [*size*] <[*size*]…>…>;

Valid *types* are defined in section 3.2.2 – Identifier Types.

The *identifier* must be a unique identifier as defined by section 3.2.1 – Identifier Rules.

The *size* is an integer constant indicating how many elements of *type* are contained in the array *identifier*. Optionally, additional *size* attributes can be appended to create multi-dimensional arrays.

Arrays can also be initialized using an initializer list in the following form:

type identifier [<*size*>] <[*size*…]> = { *listofvalues* };

Note that *size* is optional in the first dimension and the final array size, if not specified, is determined by the size of the *listofvalues*.

Listofvalues is a comma-delimited list of values, one value per element in the array. *Listofvalues* may be a simple string, "*charstring*", without the { }, if *type* is **char**.

To access a particular element in an array, use the syntax: *identifier* [*offset*] {[*offset*]…}

Note that you may also use pointer syntax, see section 3.8.3 – Pointers.

Note that *offset* is in the range 0 to *size*-1. There is no range checking performed at run-time; it is up to the developer to always ensure that the *offset* is within the declared range of the array.

Array access syntax can actually be used on any identifier that points to memory, not just those that were declared as arrays. This allows indexing through any memory starting at the address of the identifier and in units based on the identifier type.

3.8.2.1 – Translation of Arrays

The following table demonstrates some array declaration and accessing possibilities.

Table 22 – Arrays and Indexes

Example	Notes	Translation
int temp[25];	Single-dim array declaration	VAR long temp[25]
short someval[3][5];	Multi-dim array declaration	VAR word someval[3 * 5]
temp[0] = 1024;	Reference	temp[0] := 1024
temp[idx] = x;	Reference	temp[idx] := x
someval[0][4] = 50;	Reference to multi-dim array	someval[0 * 5 + 4] := 50
int myarray[3] = { 10, 20, 30 };	Initialized array declaration	VAR long myarray[3] myarray[0] := 10 myarray[1] := 20 myarray[2] := 30

3.8.3 – Pointers

Pointers simply provide an alias to a location in memory and may be used as an alternative to arrays.

In PMC, like ANSI C, arrays are really pointers, so referencing just the array identifier without an index returns the address of the 0th element. In Spin, however, the array identifier without an index returns the contents of the 0th element.

Pointers are declared using the syntax: *type *identifier*

Valid *types* are defined in section 3.2.2 – Identifier Types.

The *identifier* must be a unique identifier as defined by section 3.2.1 – Identifier Rules.

Pointers are referenced using the syntax: **identifier* --or-- *(*identifier* + *offset*)

Note that you may also use array syntax, see section 3.8.2 – Arrays.

3.8.3.1 – Translation of Pointers

The following table demonstrates some array and pointer declarations and accessing possibilities.

Table 23 – Pointers

Example	Notes	Translation

int temp[25], *p;
p = &temp;
 	
temp[0] = 100;
temp[24] = 200;
temp[12] = temp[24] * 2;
 	
*p = 100;
*(p+24) = 200;
*(&temp + 12) = *(p+24) * 2;
	Integer array and pointer.
	
	The array references and the pointer references do the same thing.
		VAR long temp[25], p
		p := @temp
		
		temp[0] := 100
		temp[24] := 200
		temp[12] := temp[24] * 2;
		
		long[p] := 100
		long[p][24] := 200
		long[@temp][12] := long[p][24] * 2

Continuing with the example above, and assuming x is an integer variable or constant, the following references are equivalent:

```
&temp[x],  temp+x,  p+x                  //pointers to the x-th array element
```

And the following references are equivalent:

```
temp[x],  *(temp+x),  *(p+x),  p[x]      /the x-th array element
```

Pointers can be set to 0 or NULL to indicate they are invalid or yet unused.

3.8.4 – Structures

Structures are groupings of data that make up a logical unit. The data entities in a structure are called members. Members are variables, arrays, or pointers with standard types (**char**, **int**, etc.) or can even be other structures.

Structures are declared using the following format:

struct <*tag*> { *member_list* } <*identifier_list*>;

The optional *tag* is an identifier of the derived type that is this structure. The *tag* can be used later, for example, to declare variables or method parameters of this structure type.

The *member_list* is a semicolon-separated list of variable declarations using the form:

type identifier;;

The optional *identifier_list* is a comma-delimited list of identifiers to declare as this structure type.

3.8.4.1 – Translation of Structures

The following table demonstrates some structure declarations, references, and their translations.

Table 24 – Structures

Example	Notes	Translation

struct date
{
byte month;
byte day;
word year;
};
	Declaration of date structure with three members.	<no code generated since no variables declared of type date>

struct date today;	Declare today as a	VAR

	structure of type date (assuming declaration above)	byte today_month byte today_day word today_year
today.month = 4;	Set member.	today_month := 4
struct date dates[5];	Declare array of date structures.	VAR byte dates_month[5] byte dates_day[5] word dates_year[5]
dates[2].year = 2010;	Set member of structure element.	dates_year[2] := 2010
dates[0] = dates[2];	Set structure element contents to that of another structure element.	dates_month[0] := dates_month[2] dates_day[0] := dates_day[2] dates_year[0] := dates_year[2]

3.8.5 – Direct Access

Direct access to memory is achieved through various PMC features.

3.8.5.1 – Typed Direct Access

PMC supports the use of a *type* as an array identifier for direct memory access. This can be used in place of a variable, array, or pointer identifier for reading and writing values in memory.

The syntax is: *type*[*base*] <[*offset*]>

The *type* can be any identifier type except **void**.

Base is the starting address to access as a *type* value.

The optional *offset* indicates how many units of size *type* to index beyond the *base* address.

When used to read memory, the type returned is always *type*. When used to write memory, the type written is the value converted to *type*, if necessary.

3.8.5.1.1. – Translation of Typed Direct Access

The following table demonstrates some array, pointer, and typed direct access possibilities.

Table 25 – Typed Direct Access

Example	Notes	Translation
int temp[25], *p; p = &temp;	Integer array and pointer.	VAR long temp[25], p p := @temp

temp[0] = 100;
temp[24] = 200;
temp[12] = temp[24] * 2;
 	
*p = 100;
*(p+24) = 200;
*(&temp + 12) = *(p+24) * 2;
 	
int[&temp] = 100;
int[p][24] = 200;
int[&temp][12] = int[p][24] * 2;
	The array references, pointer references, and first three typed direct access references do the same thing. The last two typed direct accesses reference an address via a constant value that doesn't relate to any existing identifier.	temp[0] := 100
		temp[24] := 200
		temp[12] := temp[24] * 2;
		
		long[p] := 100
		long[p][24] := 200
		long[@temp][12] := long[p][24] * 2
		
		long[@temp] := 100
		long[p][24] := 200
		long[@temp][12] := long[p][24] * 2

*p = int[0] *p = byte[4]		long[p] := long[0] long[p] := byte[4]

3.8.5.2 – memset( )

The **memset**() function in PMC is an enhanced form of its namesake in the Standard ANSI C Library. Its syntax is:

```
void *memset( void *address, int value, int count );
```

This function copies *value* into *count* elements of memory starting at *address* and returns a pointer to *address*. The size of each element depends on the type of the *address* parameter provided.

Table 26 – *Address* Type and Resulting memset() Operation

Type of *Address* Parameter Provided	*Value is*	Operation Performed
void, bool, char, byte, signed char, signed byte, or non-typed constant	Truncated to byte	Truncated *value* is copied to *count* bytes of memory starting at *address*.
short, word, signed short, or signed word	Truncated to word	Truncated *value* is copied to *count* words of memory starting at word-aligned *address*.
int, long, or float	Unchanged	*Value* is copied to *count* longs of memory starting at long-aligned *address*.

The adjustable element size feature of **memset**() provides a deterministic optimization on the Propeller that results in fast operation.

When a constant is entered as the *address,* it will be treated as a **byte** unless it is cast to another type. Example: `memset((word) 1244, 65000, 20);` will copy the value 65000 to 20 words of memory starting at address 1244 (byte locations 1244 through 1283). Note that setting the *address* to `(word) 1245` will also have the same results since the address 1245 will automatically be word-aligned to 1244.

3.8.5.2.1. – Translation of memset( )

The following demonstrates **memset**() translation.

Table 27 – Memset()

Example	Notes	Translation

char x[10];
int temp[25];
 	
memset(x, 'A', 10);
memset(temp, 75000, 25);
	Character and integer arrays are filled with 'A' and 75000, respectively.	VAR byte x[10]
		VAR long temp[25]
		
		bytefill(@x, "A", 10)
		longfill(@temp, 75000, 25)

3.8.5.3 – memmove( )

The **memmove**() function in PMC is an enhanced form of its namesake in the Standard ANSI C Library. Its syntax is:

```
void *memmove( void *destination, void *source, int count );
```

This function copies *count* elements from memory starting at *source*, stores them in memory starting at *destination* and returns a pointer to *destination*. The size of each element is that of the smallest type of the *destination* and *source* parameters provided. If both *destination* and *source* parameters are of the same type, the element size will be of that type.

Table 28 – *Address* Type and Resulting memmove() Operation

Type of Parameter Provided for		*Element size is*	Operation Performed
Destination [1]	***Source*** [1]		
byte-sized	byte, word, or long-sized	byte	*Count* byte-sized elements are copied from *source* into *destination*.
word-sized	byte-sized	byte	*Count* byte-sized elements are copied from *source* into *destination*.
	word or long-sized	word	*Count* word-sized elements are copied from word-aligned *source* into word-aligned *destination*.
long-sized	byte-sized	byte	*Count* byte-sized elements are copied from *source* into *destination*.
	word-sized	word	*Count* word-sized elements are copied from word-aligned *source* into word-aligned *destination*.
	long-sized	long	*Count* long-sized elements are copied from long-aligned *source* into long-aligned *destination*.

[1] Byte-sized types are void, char, byte, signed char, signed byte, and non-typed constant. Word-sized types are short, word, signed short, and signed word. Long-sized types are int, long, and float.

The adjustable element size feature of **memmove**() provides a deterministic optimization on the Propeller that results in fast operation.

When a constant is entered as the *source,* it will be treated as a **byte** unless it is cast to another type. Example: `memmove((word) 1244, (word) 1850, 20);` will copy 20 word-sized values starting from location 1850 into memory starting at location 1244. In other words, the contents of byte locations 1850 through 1889 are copied to byte locations 1244 through 1283. Note that setting the *destination* to (word) 1245, or the *source* to `(word) 1851` will also have the same result since the each address will be automatically word-aligned.

3.8.5.3.1. – Translation of memmove( )

The following demonstrates **memmove**() translation.

Table 29 – Memmove()

Example	Notes	Translation

char x[10];
int temp[25];
 	
memmove(temp, x, 8);
memmove((int) x, &temp[2], 2);
	Contents of character and integer arrays are copied back and forth.	VAR byte x[10]
		VAR long temp[25]
		
		bytemove(@temp, @x, 8)
		longmove(@x, @temp[2], 2)

3.9 – Conditional Statements

There are two forms of statements that affect execution conditionally, **if** and **switch**.

3.9.1 – The if..else Statement

The **if** statement tests one or more conditions and either executes or skips blocks of code accordingly. The **else** keyword works with it to clearly define alternative blocks.

3.9.1.1 – Translation of if..else

Table 30 – If..else Statements

Example	Translation
if (a = b) return 5;	if (a = b) return 5

if (x > y)
  z = x;
else
  z = y;
	if (x > y)
	  z := x
	else
	  z := y

if (temp <= 16)
  {
  xyz = temp * 2;
  return xyz;
  }
else if (temp < 32)
  xyz = temp / 2;
else
  xyz = 0;
	if (temp =< 16)
	  xyz := temp * 2
	  return xyz
	elseif (temp < 32)
	  xyz := temp / 2
	else
	  xyz := 0

3.9.2 – The switch Statement

The **switch** statement tests an expression for equality against one or more other expressions and executes the code block of the first match. The **case** keyword precedes each constant expression in which to test against and the **default** keyword is optionally used to catch all other cases. The **break** command must be included to exit the **case** block, otherwise it will continue executing statements from the cases below it.

3.9.2.1 – Translation of switch..case

Table 31 – Switch..case Statements

Example	Translation

switch (temp+1)
  {
  case 1: xyz = 2;
  break;
  case 25: action();
  default: return 0;
  }
	case (temp+1)
	  1: xyz := 2
	  25: action()
	  return 0
	  other: return 0

3.10 – Loop Statements

There are three forms of looping statements, **while**, **do**..**while**, and **for**.

3.10.1 – The while Loop

The **while** loop is a zero+ iteration mechanism; it may skip its block of statements all together if the initial condition is false, or it may execute them one or more times as long as the condition is true. The **break** command can be used to exit the loop immediately.

The **continue** command may be used to skip the remainder of the loop's current iteration and immediately start the next iteration.

3.10.1.1 – Translation of while

Table 32 – While Loops

Example	Translation

while (x < 16)
{
action();
x++;
}
	repeat while (x < 16)
	action
	x++

while (true)
{
action();
if (x++ > 5) break;
}
	repeat
	action
	if (x++ > 5)
	quit

while (x++ < 10)
{
if (x = 5) continue;
action();
y = x + 1;
}
	repeat while (x++ < 10)
	jf (x = 5)
	next
	action
	y := x + 1

3.10.2 – The do..while Loop

The **do..while** loop, in contrast to the **while** loop, is a one+ iteration mechanism; it always executes it's block of statements at least once, and may continue to execute them more times as long as the condition is true. Similar to **while**, the **break** and **continue** commands can be used to immediately exit the loop, or skip the rest of the current iteration, respectively.

3.10.2.1 – Translation of do..while

Table 33 – Do..while Loops

Example	Translation

do
{
action();
x++;
}
while (x < 16);
	repeat
	action
	x++
	while (x < 16)

3.10.3 – The for Loop

The **for** loop executes its block of statements zero or more times, depending on the condition expression. Similar to **while** and **do..while**, the **break** and **continue** commands can be used to immediately exit the loop, or skip the rest of the current iteration, respectively.

3.10.3.1 – Translation of for

Table 34 – For Loops

Example	Translation
for (x = 0; x < 16; x++)	repeat x from 0 to 15

action();	action
for (x = 20; x > 10; x--, action());	repeat x from 20 to 11 action
for (;;;) action();	repeat action

3.11 – Unconditional Branching Commands

There are three unconditional branching commands in PMC: **continue**, **break**, and **return**.

3.11.1 – continue

The **continue** command can only be used within the body of a loop (**while**, **do..while**, or **for**) to skip the rest of the current iteration and immediately start the next iteration.

3.11.1.1 – Translation of continue

See section 3.10.1.1 – Translation of while for an example.

3.11.2 – break

The **break** command can only be used in a loop (**while**, **do..while**, or **for**) or a **switch** statement to immediately exit the loop or **switch** statement.

3.11.2.1 – Translation of break

See section 3.10.1.1 – Translation of while for an example.

3.11.3 – repeat

The **return** command exits a method immediately and optionally returns a value. If **return** is given without the optional value, the value returned (by method's of types other than **void**) is that of the current local **result** variable.

3.11.3.1 – Translation of return

Table 35 – return

Example	Translation
return;	return
return 25;	return 25

3.12 – Built-in Functions

The PMC language provides much of the Propeller architecture's functionality through built-in functions. These functions closely resemble their Spin language counterparts and are available automatically without the need to "include" any header files.

3.12.1.1 – Translation of Built-in Functions

Variables are used in Table 36 to indicate the return value of the function as well as to provide a practical translation example. Note that bl is a **bool** variable, b is a **byte** variable and i is an **integer** variable.

Table 36 – Built-in Functions

Example	Notes	Translation
b = chipver();	Get Propeller version number	b := chipver
i = clkfreq();	Get current system clock freq; Hz	i := clkfreq

b = clkmode();	Get current clock mode setting	b := clkmode
clkset(*mode*, *frequency*);	Set clock mode and frequency	clkset(*mode*, *frequency*)
coginit(*cogid*, *method, stackpointer);*	Start a cog by ID for C code	coginit(*cogid*, *method*, *stackpointer)*
coginit(*cogid*, *asmmethod, mempointer);*	Start a cog by ID for Asm code	coginit(*cogid*, *@asmmethod*, *mempointer)*
b = cognew(*method*, *stackpointer*);	Start a new cog for C code	b := cognew(*method*, *stackpointer)*
b = cognew(asmmethod, *mempointer*);	Start a new cog for Asm code	b := cognew(*@asmmethod*, *mempointer)*
cogstop(*cogid*);	Stop cog by ID	cogstop(*cogid*)
bl = lockclr(*id*);	Clear semaphore; get previous state	bl := lockclr(*id*)
b = locknew;	Get new semaphore ID	b := locknew
lockret(*id*);	Return semaphore	lockret(*id*)
bl = lockset(*id*);	Set semaphore; get previous state	bl := lockset(*id*)
i = lookdown(*value*:*expressionlist);*	Get the one-based index of a value	i := lookdown(*value*:*expressionlist*)
i = lookdownz(*value*:*expressionlist);*	Get the zero-based index of a value	i := lookdownz(*value*:*expressionlist*)
i = lookup(*index*:*expressionlist);*	Get the value at a one-based index	i := lookup(*index*:*expressionlist*)
i = lookupz(*index*:*expressionlist);*	Get the value at a zero-based index	i := lookupz(*index*:*expressionlist*)
reboot;	Reset the Propeller	reboot
bl = strcomp(*straddr1*, *straddr2*);	Compare strings for equality	bl := strcomp(*straddr1*, *straddr2*)
i = strsize(*straddr*);	Get length of string	i := strsize(*straddr*)
waitcnt(*value*);	Pause cog execution temporarily	waitcnt(*value*)
waitpeq(*state*, *mask*, *port*);	Pause cog execution until I/O pins match state(s)	waitpeq(*state*, *mask*, *port*)
waitpne(*state*, *mask*, *port*);	Pause cog execution while I/O pins match state(s)	waitpne(*state*, *mask*, *port*)
waitvid(*colors*, *pixels*);	Pause cog execution until video generator is ready for data	waitvid(*colors*, *pixels*)

3.13 – Predefined Variables

A set of predefined variables exists in PMC to give read or read/write access to Propeller attributes that commonly change during runtime. These variables deliver direct access to special Propeller registers and their names exactly match their Spin language counterparts.

3.13.1.1 – Translation of Predefined Variables

An **int** variable (i) is used in Table 37 to demonstrate usage of the built-in variable.

Table 37 – Predefined Variables

Example	Notes	Translation
i = cnt;	Get current 32-bit System Counter value (read only)	i := cnt
i = ctra; --or-- ctra = i;	Get or set Counter A Control Register	i := ctra --or-- ctra := i
i = ctrb; --or-- ctrb = i;	Get or set Counter B Control Register	i := ctrb --or-- ctrb := i
dira[*pin(s)*] = i;	Set Direction Register for port A	dira[*pin(s)*] = i;
i = frqa; --or-- frqa = i;	Get or set Counter A Frequency Register	i := frqa --or-- frqa := i

i = frqb; --or-- frqb = i;	Get or set Counter B Frequency Register	i := frqb --or-- frqb := i
i = ina[*pin(s)*];	Get Input Register value for port A (read only)	i := ina[*pin(s)*]
outa[*pin(s)*] = i;	Set Output Register for port A	outa[*pin(s)*] = i
i = par;	Get cog boot parameter register (read only)	i := par
i = phsa; --or-- phsa = i;	Get or set Counter A Phase Register	i := phsa --or-- phsa := i
i = phsb; --or-- phsb = i;	Get or set Counter B Phase Register	i := phsb --or-- phsb := i
i = result; --or-- result = i;	Get or set local variable result value	i := result --or-- result := i
i = spr[*index*]; --or-- spr[*index*] = i;	Get or set Special Purpose Register	i := spr[*index*] --or-- spr[*index*] := i
i = vcfg; --or-- vcfg = i;	Get or set Video Configuration Register	i := vcfg --or-- vcfg := i
i = vscl; --or-- vscl = i;	Get or set Video Scale Register	i := vscl --or-- vscl := i

The variables **dira**, **ina**, and **outa** are I/O port registers and have special features. They are bit arrays, meaning their *pin(s)* field can indicate a single bit to address; however, *pin(s)* can also be a contiguous pin number range in the form *pinA..pinB* and the registers bits corresponding to pins A through pins B will be address.