

The μ CL Language Specification

Table of Contents

1	Introduction.....	1
2	Lexical Considerations.....	2
2.1	Character Sets and Indentation.....	2
2.2	Symbols, Constants, and Punctuation.....	4
2.2.1	Symbols.....	4
2.2.2	Constants.....	5
2.2.3	Punctuation.....	7
2.3	Comments.....	8
2.4	Continuation Lines.....	8
3	Types.....	10
3.1	Bit.....	10
3.2	Unsigned Integer.....	10
3.3	Signed Integer.....	11
3.4	Floating Point.....	11
3.5	String.....	11
3.6	Array.....	12
4	Declarations.....	13
4.1	bind Declaration.....	13
4.1.1	Simple bind Declaration.....	13
4.1.2	Array bind Declaration.....	14
4.1.3	Bit bind Declaration.....	14
4.2	code_bank Declaration.....	15
4.3	configure Declaration.....	16
4.4	constant Declaration.....	16
4.5	data_bank Declaration.....	17
4.6	debug Declaration.....	17
4.7	icd2 Declaration.....	17
4.8	icd2_configure Declaration.....	18
4.9	global Declaration.....	18
4.9.1	Simple global Declaration.....	18
4.9.2	Array global Declaration.....	19
4.10	library Declaration.....	20
4.11	library_bank Declaration.....	21
4.12	origin Declaration.....	21
4.13	package Declaration.....	22
4.13.1	pin Declaration.....	22
4.13.2	package Example.....	23
4.14	processor Declaration.....	23
4.14.1	code_bank Declaration.....	24
4.14.2	configure_address Declaration.....	24
4.14.3	configure_fill Declaration.....	24
4.14.4	configure_option Declaration.....	25
4.14.5	data_bank Declaration.....	25
4.14.6	global_region Declaration.....	25
4.14.7	icd2_global_region Declaration.....	26

Table of Contents

4	Declarations	
4.14.8	icd2_shared_region Declaration	26
4.14.9	interrupts_possible Declaration	26
4.14.10	osccal_at_address Declaration	27
4.14.11	osccal_in_w Declaration	27
4.14.12	osccal_register_symbol Declaration	27
4.14.13	packages Declaration	27
4.14.14	pin Declaration	28
4.14.15	shared_region Declaration	29
4.14.16	processor Example	29
4.15	register Declaration	31
4.16	register_array Declaration	31
4.17	string Declaration	32
4.18	ucl Declaration	32
5	procedure Declaration	34
5.1	argument Declaration	34
5.2	arguments_none Declaration	35
5.3	exact_delay Declartion	35
5.4	local Declaration	35
5.4.1	Simple local Declaration	35
5.4.2	Array local Declaration	36
5.5	returns Statement	36
5.6	returns_nothing Declaration	36
6	Statements	37
6.1	assemble Statement	37
6.2	Assignment Statement	39
6.2.1	Simple Assignment Statement	39
6.2.2	Multiple Assignment Statement	39
6.3	call Statement	40
6.4	delay Statement	41
6.5	delay_set Statement	41
6.6	do_nothing Statement	42
6.7	if Statement	42
6.8	loop_exactly Statement	43
6.9	loop_forever Statement	43
6.10	return Statement	43
6.11	switch Statement	44
6.12	watch_dog_reset Statement	45
6.13	while Statement	45
7	Expressions	46
7.1	Differences from C Expressions	46
7.2	What is Precedence?	46
7.3	Assignment Operator (:=)	47
7.4	Comma Operator (,)	47
7.5	Conditional OR ()	48

Table of Contents

7 Expressions

7.6	Conditional AND (&&)	48
7.7	Relational Operators and Bit Selection (<, =, >, <=, !=, >=, @)	49
7.8	Addition and Subtraction (+, -)	50
7.9	Multiplication, Division, and Modulo (*, /, and %)	50
7.10	Bitwise OR ()	51
7.11	Bitwise AND (&)	52
7.12	Bitwise XOR (^)	52
7.13	Shift Operators (>>, <<)	53
7.14	Unary Operators (-, !, ~)	53
7.15	Array Operator (...[...])	54
7.16	Dot Operator (L.size)	54
7.17	Procedure Invocation (P(...))	54
7.18	Debugger Support	55

1 Introduction

µCL stand for MicroController Language. It has the following basic goals:

Targeted for Microcontrollers

Microcontrollers are an interesting breed of system that differ pretty substantially from their more general purpose cousins. In general, microcontrollers are tending towards a so called Harvard architecture where the program code lives in non-volatile flash read only memory and the data lives in volatile random access memory. In addition, microcontrollers tend to have individual pins that can be programmed for specific uses.

Microcontroller Neutral

The language specification is not geared towards any specific microcontroller. While the first µCL code generators are geared towards the popular PIC[®] microcontrollers from MicroChip[®], other code generators for microcontroller families from other vendors.

Platform Neutral

The µCL compiler runs on multiple platforms -- Linux[®], Windows[®], etc.

Beginner Friendly

Every attempt has been made to make the µCL friendly to beginning users. In particular, µCL does *not* use braces ({...}) to nest statements; instead it uses indentation. The reason for this is to eliminate mismatched brace errors, a particularly nasty error that frequently results in an error message that points to a location that is quite far removed from the actual error.

This language specification tends to follow the same overall structure as other language specifications; namely it discusses lexical issues first, followed by global declarations, procedure declarations, statements, and expressions.

2 Lexical Considerations

The term "lexical" is a computer science term that roughly corresponds to the character, word, and punctuation rules for a program.

2.1 Character Sets and Indentation

Early programs were done on punched cards using the Hollerith character codes. The unlamented punched cards were eventually replaced with 7-bit ASCII (American Standard Code for Information Interchange.) μ CL programs use an 8-bit character code called Latin-9 which has 7-bit ASCII embedded in its first half and some additional character codes suitable for European languages in its second half. The official name for Latin-9 is ISO-8859-15, where ISO stands for International Standards Organization, 8859 is the number reserved for 8-bit character code standards, and 15 is the standard number used for Latin-9. What this means to the average μ CL user is that symbols and strings can contain characters with accents, tildes, and umlauts and there are some additional non-letter symbols such as the Euro currency symbol ('€' = code 164 decimal) are available for inclusion in string and character constants.

A μ CL program is broken into a sequence of lines, where each line is terminated by the LF character (Line-Feed = code 10 decimal.) There are many computer systems that insert an extra CR character (Carriage-Return = code 13 decimal) before each LF character. In μ CL, the CR character before each LF character is silently ignored. The CR and CR-LF sequence is called the end of line. The average μ CL user does not need to worry about any of this, whenever you need to start a new line, just press the [Enter] key, and the a new line will be started; the underlying editor will deal with the CR and LF stuff appropriately.

In general, each μ CL declaration and statement occupies one line. Line comments start with a hash character ('#' = code 35 decimal) and continue to the end of line. A fragment of a μ CL program is shown below:

```
ucl 1.0                                # Line 1
                                         # Line 2 (blank)
# Copyright © 2004 by Gramlich          # Line 3 (comment)
                                         # Line 4 (blank)
library PIC16F876A                     # Line 5
```

Note that this μ CL program fragment has the Latin-9 character for copyright ('©' = code 169 decimal.) on the third line.

In addition to treating end of line as a statement and declaration termination, μ CL uses indentation to specify nested statements and declarations. The Python programming language has popularized this concept. Some example code with indentation is shown below:

```
# All procedure declarations start at indentation
# level 0 (i.e. no preceeding white space.)

procedure maximum
    # Indentation level 1. All procedure arguments
    # and statements start at this level.

    argument a byte
    argument b byte
    returns byte

    if a > b
        # Indentation level 2. All "then" clause
```

The μ CL Language Specification

```
# statements occur at one indentation level
# greater than parent "if" statement.

return a

# The end of "then" clause reverts to
# previous indentation level.

# Back to indentation level 1.

return b

# End of arguments and statements reverts to
# previous indentation level:

# Back to indentation level 0.
# More declarations follow here with no indenation.

global variable x
global variable y
```

This example shows an μ CL procedure for computing the maximum of two values. The initial declaration line is not indented at all and specifies the procedure name -- `maximum`. All subsequent procedure declarations (e.g. `argument` and `returns`) and procedure statements (e.g. `if` and `return`) must be indented by at least one level. The `if` statement starts at indentation level 1 and has its "then" clause at indentation level 2. At the end of the "then" clause, indentation returns level 1. Similarly, the end of the procedure is indicated by a return to indentation level 0.

The details for figuring out indentation are based on the column number for the first printing character (i.e. not a space or tab) on a line. If a line contains no printing characters, it is treated as a blank line and is ignored by the μ CL language for indentation level determination. All lines in a sequence that are at the same column number (ignoring blank lines) are considered to be at the same indentation level.

Determining the column number of a line is complicated by the TAB character (= code 8 decimal.) For μ CL, tab stops occur every 8 spaces. The occurrence of a tab advances the column number to the next multiple of 8 plus 1. The beginning of the line is at column 1. A tab at the beginning of a line advances to column 9 (= 8 + 1.) Another tab advances to column 17 (= 2 \times 8 + 1.) A tab at column 13 also advances to column 17. For the average μ CL user, if the code looks properly indented on the screen, the compiler will "see" the same indentation and behave accordingly.

Each time the column number for a line is greater than the previous printing line, the indentation level is incremented by one. When the column number for a line is less than the previous printing line, the indentation level reverts back to the last indentation level at that same column number. Some examples of different indentation levels are shown below:

```
# Column labels:
#   A   B   C   D
#
# ...
# Indentation level N:
if a < b
  # Indentation level N + 1:
  if c < d
    # Indentation level N + 2:
    e := f
  else # Indentation level N + 1
    # Indentation level N + 2 again:
    # Note that it does not line up
```

The μ CL Language Specification

```
# with previous N + 2 level.
# This ugly but legal.
g := h

# Indentation level N again:
i := j
```

The first indentation level occurs for lines in column A. The second indentation level occurs for lines in column B. The third indentation level occurs in column C for the first occurrence and in level D in the second occurrence. While this legal μ CL code, it is strongly urged that you be consistent with indentation levels. Having each indentation level vary by 4 columns is the recommended style.

The C, C++, and Java programming languages use braces (`{` = code 123 decimal and `}` = code 125 decimal) to indicate statement and declaration nesting. For programmers who type in braces without even thinking, the μ CL language permits braces and semi-colons to be inserted into valid μ CL programs. The μ CL language ignores these characters except for computing column positions for indentation levels. Thus, the following code fragment is also legal μ CL program.

```
if (a < b) {
    return a;
} else {
    return b;
}
```

μ CL does not attempt to verify that the braces match. No further examples in this language use either braces or semicolons.

2.2 Symbols, Constants, and Punctuation

In μ CL, a program is first broken into a sequence of tokens. Each token is one of the following categories:

Symbol

Symbols are used for variable names, procedure names, statement keywords, declaration keywords, assembler opcodes, etc.

Literal

The word literal is the computer science word for a constant. There are number literals (decimal, octal, hexadecimal, and floating point), string literals, and character literals.

Punctuation

There are binary operators (e.g. `+`, `-`, `*`, `/`, etc.), unary operators (e.g. `!`, `~`, `-`, `+`), and grouping operators (e.g. `(`, `)`, `[`, `]`).

Miscellaneous

The two miscellaneous tokens are end of line and line comment.

2.2.1 Symbols

In μ CL, a symbol is a sequence of letters (`'A'-'Z'`, `'a'-'z'`), digits (`'0'-'9'`), dollar sign (`'$'`), and underscore (`'_'`). The following restrictions are in place:

- The first character of a symbol can be a letter or dollar sign. No underscores or digits are allowed at the beginning.
- The middle characters of symbol can be letters, digits, and underscores. No dollar signs are allowed.
- The last character of a symbol can be a letter or digit. No underscores or dollar signs are allowed.

The µCL Language Specification

- Two underscores may not occur next to one another.

As a point of clarification, a letter includes the standard ASCII letters in addition to the Latin-9 letters with embedded punctuation.

Some examples of good symbols are listed below:

```
a
Z
$c
bunny
StartHere
start_here
test1
```

Some examples of bad symbols are listed below:

```
_Special_      # Starts (and ends) with underscore
Micro$oft      # Dollar sign in middle
Bad__news      # Two underscores in a row
1potato        # Starts with digit
```

By convention, system libraries will tend to define symbols that start with a dollar sign ('\$'). This allows µCL programmers to freely pick names without having to worry about accidentally conflicting with a system library symbol.

2.2.2 Constants

µCL supports decimal, hexadecimal, and floating point numbers, character constants, and string constants.

2.2.2.1 Decimal Numbers

A decimal number consists of one or more decimal digits ('0' through '9') inclusive. Some example decimal numbers are shown below:

```
0
1
9
12
1234567
```

2.2.2.2 Hexadecimal Numbers

A hexadecimal number starts with a "0x" or "0X" prefix. The prefix is followed by one or more hexadecimal digits ('0'-'9', 'A'-'F', and 'a'-'f'.) Some example hexadecimal numbers are shown below:

```
0x0
0x1
0x9
0xa
0xA
0xf
0xF
0xf12
0xABE
```

2.2.2.3 Floating Point Numbers

{Floating point *is* coming, but it has not been implemented yet. Thus, nothing in this section, 2.2.2.3, acutally works yet.}

A floating point number constant is a decimal number with a decimal point ('.' = 46 code decimal) character in it optionally followed by an exponent in "e" notation. "E" notation is the letter 'e' or 'E' followed by an optional sign, '+' or '-', followed by a decimal number.

```
0.
.0
0.0
1.
1.2
1.2345
123.45
.1e-6          -6  # .1×10
1.23E-12       -12# 1.23×10
9.87E8         8   # 9.8×10
```

Please note that 1e9 is not a valid floating point constant because there is no decimal point in the first number (i.e. the mantissa.)

2.2.2.4 String Constants

String constants are enclosed in double quotes ("" = code 34 decimal.) Only printing characters and spaces are allowed between the quote characters. A pair of backslash characters ('\ = code 92 decimal) embedded in the string is used to delineate the non-printing characters. The non-printing characters can be expressed as decimal numbers, hexadecimal numbers, or symbolic names. Multiple non-printing characters have their number separated by a comma. The symbolic names come from the table below:

Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value
nul	0	ht	9	dc2	18	esc	27	t	9
soh	1	lf	10	dc3	19	fs	28	n	10
stx	2	vt	11	dc4	20	gs	29	v	11
etx	3	ff	12	nak	21	rs	30	f	12
eot	4	cr	13	syn	22	us	31	r	13
enq	5	so	14	etb	23	sp	32	tab	8
ack	6	si	15	can	24	del	127	bsl	92
bel	7	dle	16	em	25	a	7	dq	34
bs	8	dc1	17	sub	26	b	8	sq	39

The symbols in the table come mostly from the ASCII control code symbols. The single letter codes are from the escape codes used in C and C++. The last three codes are used to encode backslash (bsl = '\'), single quote (sq = '"') and double quote (dq = '"'). Some string constant examples are shown below:

```
" "          # Empty string
"a"          # String containing single letter "a"
" "          # String containing single space
"Hello, World!\n" # "Hello, World" followed by line feed
"No.\tab\Desc.\lf\" # String with tab and line feed in it
"\bsl\"      # String containing single backslash
```

2.2.2 Constants

The µCL Language Specification

```
"'"          # String containing single quote.
"\sq\"      # String containing single quote
"\dq\"      # String containing double quote
"\dq\Hi\dq\" # String encloses "Hi" in double quotes
"Done!\10,13\" # String followed by CR and LF
"Done!\r,n\"  # String followed by CR and LF
"Done!\cr,lf\" # String followed by CR and LF
```

2.2.2.5 Character Constants

Character constants are just like string constants except that they are enclosed in single quotes and they must contain a single character. Some character constant examples are shown below:

```
'a'          # The letter 'a'
' '          # A space
'\tab\'      # A tab
'©'          # A Latin-9 copyright '©'
```

2.2.3 Punctuation

The following tokens are used in µCL as binary operators:

Symbol	Usage	Symbol	Usage	Symbol	Usage
+	Addition	<	Less Than	<<	Shift Left
−	Subtraction	<=	Less Than or Equals	>>	Shift Right
*	Multiplication	=	Equals	~	Concatenate
/	Division	!=	Not Equals	&&	Conditional AND
%	Remainder	>	Greater Than		Conditional OR
&	Bitwise AND	>=	Greater Than or Equals	,	Expression Separate
	Bitwise OR	@	Bit Selection		
^	Bitwise XOR	:=	Assignment		

The following tokens are used as unary operators:

Symbol	Usage
+	Positive
−	Negative
~	Bitwise NOT
!	Logical NOT

The remaining tokens are:

Symbol	Usage	Symbol	Usage	Symbol	Usage
(Open Parenthesis)	Close Parenthesis	?	Arithmetic If
[Open Bracket]	Close Bracket	:	Arithmetic If

2.3 Comments

A comment line starts with a sharp (# = code 35 decimal) character and goes until the end of line. All characters from the sharp character to the end of line are ignored by the μ CL language. A comment can be placed at the end of any line. Some examples of comments are listed below:

```
# This is a one line comment followed by an empty line

# This a sequence of comments that spans a total of
# three lines. Each line is technically treated
# as an independent comment by the compiler.

global x float24    # A comment after a declaration.
```

2.4 Continuation Lines

Sometimes an expression gets too long to conveniently fit on one line in the editor. Such an expression can be broken into multiple continuation lines using the following rules:

- the last token on each continuation line (except the last one) must a punctuation token other than close parenthesis (') or close bracket (]), and
- each continuation line after the first one, must be indented by the same amount and it must be indented more than the first continuation line.

Some examples should help clarify things.

Example 1:

```
if a * b > c && d + e < f ||      # Cont. line 1
    g != 0 && h / g > 0          # Cont. line 2
    a := a + 1
```

In this example, the if statement expression is broken across two lines. The first line ends in '||' which tells the compiler to look at the next line for the rest of the expression.

Example 2:

```
call long_procedure_name(      # Cont. line 1
    expression1,               # Cont. line 2
    expression2)               # Cont. line 3
```

In this example, the call statement is broken across three lines. The open parenthesis '(' after 'long_procedure_name' indicates that the first line is continued onto the second. The second line ends in a comma (',') which indicates that the second line continues to the third line. The third line ends in a close parenthesis (') which is one of the two punctuation characters that does not force a continuation line.

Here are some examples of bad continuation lines:

Example 3:

```
call long_procedure_name(
    expression1, expression2)
```

The μ CL Language Specification

In this example, the first line ends in an open parenthesis ('('), but the second line is not indented by more than the first line.

Example 4:

```
call long_procedure_name(  
    expression1,  
    expression2)
```

In this example, second and third lines are not indented by the same amount.

3 Types

The μ CL language supports the following basic types:

Bit

A single bit that contains 1 or 0. In other programming languages this is called Boolean or Logical. An I/O pin is an important special case of the Bit type.

Unsigned Integer

An unsigned integer is one whose lowest legal value is 0 and its highest value depends up on the number of bits available. Currently, μ CL only supports 8 bit unsigned integers.

Signed Integer (unimplemented)

A signed integer is one that can represent positive numbers and zero. The largest legal value depends upon the number of bits available.

Floating Point Number (unimplemented)

For μ CL, a floating point number is one that can represent a number between approximately $\pm 10^{38}$.

String

In μ CL, a string is a read only sequence of characters.

In addition, μ CL supports linear arrays of these types.

3.1 Bit

A bit can contain the values '0' and '1'. Bits can be stored in variables and transferred via assignment statements.

Some examples of bit code are shown below:

```
local a bit
local b bit
local c bit

a := 1
b := c
c := a && c          # a AND c
a := !b              # NOT b
```

3.2 Unsigned Integer

The unsigned integer types are listed in the table below:

Name	Precision	Lowest Value	Highest Value	Status
byte	8 Bits	0	255	Implemented
unsigned8	8 Bits	0	255	Unimplemented
unsigned16	16 Bits	0	65535	Unimplemented
unsigned24	24 Bits	0	16777215	Unimplemented
unsigned32	32 Bits	0	4294967295	Unimplemented

The µCL Language Specification

The types `byte` and `unsigned8` can be used interchangeably. On an 8-bit microcontroller, arithmetic using the higher precisions takes more cycles to compute. Currently only `byte` is implemented.

Some example code with bytes is shown below:

```
variable a byte
variable b byte
variable c byte

a := 23
b := a
c := a + b << 3
```

3.3 Signed Integer

The signed integer types are listed in the table below:

Name	Precision	Lowest Value	Highest Value	Status
<code>signed8</code>	8 Bits	-128	+127	Unimplemented
<code>signed16</code>	16 Bits	-32769	+32768	Unimplemented
<code>signed24</code>	24 Bits	-8388608	+8388607	Unimplemented
<code>signed32</code>	32 Bits	-2147483648	+2147483647	Unimplemented

None of these types are currently implemented.

3.4 Floating Point

The floating point type is listed in the table below:

Name	Precision	Largest Number	Smallest Number	Digits of Accuracy	Status
<code>float32</code>	32 Bits	$\pm 10^{38}$	$\pm 10^{-38}$	~7	Unimplemented

Notice that `float24` and `float32` can represent the same sized numbers; however, the number of digits of accuracy varies. Lastly, the floating point arithmetic operations are significantly slower on the 8-bit microcontrollers. The libraries that implement the operations take up a substantial amount of code space.

3.5 String

In µCL, a string is a read only string that is stored in program memory; strings can not be modified.

```
local text string
local chr byte
local length byte

text := "A string literal" # String assignment
chr := text[3]             # chr = 't'
length := text.size        # length = 16
```

3.6 Array

Any type can be put into an array. Arrays have a fixed size. The array can be indexed into with a byte (i.e. unsigned8) index value. There is no bounds checking when accessing a byte array. The code below shows some examples array types.

Some sample code with byte arrays is shown below:

```
global buffer[10] array[byte]      # Ten byte buffer
global index byte
global size byte

index := buffer.size - 4
buffer[0] := 0
buffer[index] := 8
buffer[index + 1] := buffer[index]
buffer[23] := 17                  # Out of bounds!
size := buffer.size
```

{more here}

4 Declarations

All declarations occur at the outer most level in a program. With the exception of the `procedure` declaration, all declarations are listed in alphabetical order in the following sections.

4.1 `bind` Declaration

The `bind` declaration has three different forms:

Simple `bind` Declaration

Defines a new variable that is equivalent to a previously defined global variable or register.

Array `bind` Declaration

Defines a new variable that is equivalent to a specific entry in a global array.

Bit `bind` Declaration

Defines a new bit variable that is equivalent to a specific bit in a previously defined global variable or register.

All three forms define a new variable that is in some way equivalent to a previously defined variable.

4.1.1 Simple `bind` Declaration

A simple `bind` declaration has the following form:

```
bind new_name = old_name
```

where

new_name

is the new symbol name, and

old_name

is a previously defined register or global variable.

After a simple `bind` declaration, *new_name* is a variable that is equivalent to *old_name*.

For example, for the following declarations:

```
global a byte
register b = 0x88
bind aa = a
bind bb = b
```

the following are equivalent:

Original	Equivalent
<code>a := 17</code>	<code>aa := 17</code>
<code>b := 17</code>	<code>bb := 17</code>
<code>c := a + 1</code>	<code>c := aa + 1</code>
<code>c := b - 1</code>	<code>c := bb - 1</code>

where *c* is a simple global byte variable (e.g. `global c byte`.)

4.1.2 Array bind Declaration

An array `bind` declaration has the following form:

```
bind new_name = array_name [ constant_expression ]
```

where

new_name

is the new symbol name,

array_name

is a previously defined `register_array` or `global array`, and

constant_expression

is a constant expression.

After an array `bind` declaration, the new variable is equivalent to the specified array location.

For example:

```
global state[12] array[byte]
bind command = state[0]
bind temporary = state[1]
...
bind last = state[11]
```

defines a bunch of variables that live in an array of bytes. The following statements are equivalent:

Original	Equivalent
<code>state[0] := 17</code>	<code>command := 17</code>
<code>state[1] := 17</code>	<code>temporary := 17</code>
<code>state[11] := 17</code>	<code>last := 17</code>
<code>c := state[0] + 1</code>	<code>c := command + 1</code>
<code>c := state[1] - 1</code>	<code>c := temporary - 1</code>
<code>c := state[11] & 1</code>	<code>c := last & 1</code>

where *c* is a simple global byte variable (e.g. `global c byte`.)

4.1.3 Bit bind Declaration

The bit `bind` declaration has the following form:

```
bind new_name = old_name @ constant_expression
```

where

new_name

is the new symbol name,

old_name

4.1.1 Simple bind Declaration

The μ CL Language Specification

is a previously defined register or global variable, and
constant_expression
is a constant expression that evaluates to a value between 0 and 7.

A new variable called *new_name* is defined that is equivalent to the specified bit expression.

For example,

```
register $intcon 0xb
bind $gie $intcon @ 7
bind $eeie $intcon @ 6
bind $t0ie $intcon @ 5
bind $inte $intcon @ 4
bind $rbie $intcon @ 3
bind $t0if $intcon @ 2
bind $intf $intcon @ 1
bind $rbif $intcon @ 0
```

defines the various interrupt enable bits for the `$intcon` register for the PIC16F84. For example the following statements are equivalent

Original	Equivalent
<code>\$intcon@7 := 1</code>	<code>\$gie := 1</code>
<code>\$intcon@4 := 0</code>	<code>\$inte := 0</code>
<code>\$intcon@1 := 0</code>	<code>\$intf := 0</code>

4.2 code_bank Declaration

The `code_bank` declaration has the follow form:

```
code_bank constant_expression
```

where

constant_expression
is an constant expression that evaluates to a valid code bank (typically between 0 and 3.)

This declaration specifies that until further notice, all `procedure` declarations are to be located in the specified code bank.

For example,

```
code_bank 0
procedure interrupt
...

code_bank 3
procedure main
...
```

causes the procedure named `interrupt` to be placed into code bank 0 and the procedure named `main` to be placed into code bank 3.

4.3 `configure` Declaration

The form of the `configure` declaration is as follows:

```
configure option_name = option_value , ...
```

where

option_name

is a valid configuration word option name,

option_value

is a valid value for the specified configuration option.

This declaration is used to specify the various configuration options required to set up the microcontroller configuration word (or words.)

For example, the configure declarations below set up the configuration options for a PIC16F767:

```
configure cp=off, cpmx=rcl, debug=off, borv=borv00, boren=off
configure mclre=off, pwrt=off, wdte=off, fosc=hs
configure borsen=off, ieso=off, fcmen=off
```

Note that configuration options can be changed. Thus, the initial default configuration options can be defined in a library file and the main program can subsequently override them. The options in effect at the end the program are the ones that are actually output to the final output file.

The valid configuration options are listed under the `processor` declaration. Thus, all `configure` declarations must occur after the `processor` declaration.

4.4 `constant` Declaration

The constant declaration has the following form:

```
constant new_name = constant_expression
```

where

new_name

is the name of the constant, and

constant_expression

is an expression that evaluates to a constant value.

Some example constant declarations are shown below:

```
constant crystal_speed = 20000000
constant instructions_per_second = crystal_speed / 4
constant prefix = "Hello "
constant george = prefix ~ "George"
```

```
constant alice = prefix ~ "Alice"  
constant bottom = -$u2i(8)  
constant pi = 3.14151926  
constant pi2 = pi / 2.0
```

note that the constants can be either numerical or string (i.e. string concatenation.)

The constant declarations are evaluated in sequence. No forward references to constants that are defined later on are permitted.

4.5 data_bank Declaration

The `data_bank` declarations has the following form:

bank constant_expression

where

constant_expression

is the register bank to select.

Some microcontrollers have more than one bank of memory (e.g. the PIC1687x). The `data_bank` declaration is used to select which memory bank to allocate subsequent variables from.

The programmer is responsible for ensuring that the memory within a given bank are not exhausted. The compiler will generate an error message if the number of registers within a given bank are exhausted.

The compiler is responsible for generating the additional data bank select instructions for accessing variables that located in various different memory banks.

4.6 debug Declaration

The `debug` declaration has the following form:

debug procedure_name , ...

where

procedure_name

is the name of a procedure.

The `declaration` declares the procedures for which additional debugging code will be inserted. Each procedure listed in a `debug` declaration can support break points when being used by the µCL integrated programming environment.

4.7 icd2 Declaration

The form of the `icd2` declaration is as follows:

icd2

This declaration tells the compiler to use the various configurations needed for Microchip In Circuit Debugger 2. Obviously, this declaration is extremely specific to the microchip product line.

4.8 `icd2_configure` Declaration

The form of the `icd2_configure` declaration is as follows:

```
icd2_configure option_name = option_value , ...
```

where

option_name

is a valid configuration word option name,

option_value

is a valid value for the specified configuration option.

This declaration is used to specify the various configuration options required to set up the microcontroller configuration word (or words) for the Microchip ICD2. This declaration is very specific to the Microchip product line.

For example, the ICD2 configure declarations below set up the configuration options for a PIC16F767:

```
icd2_configure cp=off, cpmx=rcl, debug=on, borv=borv00, boren=off
icd2_configure mclre=off, pwrt=off, wdte=off, fosc=hs
icd2_configure borsen=off, ieso=off, fcmen=off
```

{Note: Fix compiler to use this declaration.}

4.9 `global` Declaration

There are two related forms for the `global` declaration:

Simple `global` Declaration

This declares a simple global variable

Array `global` Declaration

This declares a global array.

The defined global variable is accessible from all procedures.

4.9.1 Simple `global` Declaration

A simple `global` declaration has the following form:

```
global variable_name variable_type
```

where

variable_name

is the name of the new global variable, and

variable_type

is the type of the new global variable.

For example,

```
global trace bit
global command byte

...

procedure main
  ...

  if trace
    call $uart_byte_print(command)
    trace := 0
```

shows the definition of two global variables, `trace` and `command`, where the first is of type `bit` and the second is of type `byte`. Later on in the procedure `main`, both `trace` and `command` are accessed.

4.9.2 Array global Declaration

The array global declaration has the following form:

```
global array_name [ size_expression ] array [ simple_type ]
```

where

array_name

is the variable name for the global array,

size_expression

is an expression that evaluates to a constant that specifies the array size, and

simple_type

is the type of each element in the global array. Currently, only `byte` is allowed.

The defined global variable is an array that is accessible from all procedures.

For example,

```
global buffer[16] array[byte]
global buffer_in byte

...

procedure main
  ...

  variable datum byte

  ...

  buffer_in := buffer_in & 0xf
  buffer[buffer_in] := datum
```

shows the definition of the global variable `buffer` which is an array of 16 bytes. Later on in the `main` procedure, `datum` is stored into `buffer`.

4.10 library Declaration

The library declaration has the following form:

```
library library_name
```

where

library_name

is the name of the library to use with the program.

All of the declarations from the library file are included into the main program.

A library named `my_library` will have a file name of `my_library.ucl`. A library file can reside in either the µCL system directory/folder, or in the directory/folder that contains the main program. If a file with the same name resides in both directories/folders, the compiler will complain about the ambiguity and refuse to load either one. { Actually, I've been burned by this several times, and I need to add the test! }

When it comes to `constant` declarations that reside in an µCL library, care must be taken to ensure that there are no forward references. Thus, if `constant1` is defined in `library1` and `constant2` is defined using `constant1` in `library2`, `library1` must be referenced before `library2`.

```
# In library1.ucl:
...
constant constant1 = 1
...

# In library2.ucl:
...
constant constant2 = constant1 * 3
...

# In main.ucl:
...
library library1    # Defines constant1
library library2    # Defines constant2
```

By convention, library names that start with a dollar sign ('\$') are system libraries and library names that do not start with a '\$' are user supplied libraries. By convention, system libraries should only define symbols with a dollar sign in them in order to avoid conflicts with user supplied symbols.

Some example code using the library declarations is shown below:

```
library $pic16f767  # System definitions for PIC16F767
library navigate    # User navigation package
```

The library `$pic16f767` contains a whole slew of `constant` and `bind` declarations that declare symbols that start with a '\$'. The `librarynavigate` is a user library where symbols start with the letters 'a' through 'z'.

Libraries can recursively reference other libraries. Thus, the library `$pic16f767` loads the library `$pic16f7x7`.

The compiler will only load a library once. If a library is referenced a second time, the compiler silently ignores the second load request. For those of you who know what the word idempotent means, μ CL libraries are idempotent.

μ CL libraries can contain `procedure` declarations. The compiler delays producing code for these loaded procedures until after it has produced code for the main procedures. The `library_bank` declaration allow you to load libraries into different code banks.

4.11 `library_bank` Declaration

The `library_bank` declaration has the following form:

```
library_bank bank_expression
```

where

bank_expression

is an expression that evaluates to a constant, typically between 0 and 3 inclusive.

This declaration specifies that any procedures loaded from a library will be loaded into the *bank_expression* code bank.

For example,

```
...
library_bank 2
library $float32
library_bank 3
library $trig
```

4.12 `origin` Declaration

The `origin` declaration has the following form:

```
origin constant_expression
```

where

constant_expression

is the a constant expression that specifies where subsequently generated code is to be placed.

The `origin` declaration is used to retarget code placement.

For microcontrollers that have the concept of code banks (e.g. the PIC1687x), the origin declaration uses the high order bits of the origin to implicitly set the code bank. It is up to the programmer to manage the placement of procedures within code banks using the origin declaration. The compiler will generate an error if the procedures within a given code bank spill over into another code bank.

The compiler is responsible for generating the appropriate additional instructions for placing calls from procedures in one code bank to procedures in a different code bank.

4.13 package Declaration

The `package` declaration has the following form:

```
package name
=> pin_declarations
...
<=
```

where

name
is the name of the desired package, and
pin_declarations
is an indented list of `pin` declarations.

The `package` declaration selects which integrated circuit package is being used for a particular microcontroller project. The subsequent nested `pin` declarations further identify how the various pins on the microcontroller are being used. Using this information, the compiler generates the appropriate initialization code for specified pin usages.

Many microcontrollers are supplied more than one package (e.g. DIP, SOIC, SSOP, etc.) For example, the PIC16F87 comes in an 18-pin dual in-line package (DIP), an 18-pin small outline integrated circuit (SOIC), a 20-pin small shrink outline package (SSOP), and a 28-pin quad flat no leads package (QFN). When you select a particular package for your project, the `package` declaration helps capture the relevant information about how the package is being used.

4.13.1 pin Declaration

The `pin` declarations has the following form:

```
pin pin_number = pin_usage , new_name = usage , ...
```

where

pin_number
is the decimal pin number being selected,
pin_usage
is the pin usage,
name
is a new variable or constant, and
usage
is one of `name`, `bit`, or `mask`.

Everything up to the first comma is required; everything from the comma onwards is optional. The *pin_number* and *pin_usage* must match the corresponding information in the `processor` declaration. If they do not

match, the compiler generates a fatal error.

The table below summarizes what happens for name, bit, and mask binding:

pin Syntax	Is Equivalent To
<i>new_name = name</i>	<i>bind new_name = port_name @ bit_number</i>
<i>new_name = bit</i>	<i>constant new_name = bit_number</i>
<i>new_name = mask</i>	<i>constant new_name = 1 << bit_number</i>

4.13.2 package Example

The example below shows an example of a package declaration:

```
library $pic16f676

package pdip
  pin 1 = power_supply
  pin 2 = clk_in
  pin 3 = an3
  pin 4 = mclr
  pin 5 = rc5_out, name=coil0a, mask=coil0a_mask
  pin 6 = rc4_out, name=coil0b, mask=coil0b_mask
  pin 7 = rc3_out, name=coil1a, mask=coil1a_mask
  pin 8 = rc2_out, name=coil1b, mask=coil1b_mask
  pin 9 = rc1_in, name=step
  pin 10 = rc0_in, name=dir
  pin 11 = ra2_in, name=mode
  pin 12 = ra1_in
  pin 13 = an0
  pin 14 = ground
```

In this example, the microcontroller being used is the PIC16F676. The processor declaration is in the `$pic16f676` library. The 14-pin plastic DIP (pdip) package is being used for this particular project. Pin 1 and 14 are used for the power supply and ground respectively. Pin 2 is used as an oscillator input and pin 4 is used for master clear. Pins 5 through 8 are used as digital outputs on port C bits 2 through 5. Pins 9 through 12 are digital inputs on port C bits 0 and 1 and port A bits 1 and 2. Pin 3 and 13 are analog inputs for A/D selects of 0 and 3. All of the digital I/O pins are given names (i.e. `coil0a`, `coil0b`, `coil1a`, `coil1b`, `step`, `dir`, and `mode`.) The digital output pins also have bit masks defined (i.e. `coil0a_mask`, `coil0b_mask`, `coil1a_mask`, `coil1b_mask`.) As you can see, there is quite a bit of documentation about the project embedded into the package declaration.

4.14 processor Declaration

The package declaration has the following form:

```
processor name
=> processor_declarations
...
<=
```

where

name

is the processor name, and

processor_declarations

is an indented list of processor declarations.

The *processor_declarations* are listed in alphabetical order in the sections that follow.

4.14.1 **code_bank Declaration**

The `code_bank` declaration has the following format:

```
code_bank start_address : start_address
```

where

start_expression

is the address of the first instruction location in the code bank, and

end_expression

is the address of the last instruction location in the code bank.

This declaration specifies the starting address and ending address of one code bank for the processor. For the PIC16 series of processors, most code banks are 2K words in size.

4.14.2 **configure_address Declaration**

The `configure_address` declaration has the following format:

```
configure_address address_expression
```

where

address_expression

is a constant expression that specifies the address in the Intel Hex file at which the configuration word is located.

The address specified by this declaration is used by all following `configure_fill` and `configure_option` declarations.

A `configure_address` declaration can occur more than once. Each `configure_address` declaration remains in effect until superceded by the next `configure_address` declaration. Multiple `configure_address` declarations are only needed for microcontrollers that have more than one configuration word (e.g. the PIC16F7x7 series.)

4.14.3 **configure_fill Declaration**

The `configure_fill` declaration has the following format:

```
configure_fill fill_mask_expression
```

where

fill_mask_expression

is a constant expression of a value that is always OR'ed into the current configuration word.

This declaration is used to set one or bits of the current configuration word to a 1.

4.14.4 **configure_option Declaration**

The `configure_option` declaration has the following format:

`configure_option field_name : option_name = option_value`

where

field_name

is the name of the configuration word field that is being specified,

selection_name

is the name of the configuration selection, and

option_value

is a constant expression that specifies what bits to OR into the configuration word if selected.

Each `configure_option` specifies a possible value for a configuration word field. The `configure` declaration is used to set up the configuration word (or words) for project.

4.14.5 **data_bank Declaration**

The `data_bank` declaration has the following format:

`data_bank start_address : end_address`

where

start_address

is the address of the first memory address in the data bank, and

end_address

is the address of the last memory address in the data bank.

This declaration specifies the first and last addresses of a processor data bank. For the PIC16 series of microcontrollers, data banks are typically 128 bytes in size.

4.14.6 **global_region Declaration**

The `global_region` declaration has the following format:

`global_region start_address : end_address`

where

start_address

4.14.3 **configure_fill Declaration**

is the address of the first memory address of a global memory region, and
end_address
is the address of the last memory address of the same global memory region.

The compiler uses a global region to allocate variables from (i.e. global variables, procedure arguments, local variables, temporary variables, etc.)

4.14.7 *icd2_global_region* Declaration

The *icd2_global_region* declaration has the following format:

```
icd2_global_region start_address : end_address
```

where

start_address
is the address of the first memory address of a global memory region, and
end_address
is the address of the last memory address of the same global memory region.

The compiler uses a global region to allocate variables from (i.e. global variables, procedure arguments, local variables, temporary variables, etc.) When the In Circuit Debugger 2 is enabled, this global region is used for allocation instead.

4.14.8 *icd2_shared_region* Declaration

The *icd2_shared_region* declaration has the following format:

```
icd2_shared_region start_address : end_address
```

where

start_address
is the address of the first memory location where is sharable across multiple data memory banks, and
end_address
is the address of the last memory location where is sharable across multiple data memory banks.

This declaration identifies a region of shared variables to be used when the In Circuit Debugger 2 (ICD2) is enabled.

4.14.9 *interrupts_possible* Declaration

The *interrupts_possible* declaration has the following format:

```
interrupts_possible
```

When this declaration is present, it specifies that interrupts are supported by the microcontroller. Thus, an interrupt procedure named *interrupt* is allowed.

4.14.10 `osccal_at_address` Declaration

The `osccal_at_address` declaration has the following format:

```
osccal_at_address address_expression
```

where

address_expression

is the address address at which the oscillator callibration RETLW instruction is located.

Some of the PIC16 microcontrollers store an oscillator callibration value in program memory as a RETLW instruction. This declaration informs the compiler that the currently defined processor does this and where in program memory the RETLW instruction is located.

4.14.11 `osccal_in_w` Declaration

The `osccal_in_w` declaration has the following format:

```
osccal_in_w
```

When this declaration is present, it specifies that the oscillator callibration value is loaded into the W register at the beginning of processor start up. The compiler will take this value and stuff it into the appropriate oscillator callibration register.

4.14.12 `osccal_register_symbol` Declaration

The `osccal_register_symbol` declaration has the following format:

```
osccal_register_symbol osccal_register_name
```

where

osccal_register_name

is the name of the oscillator callibration register.

The compiler uses this register name when generating code for the `osccal_in_w` or `osccal_at_address` declaration.

4.14.13 `packages` Declaration

The `packages` declaration has the following format:

```
packages package_name = number_of_pins , ...
```

where

package_name

is the name of a package, and

number_of_pins

is the number of pins on the package.

4.14.14 **pin Declaration**

The `pin` declaration has the following format:

```
pin pin_usage , ...
=> pin_declarations
...
<=
```

where

pin_usage

is the name an acceptable pin usage for the pin, and

pin_declarations

is an indented list of pin declarations.

The various *pin_declarations* are listed alphabetically in the sections below.

4.14.14.1 **bind_to Declaration**

The `bind_to` declaration has the following form:

```
bind_to port_name @ bit_number
```

where

port_name

is the name of the port associated with the pin, and

bit_number

is the bit number in the port associated with the pin.

4.14.14.2 **or_if Declaration**

The `or_if` declaration has the following form:

```
or_if pin_usage register_name or_value
```

where

pin_usage

is one of the pin usage names listed by the `pin` declaration one level up,

register_name

is a register name to OR an initialization value into, and

or_value

is the value that is value that is OR'ed in.

4.14.13 packages Declaration

4.14.14.3 **pin_bindings Declaration**

The `pin_bindings` declaration has the following form:

```
pin_bindings package_name = pin_number , ...
```

where

package_name

is a valid package name, and

pin_number

is the pin number associated with the pin for the specified package.

{more goes here}

4.14.15 **shared_region Declaration**

The `shared_region` declaration has the following format:

```
shared_region start_address : end_address
```

where

start_address

is the address of the first memory location where is sharable across multiple data memory banks, and

end_address

is the address of the last memory location where is sharable across multiple data memory banks.

4.14.16 **processor Example**

An example `processor` declaration for the 8-pin PIC12C675 is shown below:

```
processor pic12f675
```

The name of the processor, `pic12f675`, immediately follows `processor`.

```
configuration_address      0x2007
configuration_fill         0x0
```

Next, the configuration word is declared. There is one configuration word that is at address 0x2007 as specified by the `configuration_address` declaration. The configuration word has 8 fields starting with `bg` through `fosc`. While 3 bits of the configuration word are unimplemented, the microcontroller specification sheet says that they should be written as zeros; thus, `configuration_fill` is set to 0.

```
configuration_option  bg:    bg11 =    0x3000
configuration_option  bg:    bg10 =    0x2000
configuration_option  bg:    bg01 =    0x1000
configuration_option  bg:    bg00 =    0x0000
configuration_option  cpd:    on  =      0x000
configuration_option  cpd:    off =      0x100
```

The μ CL Language Specification

Only 2 fields are shown to keep the example size down. The `bg` field has four possible values of `bg11`, `bg10`, `bg01`, and `bg00` with configuration word mask values of `0x3000`, `0x2000`, `0x1000`, and `0` respectively. The `cpd` field has two possible values of `on` and `off` with values of `0x000` and `0x100` respectively.

```
code_bank 0x0 : 0x3ff
```

There is one code bank of 1024 (`0x400`) words in size.

```
data_bank 0x0 : 0x7f
data_bank 0x80 : 0xff
```

There are two data banks, where the first data bank is from 0 through `0x7f` and the other is from `0x80` through `0xff`.

```
shared_region 0x20 : 0x5f
```

This microcontroller is a little strange because all of its memory is accessible from both data banks. Thus, there are no `global_region` declarations.

```
interrupts_possible
```

This microcontroller supports interrupts.

```
osccal_register_symbol $osccal
osccal_at_address 0x3ff
```

The oscillator calibration value is stored as a RETLW instruction at program memory address `0x3ff` as specified by the `osccal_at_address`. The register that contains the oscillator calibration value is `$osccal` as specified by the `osccal_register_symbol` declaration.

```
packages pdip=8, soic=8, dfn_s=8
```

This microcontroller comes in 3 different packages -- `pdip`, `soic`, and `dfn_s`. All three packages have 8 pins each.

```
pin vdd, power_supply
pin_bindings pdip=1, soic=1, dfn_s=1
```

The power supply pin is called either `vdd` or `power_supply`. It is located on pin 1 for all three packages as is listed in the `pin_bindings` declaration.

```
pin gp5_in, gp5_out, tlcki, osc1, clkin, gp5_unused
pin_bindings pdip=2, soic=2, dfn_s=2
bind_to $gpio@5
or_if gp5_in $trisio 16
or_if gp5_out $trisio 0
or_if tlcki $trisio 16
or_if osc1 $trisio 16
or_if clkin $trisio 16
or_if gp5_unused $trisio 16
```

The GP5 pin can be called `gp5_in`, `gp5_out`, `tlcki`, `osc1`, `clkin`, or `gp5_unused`. The `pin_bindings` declaration binds these names to pin 2 on all three packages. When used as a digital input

or output, the pin is connected to bit 5 of the `$gpio` register as specified by the `bind_to` declaration. The six `or_if` declarations specify what value to OR into the `$triso` register depending upon how the pin is used. In general, by setting the `$triso` bit 5 to a 1, sets it as an input. Only when `gp5_out` is specified, does the `$triso` bit 5 get set to 0.

As you can see there a a lot of information to specify for each new microcontroller, but once it is specified, the end-user does not need to be worry about it.

4.15 register Declaration

The `register` declaration has the following form:

```
register name = address , ...
```

where

name

is the name of the register, and

address

is a memory address that accesses the register. There can be multiple addresses for those registers that accessible from different data memory banks.

An example of some `register` delcarations is listed below:

```
register $indf = 0, 0x80
```

The `$indf` register is accessible from both data memory banks at address 0 and 0x80.

```
register $tmr0 = 1
```

The `$tmr0` register is accessible from the first memory bank at address 1.

```
register $pcl = 2, 0x82
```

```
register $status = 3, 0x83
```

Both the `$pcl` and `$status` registers are accessible from both memory banks.

4.16 register_array Declaration

The `register_array` declaration has the following form:

```
register_array name start_address size_expression
```

where

name

is the name of the register array,

first_constant_expression...

is a constant expression that evaluates to the address of the first register, and

size_expression

is a constant expression that specifies how many registers are in the array.

The `register_array` specifies an array of bytes at specific address for a specific size. Frankly, it is kind of a hack and I need to figure out a better way of accomplishing the task.

4.17 `string` Declaration

The `string` declaration has the following form:

```
string strings_name = string_expression
```

where

strings_name
is a name of the string, and
string_expression
is the value of the string.

This declaration defines a global string that can accessed from any procedure.

The following example shows a `string` declaration and some code that access the string:

```
...
string hello = "Hello world!\cr,lf\"
...

procedure main
...
  local index byte

  ...
  index := 0
  loop_exactly hello.size
    call $uart_byte_put(hello[index])
    index := index + 1
  ...
```

The string `hello` is declared to have a value of `"Hello world!\cr,lf\"` in the `string` declaration. Later on, the code prints out the string value using a `loop_exactly` loop. The size of the string is accessed via `hello.size`. Individual characters are accessed via `hello[index]`.

4.18 `uc1` Declaration

The `uc1` declaration has the following form:

```
uc1 major . minor
```

where

major
is the μ CL major version number, and
minor

The μ CL Language Specification
is the μ CL minor version number.

5 procedure Declaration

The `procedure` declaration has the following form:

```
procedure name
=> argument_declarations
   return_declaration
   optional_exact_delay_declaration
   local_declarations
   statement_declarations
...
<=
```

where

name
is the procedure name,
argument_declarations
is either an `arguments_none` declaration or one or more argument declarations,
return_declaration
is either a `returns` declaration, or `returns_nothing` declaration,
optional_exact_delay_declaration
is an optional `exact_delay` declaration,
local_declarations
is one or more `local` variable declarations, and
statement_declarations
is one or more statement declarations.

Everything after the first line is indented by one level.

There are currently two special procedure names -- `main` and `interrupt`. The `main` procedure is the where program execution starts. The compiler places register initialization code into the beginning of the `main` procedure. The `interrupt` procedure is called each time an interrupt occurs. The compiler adds code to save and restore both the `W` and status registers for the `interrupt` procedure. {Eventually, it will save and restore the `FSR` register as well.}

5.1 argument Declaration

The `argument` declaration has the following form:

```
argument name type
```

where

name
is the argument name, and
type
is the argument type -- currently one of `bit` or `byte`.

There is one `argument` declaration for each procedure argument passed in.

5.2 `arguments_none` Declaration

The `arguments_none` declaration has the following form:

```
arguments_none
```

This specifies that the procedure has no arguments.

5.3 `exact_delay` Declaration

The `exact_delay` declaration has the following form:

```
exact_delay cycles_expression
```

where

cycles_expression

is the exact number of processor that the procedure should take.

This declaration that takes exactly *cycles_expression* cycles to execute. This means that conditional code generated for ``if'`, ``&&'`, and ``||'` will be padded with NOP instructions to ensure that each path taken through the code takes exactly the same number of instruction cycles.

5.4 `local` Declaration

The `local` declaration has two possible forms:

Simple `local` Declaration

This form defines a simple local variable. Currently, on type of `bit` and `byte` are supported.

Array `local` Declaration

This form define a local array variable. Currently, only an array of bytes is supported.

5.4.1 Simple `local` Declaration

The simple `local` declaration has the following form:

```
local name type
```

where

name

is the local variable name, and

type

is the local variable type -- currently one of `bit` or `byte`.

All `local` declaration defines a variable that is local to the procedure body. It is not possible for one procedure to directly access the local variables of another procedure. There are no nested variables in this language.

5.4.2 Array `local` Declaration

The array `local` declaration has the following form:

```
local name [ size_expression ] array [ type ]
```

where

name

is the local array variable name,

size_expression

is the array size, and

type

is the type of each array element. Currently, only of `byte` is supported.

All `variable` statements defines a variable that is local to the procedure body. It is not possible for one procedure to directly access the local variables of another procedure. There are no nested variables in this language. A variable that occurs deep within a some nested statements is accessible through out the entire procedure.

5.5 `returns` Statement

The `returns` declaration has the following form:

```
returns type , ...
```

where

type...

is the type for each returned value.

This declaration lists one or more return types. Unlike most other programming languages, μ CL allows for the return of more than one value from a procedure call.

5.6 `returns_nothing` Declaration

The `returns_nothing` declaration has the following form:

```
returns_nothing
```

This declaration specifies that the procedure returns no values.

6 Statements

Statements can only occur within the body of `procedure` declaration. The various statements are listed alphabetically below.

6.1 `assemble` Statement

The `assemble` statement has the following form:

```
assemble
=> assembly_lines
...
<=
```

where

assembly_lines
is a list of assembly statements to insert into the code.

The assembly lines have one of the following forms:

Form	Description
<i>opcode</i>	Instruction with no operands
<i>opcode operand</i>	Instruction with one operand
<i>opcode operand1, operand2</i>	Instruction with two operands
<i>:label</i>	Label definition (note that <code>:</code> is before label name)
<i>define name = expression</i>	Constant definition

The following instruction opcodes are currently supported:

Opcode	Operands	Description	Operation
<code>addlw</code>	<code>k</code>	ADD <code>k</code> to <code>W</code>	<code>W := W + k</code>
<code>addwf</code>	<code>f, d</code>	Add <code>W</code> and <code>f</code>	<code>Wlf := W + f</code>
<code>andlw</code>	<code>k</code>	AND <code>k</code> to <code>W</code>	<code>W := W & k</code>
<code>andwf</code>	<code>f, d</code>	AND <code>W</code> and <code>f</code>	<code>Wlf := W & f</code>
<code>bcf</code>	<code>f, b</code>	Bit clear <code>f</code>	<code>f@b := 0</code>
<code>bsf</code>	<code>f, b</code>	Bit set <code>f</code>	<code>f@b := 1</code>
<code>btfsc</code>	<code>f, b</code>	Bit test <code>f</code> , skip if clear	if <code>!f@b</code> then skip
<code>btfss</code>	<code>f, b</code>	Bit test <code>f</code> , skip if set	if <code>f@b</code> then skip
<code>call</code>	<code>a</code>	Call subroutine at <code>a</code>	<code>call a()</code>
<code>clrf</code>	<code>f</code>	Clear <code>f</code>	<code>f := 0</code>
<code>clrwdt</code>		Clear watchdog timer	
<code>clrwf</code>		Clear <code>W</code>	<code>W := 0</code>
<code>comf</code>	<code>f d</code>	Complement <code>f</code>	<code>Wlf := ~f</code>

The μ CL Language Specification

decf	f, d	Decrement f	Wlf := f - 1
decfsz	f d	Decrement f, skip if 0	Wlf := f - 1; if Wlf = 0 then skip
goto a	a	Go to address a	goto a
incf	f, d	Increment f	Wlf := f + 1
incfsz	f, d	Increment f, skip if 0	Wlf := f + 1; if Wlf = 0 then skip
iorwf	f, d	Inclusive OR W and f	Wlf := W f
iorlw	k	Inclusive k to W	W := W k
movf	f, d	Move f	Wlf := f
movwf	f	Move W to f	f := W
movlw	k	Move k to W	W := k
nop		No Operation	
option		Option instruction	
retfie		Return from interrupt	
retlw	k	Return from subroutine with k in W	W := k ; return
return		Return from subroutine	return
rlf	f, d	Rotate left through carry	
rrf	f, d	Rotate right through carry	
sleep		Enter low power mode	
sublw	k	Subtract W from k	W := k - W
subwf	f, d	Subtract W from f	Wlf := f - W
swapwf	f, d	Swap nibbles in f	Wlf := ((f & 0xf) << 4) ((f >> 4) & 0xf)
xorwf	f, d	Exclusive OR W and f	Wlf := W ^ f
xorlw	k	Exclusive OR k to W	W := W ^ k

The example below shows the `assemble` statement in action:

```
...
procedure left_rotate
  argument value byte
  returns_nothing

  # This procedure will return the value of
  # rotation value to the left by 8 bits.

  # {value} is sitting in W:
  $c := value@7
  assemble
    rlf left_rotate__value, w
  return
```

This procedure will take an 8-bit byte value and rotate it left in 8-bits. The first statement will set the carry bit (\$c) to the the high order bit of value. The `assemble` statement executes an RLF instruction. The full name of the argument is the procedure name `left_rotate`, followed by two underscores `__`, followed by the argument name `value`, resulting in `left_rotate__value`. The result is in the W register, so a simple RETURN instruction ends the procedure.

6.2 Assignment Statement

The assignment statement has two possible forms:

Simple Assignment

Simple assignment takes the result of an expression and assigns it to a variable.

Multiple Assignment

Multiple assignment takes several results from an expression and assigns it to two or more variables.

The assignment statement is unique in that it is the only statement that does not start off with keyword. Instead, the parser prescans the line and if it encounters an assignment operator (`:=`), it assumes the entire statement is an assignment statement.

6.2.1 Simple Assignment Statement

The simple assignment statement has the form:

variable := *expression*

where

variable

is either a simple variable or an array variable (e.g. *array_name*[*index_expression*]),
and

expression

is an evaluated expression that is assigned into *variable*.

6.2.2 Multiple Assignment Statement

The multiple assignment statement has the form:

*variable*₁ , ... , *variable*_{*n*} := *multiple_expression*

where

*variable*_{*i*}

is either a simple variable or an array variable (e.g. *array_name*[*index_expression*]),
and

multiple_expression

is an evaluated expression that has multiple values.

The type of the variables and expressions must match up.

The following example shows a simple multiple assignment:

```
procedure main
...
  local a byte
  local b byte
```

```
...
# Swap A and B:
a, b := b, a
```

Another example is shown below:

```
procedure $uart_byte_read
  argument usec_wait byte
  returns bit, byte

  while ...
    value := ...
    # No time out:
    return 0, value
  # Time out:
  return 1, 0

procedure main
  ...
  local command byte
  local time_out bit
  ...

  time_out, command := $uart_byte_read(100)
  if time_out
    ...
  else
    ...
  ...
```

For this code fragment, there is a procedure called `$uart_byte_read` that takes a single argument `usec_wait` that specifies the maximum wait time for read a byte form the UART (Universal Asynchronous Receiver Transmitter). If a byte is received before the time out period has elapsed, the time out bit is returned a 0 and the UART value is returned as they byte value. Otherwise, a time out has occurred and the time out bit is returned as a 1 and the value is returned as 0. Later on in the main procedure, `$uart_byte_read` is invoked and its two return values are assigned to the local variables `time_out` and `command` via a multiple assignment.

6.3 call Statement

The `call` statement has the following form:

```
call procedure_name ( argument_expressions )
```

where

procedure_name

is the name of the procedure to be called, and

argument_expressions

is a list of zero, one, or more expressions to be evaluated and passed as arguments to the procedure.

The number and type of each argument expressions must match the procedure types.

The `call` statement is used to call procedures that have no return values (i.e. `returns_nothing`). In addition, the `call` statement can be used to invoke a procedure that returns values, but where the return

values are not needed.

6.4 `delay` Statement

The `delay` statement has the following form:

```
delay
  expression
=> statements
...
<=
```

where

expression

is a constant expression that specifies the exact number of instruction cycles to be to be executed, and

statements

is a nested sequence statements that are executed. Each statement in *statements* is compiled to have uniform execution time. Conditional code, like ``if'`, ``&&'`, and ``||'`, is padded with `nop` instructions to cause the execution time to be uniform.

Each procedure called in *statements* must have a `exact_delay` declaration in its `procedure` declaration.

The example below delays for 50 μ Sec

```
# 5 cycles per microsecond
constant usec = 5

procedure main
  ...

  # Delay for 50 usec:
  delay 50 * usec
  do_nothing
```

6.5 `delay_set` Statement

The `delay_set` statement has the following form:

```
delay_set
  expression
=> statements
...
<=
```

where

expression

is a constant expression that sets the delay value that the compiler will use for the nested statements, and

statements

is a nested sequence of statements.

This statement is used to tell the compiler exactly how many cycles a give chunk of code is supposed to take.

6.6 do_nothing Statement

The `do_nothing` statement has the following form:

```
do_nothing
```

In fact, this statement does not do anything.

6.7 if Statement

The `if` statement has the following form:

```
if
  bit_expression
=> statements
...
<=
else_if
  bit_expression
=> statements
...
<=
else
=> statements
...
<=
```

where

bit_expression

is an expression that evaluates to a bit value, and

statements

is a nested sequence of statements to be executed if *bit_expression* evaluates to 1 (i.e. true.)

There can be zero, one or more `else_if` clauses. The `else` clause is optional. The first `if` clause and its nested statement block is required.

At most, only one nested statement block is executed. The first *bit_expression* that evaluates to 1 (i.e. true) causes the associated statement block to be executed. If none of the *bit_expression*'s evaluate to 1, the `else`

clause statement block is executed (if present.)

6.8 `loop_exactly` Statement

The `loop_exactly` statement has the following form:

```
loop_exactly
  expression
=> statements
  ...
<=
```

where

expression
is the number of times the expression is executed, and
statements
is the nested statement block that is executed each iteration through the loop.

Note that *expression* must evaluate to a non-zero value. {This is a bogus restriction and needs to be fixed.}

6.9 `loop_forever` Statement

The `loop_forever` statement has the following form:

```
loop_forever
=> statements
  ...
<=
```

where

statements
is a nested statement block that is continuously executed.

For embedded applications, the `main` procedure typically has a `loop_forever` statement, since it never makes sense to return from `main`.

6.10 `return` Statement

The `return` statement has the following form:

```
return expression , ...
```

where

expression

is an expression that is evaluated and returned from the procedure.

The number and type of expressions in the `return` statement must match the types in the `returns` clause for the enclosing procedure declaration. If the procedure declaration has a `returns_nothing` clause, the `return` statement must not have any expression.

6.11 `switch` Statement

The `switch` statement has the following form:

```

switch
  switch_expression
=>  case_maximum
    maximum_expression
    case
      case_expression1
    =>  statements
      ...
    <=
    ...
    case
      case_expressionN
    =>  statements
      ...
    <=
    default
    =>  statements
      ...
    <=
  <=

```

where

switch_expression
is an expression is used to steer the switch,
maximum_expression
is a constant expression that specifies the maximum possible value of
switch_expression,
case_expression_i
is a constant expression, and
statements
is a nested statements block.

The *switch_body* consists of a sequence of one or more *case_clause*'s, followed by an optional *default_clause*. The expression is evaluated and control is transferred to one of the clauses. Only *case_clause*'s, a *default_clause*, comments and blank lines can occur inside of *switch_body*.

An example `switch` statement is shown below:

```
switch command >> 5
  case_maximum 7
  case 0
    # First command:
    ...
  case 1, 2
    # Second and third commands:
    ...
  case 3
    # Forth command:
    ...
  default
    # All other commands are undefined:
    ...
```

6.12 `watch_dog_reset` Statement

The `watch_dog_reset` statement has the following form:

```
watch_dog_reset
```

This statement generates the code to reset a watch dog timer in a microcontroller.

6.13 `while` Statement

The `while` statement has the following form:

```
while
  while_expression
=> statements
...
<=
```

where

while_expression
is an expression the returns a bit value,
statements
is the indented statemenst for the while loop.

This statement will evaluate *while_expression* and if it evaluates to 1 (i.e. true) it will execute the indented statement block *statements*. It will do this repeatably until *while_expression* evaluates to 0 (i.e. false.)

7 Expressions

Expressions in μ CL are modeled after expressions in ANSI-C. There are a few differences between μ CL expressions and C expressions.

7.1 Differences from C Expressions

If you do not know C-expressions, you should probably skip reading this section.

The differences from C-expressions itemized below:

- The μ CL assignment operator is ``:='`, not ``='` as in C.
- Serial assignment (e.g. ``a := b := c := 0'`) is not allowed in μ CL, whereas it is allowed in C.
- In-line assignment (e.g. ``a < (b := b + 1)`) is not allowed in μ CL, whereas it is allowed in C.
- μ CL supports multiple assignment (e.g. ``a, b := b, a'`) whereas C does not.
- In μ CL, procedures can return more than one value; whereas in C only one value can be returned.
- The order of execution is strictly left-to-right in μ CL, whereas in C it is frequently right-to-left or undefined.
- In μ CL, the equality operator is ``='` and not ``=='` as in C.
- The precedence of the bitwise AND (``&'`) and bitwise OR (``|'`) instructions is higher than the relational operators (``=`, ``!=`, ``<`, and ``>=`). In C, the precedence is lower.
- There are no auto-increment (``++`) and auto-decrement (``--`) operators in μ CL.
- μ CL has an extra bit selection operator called ``@'`.
- μ CL does not implement records, so it does not implement either the C ``.'` or ``->` operators.
- In μ CL, the logical statements (e.g. ``if`, ``while`, etc.) and logical operators (e.g. ``&&`, ``||`, and ``!'`) all require an expression of type ``bit'`. No sloppiness with looking at the least significant bit of the expression is permitted. You either use a relational operator (``=`, ``!=`, ``<`, and ``>=`) or a bit selector operator (``@'`) that returns an expression of type ``bit'` to convert a byte expression into a bit one.
- There is no arithmetic if (``E1 ? E2 : E3'`) in μ CL yet.

7.2 What is Precedence?

This section is for reference purposes for those people who are not familiar with the concept of operator precedence.

In regular arithmetic expressions like ``1 + 2 \times 3 - 4 \times 5 + 6'`, the `` \times '` operator has a higher precedence than the ``+'` and ``-'` operator. Thus, multiplication and division are performed before addition and subtraction. This can be made more explicit by adding parenthesis as follows -- ``1 + (2 \times 3) - (4 \times 5) + 6'`. In many programming languages, there are usually a couple of dozen operators and each of them have different precedences.

As usual, parenthesis are used to change the order operation execution. Thus, ``(1 + 2) \times (3 - 4) \times (5 + 6)` causes the addition and subtraction to occur before the multiplication.

The sections below list operators in μ CL expressions from lowest precedence to highest precedence. The lowest precedence operators are performed after all higher precedence operators have been done.

7.3 Assignment Operator (:=)

The assignment operator is:

``:='`

Straight assignment ($L := R$)

μ CL supports multiple assignment. In multiple assignment, a list of expressions to the right are assigned to a list of variables on the left. All of the expressions to the right are evaluated and stored in temporary variables before any of the assignments take place. The example below will swap the contents of the ``a'` and ``b'` variables:

```
a, b := b, a
```

which is equivalent to:

```
T1 := b
T2 := a
a := T1
b := T2
```

where T1 and T2 are temporaries.

Another form of multiple assignment occurs when a procedure returns more than one value. An example should help clarify this. Let us assume that the procedure `plus_minus(a, b)` returns $a+b$ and $a-b$. This procedure would be written as follows:

```
procedure plus_minus
  argument a byte
  argument b byte
  returns byte byte

  return a + b, a - b
```

and we can invoke ``plus_minus'` as follows:

```
a_plus_b, a_minus_b := plus_minus(a, b)
```

The first value returned from `plus_minus` is $a+b$ and it is assigned to the variable `a_plus_b`. The second value returned from `plus_minus` is $a-b$ and it is assigned to the variable `a_minus_b`.

7.4 Comma Operator (,)

The comma operator `,` does not perform any computation per se. All it does is cause other expressions of higher precedence to be executed in left-to-right order. It is used to separate variables and expressions in multiple assignment statements (e.g. `a, b := b, a`) and it is used to separate the arguments passed to procedures.

A lot of C programmers have programmed in C for years without realizing that in C most implementations use right-to-left order of execution. The C language specification does not mandate either left-to-right or right-to-left execution order. However, the first implementations of C pretty uniformly implemented right-to-left execution in order to support variadic (i.e. many variables) functions like ``printf'`. Pretty much all

subsequent C implementations after that have followed the same rule. The order of execution only crops up when the expressions involve side effects. For example, in C:

```
(void)printf("first:%d second:%d\n", get_byte(), get_byte())
```

does not do what most people think it does. The first call to `get_byte` is the right-most one followed by the left-most one. Thus, in the example above, the first byte that is read is actually printed as the decimal number next to `second` and vice versa. This is probably not what the C programmer had in mind.

In μ CL, execution order is always left-to-right and there are no surprises like there are in C.

7.5 Conditional OR (`||`)

The expression `L || R` returns 1 if either `L` or `R` evaluate to 1. It is a conditional OR because it will not execute `R` if `L` returns a 1.

Consider the code fragment below:

```
if (denominator = 0 || numerator/denominator = 0) {  
    # ...  
}
```

In this example, it would be erroneous to divide the numerator by zero, so we can check for zero beforehand and only if it is non-zero does the division take place.

If it is desirable to always execute the left and the right side before computing the OR, the bitwise OR (`|`) operator can be used.

7.6 Conditional AND (`&&`)

The expression `L && R` returns 1 if both `L` and `R` evaluate to 1. It is a conditional AND because, it will not bother to execute `R` if `L` returns a 0. Consider the code fragment below:

```
if (denominator != 0 && numerator/denominator > 0) {  
    # ...  
}
```

In this example, it would be erroneous to divide the numerator by zero, so we can check for zero beforehand and only if it is non-zero does the division take place.

If it is desirable to always execute the left and the right side before computing the AND, the bitwise AND (`&`) operator can be used.

The precedence of `&&` is higher than `||`. Thus,

```
A && B || C && D
```

would be executed as

```
(A && B) || (C && D)
```

As usual, it never really hurts to add parentheses to improve readability.

7.7 Relational Operators and Bit Selection (<, =, >, <=, !=, >=, @)

There is one bit selection operator and six relational operators:

$L @ N$	The N 'th bit of L .
$L < R$	L less than R
$L = R$	L equal to R
$L > R$	L greater than R
$L <= R$	L less than or equal to R
$L != R$	L not equal to R
$L >= R$	L greater than or equal to R

The equality operators (= and !=) work for operands of type `byte` and `bit` and all the other operators (<, >, <=, >=, and @) only work for operands of type `'byte'`.

NOTE: bit equals and not-equals is not implemented yet. The work around is quite ugly -- $A \ \&\& \ B \ || \ !A \ \&\& \ !B$ for equality and $A \ \&\& \ !B \ || \ !A \ \&\& \ B$ for inequality.

It is not permissible to stick an extra space between any of the two character relational operators (<=, !=, and >=).

The bit selection operator is not present in C and is unique to µCL. $A @ N$ is the N 'th bit of A . Both A and N must be of type `byte`. The result is of type `bit`. $A @ 0$ selects the least significant of A . $B @ 7$ selects the most significant bit of B .

NOTE: Currently, the right operator of bit selection must be a constant. A reasonable work around would be $A \ \& \ 1 \ << \ N \ != \ 0$ which is grouped as $(A \ \& \ (1 \ << \ N)) \ != \ 0$; unfortunately, $1 \ << \ N$ where N is not a constant is also unimplemented. Sigh.

C programmers should note that the equality operator consists of a single = not the double == used in C.

In C, the `char` type usually stands for a byte. The ANSI-C standard allows `char` to be either signed or unsigned. This ambiguity causes all sorts of grief when porting C code between C compilers. There is no such ambiguity in µCL, the `byte` type is always represents non-negative numbers between 0 and 255 inclusive. Thus, in µCL, 128 is always greater 127 whereas in C (`char`) 128 is sometimes less than (`char`) 127 and sometimes greater.

The precedence of the relation operators is greater than both conditional AND (&&) and conditional OR (||). Thus, the following code fragment

```
0c'a' <= c && c <= 0x'z'
```

is grouped as

```
(0c'a' <= c) && (c <= 0x'z')
```

7.8 Addition and Subtraction (+, −)

The addition operator is + and the subtraction operator is −. These operators are only defined for expressions of type `byte`. The order of execution is strictly left to right. Thus,

```
a + b - c + d
```

is evaluated as:

```
((a + b) - c) + d
```

Currently, µCL does not implement 16-bit or 32-bit arithmetic. You can use the following work-around:

```
variable a_lo byte
variable a_hi byte
variable b_lo byte
variable b_hi byte
variable c_lo byte
variable c_hi byte

c_hi := a_hi + b_hi
c_lo := a_lo + b_lo
if $c
    c_hi := c_hi + 1
```

The precedence of addition and subtraction is higher than the relational operators. Thus,

```
a + b > 20
```

is grouped as

```
(a + b) > 20
```

The more complicated expression

```
a + b > 20 && c - d <= e + f
```

is grouped as

```
((a + b) > 20) && ((c - d) <= (e + f))
```

7.9 Multiplication, Division, and Modulo (*, /, and %)

NOTE: Currently, none of these operators are implemented for generated code. It is permissible to use them in constant expressions though.

The µCL Language Specification

The multiplication operator is `*`, the division operator is `/`, and the modulo operator is `%`. The modulo operator returns the remainder of a division. These operators are only defined for expressions of type ``byte'`. The order of execution is strictly left to right. Thus,

`a * b / c * d`

is evaluated as:

`((a * b) / c) * d`

The precedence of multiplication and division is higher than addition and subtraction. Thus,

`a * b + c / d`

is grouped as

`(a * b) + (c / d)`

The more complicated expression

`a * b + c > 20 && d / e - f < 30`

is grouped as

`((a * b) + c) > 20 && ((d / e) - f) < 30`

7.10 Bitwise OR (`|`)

The expression `A | B` computes the bitwise OR of `A` and `B`. `A` and `B` must both be the same type. This operation is defined for both type `bit` and `byte`. Execution order is strictly left to right.

NOTE: Currently, bitwise OR of type `bit` is not implemented. Usually, you can make do with the conditional OR operator (`| |`).

The precedence of bitwise OR is higher than multiplication, division, addition, subtraction and all relational operators.

`a | 1 + b | 2 * c | 3`

is grouped as

`(a | 1) + ((b | 2) * (c | 3))`

The expression

`a | 1 >= b | 2`

is grouped as

`(a | 1) >= (b | 2)`

In C, this expression would group as

```
(a | (1 >= b)) | 2
```

which is pretty counter-intuitive.

7.11 Bitwise AND (&)

The expression $A \& B$ computes the bitwise AND of A and B . A and B must both be the same type. This operation is defined for both type `bit` and `byte`. Execution order is strictly left to right.

NOTE: Currently, bitwise AND of type `bit` is not implemented. Usually, you can make do with the conditional AND operator (`&&`).

The precedence of bitwise AND is higher than bitwise OR, multiplication, division, addition, subtraction and all relational operators.

```
a & 1 | b & 2 | c & 4
```

is grouped as

```
((a & 1) | (b & 2)) | (c & 4)
```

The expression

```
a & 0xf = 0xb
```

is grouped as

```
(a & 0xf) = 0xb
```

In C, this expression would group as

```
a & (0xf >= b)
```

which is pretty counter-intuitive.

7.12 Bitwise XOR (^)

The expression $A \wedge B$ computes the bitwise XOR (eXclusive OR) of A and B . A and B must both be the same type. This operation is defined for both type `bit` and `byte`. Execution order is strictly left to right.

NOTE: Currently, bitwise XOR of type `bit` is not implemented.

Bitwise XOR is used to toggle bits. If you want to complement the third bit in a register, try the following:

```
a := a ^ 4
```

The precedence of bitwise XOR is higher than bitwise AND, bitwise OR, multiplication, division, addition, subtraction and all relational operators. This means you can twiddle bits, mask them off, the then assemble them together without having to fight the operator precedence. For example,

```
a ^ 9 & 0xf | 0xa0
```


is grouped as

```
((a ^ 9) & 0xf) | 0xa0
```

and toggles the first and forth bits of `a`, masks off the four high order bits, and sets the fifth and seventh bits.

7.13 Shift Operators (>>, <<)

The shift right operator is `>>` and the shift left operator is `<<`. Order of operation is strictly left to right. $A \ll N$ causes A to be shifted left by N bits with 0 being shifted into the least significant bits. $B \gg N$ causes B to be shifted right by N bits with 0 being shifted into the most significant bits. Both the left and right operands the shift operators must be of type `byte`. Thus, $A \ll 1$ is equivalent to efficiently multiplying by 2 and $A \ll 2$ is equivalent to multiplying by 4. Similarly, $B \gg 1$ is equivalent to dividing by 2 and $B \gg 2$ is equivalent to dividing by 4.

NOTE: Currently, the code for shifting by a non-constant expression has not been implemented.

The precedence of the shift operators is greater than the bitwise operators. Thus,

```
a >> 4 | a << 4
```

is grouped as

```
(a >> 4) | (a << 4)
```

Incidentally, this piece of code results in a value where the nibbles of `a` have been exchanged.

7.14 Unary Operators (–, !, ~)

There are three unary operators:

<code>– R</code>	Minus R
<code>~ R</code>	Bitwise NOT of R
<code>! R</code>	Logical NOT of R

The minus (`–`) and bitwise NOT (`~byte`). The logical NOT operator (`!`) only works on a operand of type `bit`

The precedence of the unary operators is higher than all the arithmetic and bit twiddling operators. Thus,

```
–a – b
```

is grouped as

```
(–a) – b
```

and

```
~a & b
```

7.12 Bitwise XOR (^)

is grouped as

$(\sim a) \ \& \ b$

NOTE: Currently, there is a bug in the expression parser such that $a \ \& \ \sim b$ is not properly parsed.

NOTE: I've often thought that logical NOT (!) should have a precedence between && and bit selection (@). Thus, $!A@N$ would group as $!(A@N)$ rather than $(!A)@N$. Similarly, I've often thought that that unary minus should have a precedence right above multiplication ().*

7.15 Array Operator (. . . [. . .])

The array operator looks as follows:

$L \ [\ R \]$

where L is an expression that evaluates to either a string or a byte array, and R is an expression that evaluates to a byte. When the array operator occurs to the left of an assignment, a byte value is stored into the array; otherwise, a byte value is fetched.

7.16 Dot Operator ($L.size$)

Currently, the dot operator has only one forms:

$L.size$

Fetch the size of L where L is an expression that evaluates to either a string or byte array.

7.17 Procedure Invocation ($P (. . .)$)

Procedure invocation is the act invoking a procedure using its return value (or values) in an expression. $P ()$ invokes a procedure P with no arguments, $P (A1)$ invokes a procedure with a single argument expression $A1$, and $P (A1, A2)$ invokes procedure P with argument expressions $A1$ and $A2$.

The arguments to the procedure are evaluated in strict left to right order.

It is an error to invoke a procedure that does not return any return value (i.e. `returns_nothing`) in an expression.

It is legal to invoke a procedure in an expression that returns multiple values, provided the returned values are directly passed on as arguments to another procedure invocation (or a multiple assignment.) For example, assume that `swap(a, b)` returns `b, a`. Further assume that `p3` is a procedure that takes three arguments. `p3(swap(a, b), c)` is legal and equivalent to `p3(b, a, c)`.

The precedence of procedure invocation is higher than all other operators.

7.18 Debugger Support

The µCL compiler provides support for debugging programs. The basic concept is that the user identifies a number of routines that are to be compiled with extra debugger support. A routine that has debugger support compiled in can be stopped at the beginning of each statement in the procedure. In addition, the user can view the call stack and all local and global variables for all procedures whenever the program is stopped at the beginning of a statement.

Compiler support starts with the `debug_base` declaration. This specifies the base name of the libraries needed. For example:

```
debug_base my_debugger
```

specifies the prefix for all debugger libraries, procedures, and variables will be `my_debugger`. If no `debug_base` declaration is present, the default prefix is `$debug`. The `debug_base` declaration allows the compiler to interface to a multitude of different debuggers. More importantly, the way that the debugger communicates with the debugger library is strictly encapsulated by the debugger library. Thus, one library may use a UART for communication and another may use some extra pins and bit bang the data back and forth with the debugger. The compiler simply does not care how the debugger library implements the debugger communication protocol.

The user specifies which procedures to generate debugging information for via the `debug` declaration. For example,

```
debug main, my_proc1, my_proc2
```

will cause debugging code to be inserted into the code generated for the procedures `main`, `my_proc1` and `my_proc2`. Since there can be substantial code expansion, the user is encouraged to only generate debugger support code for procedures that are likely to require breakpoints during a debugging session. If the user discovers that they need to plant a breakpoint on a procedure that is not listed in the `debug` declaration, the `debug` declaration is changed, the code is recompiled, reinstalled, and debugging resumes.

In general, the debugger library is split into 4 library files called `prefix0.ucl` through `prefix3.ucl`. The digit 0-3 specifies which code bank the debugger library file lives in. The bulk of the debugger library resides in `prefix0.ucl` which is always loaded. The other debugger files are loaded only as needed.

There are a total of 13 entry points defined by the debugger library files. There is one initialization entry point, and 3 three other entry points that are defined for each of the possible 4 code banks. Thus, $1+3\times 4=13$. These entry points are:

prefix_initialize

The debugger library entry point that is called from `main` before any other calls into the debugger library.

prefixN_enter

The debugger library entry point for when a procedure is first called.

prefixN_breakpoint

The debugger library entry point for when an active breakpoint is encountered.

prefixN_exit

The debugger library entry point for when a procedure returns.

The μ CL Language Specification

The *prefix* is the prefix specified by the `debug_base` declaration. The *prefixN* is the same as *prefix*, but with the code bank number appended onto the end. The code bank digit specifies which code bank the entry point lives in. The reason for providing a separate entry point for each code bank is to reduce the amount of code bank swapping code that is generated for each breakpoint. For example, assume that the following code fragment is having debug code generated:

```
origin 0
data_bank 0
procedure main
    arguments_none
    returns_nothing

    call bank1()
    call bank2()
    call bank3()
```

will have calls to the following debugger library entry points:

```
call prefix_initialize()
call prefix0_enter()

call prefix0_breakpoint()
call bank1()

call prefix1_breakpoint()
call bank2()

call prefix2_breakpoint()
call bank3()
call prefix0_exit()
```

Notice that the breakpoint debugger library entries are always use the most recent code bank rather than the "upcoming" code bank. The entry and exit debugger library entries are always using the procedure's home code bank.

The *prefix0.ucl* debugger library file must define a single global shared variable:

```
prefix_current
    The current number of the procedure being debugged.
```

Each procedure in the `debug` declaration is given a number between 0 and 254. The number 255 is what *prefix_current* is initialized to by *prefix_initialize*.

For each procedure listed in the `debug` declaration, the compiler allocates two variables:

```
proc_name_caller
    This variable contains the procedure number of the procedure that is calling this
    procedure.
proc_name_breakpoint
    This variable contains the breakpoint number of the current statement being executed.
```

The μ CL Language Specification

where *proc_name* is the procedure name. Upon entry into the procedure, the compiler generates the following code sequence:

```
proc_name_caller := prefix_current
prefix_current := proc_name_number
call prefixN_enter()
```

where *prefix_current* is a global shared variable (i.e. a variable that is accessible from on data banks at the same time.)

Upon exit from a procedure the following code is generated:

```
prefix_current := proc_name_caller
```

Using the code generated for procedure entry and exit, the debugger can chain back through *prefix_current* to generate the call stack. It reads the value of "current", looks up the associated "caller" repeatable until it gets a "caller" of 255, at which point the call stack is at the end. For each procedure in the call stack, the debugger must fetch the value of the *proc_name_caller* variable.

For each statement, the compiler generates the following code:

```
proc_name_breakpoint := N
if breakpointN
call prefixN_breakpoint()
```

A single procedure can have up to 256 statement breakpoints. The compiler allocates a single bit for each breakpoint. The compiler does **not** generate code to initialize these bits -- it is the responsibility of the debugger initialize all of the breakpoint bits.

Finally, the compiler outputs a ton of information in a *base_name.info* file. This file provides the procedure names, variable names and types, and the locations of all breakpoint bits. A debugger can read the .info file and figure everything out (heh-heh.)

Next, some example code fragments are provided to give some idea of how to write some debugger code. Here is *prefix0.ucl*:

```
ucl 2.0
# Copyright (c) year by your name
# All rights reserved.

# This portion of the debug library lives in code
# bank 0.

library_code_bank 0
data_bank 0

shared prefix_current byte # Currently debugged proc.
shared prefix_status byte  # Current $status reg.
global prefix_why          # Why we paused

constant prefix_debug_address = 0
constant prefix_why_initialize = 0
constant prefix_why_enter = 1
constant prefix_why_breakpoint = 2
```

The μ CL Language Specification

```
constant prefix_why_exit = 3

procedure prefix_initialize
  arguments_none
  returns_nothing

  # This procedure is called before anything else.

prefix_current := 255
prefix_why := prefix_why_initialize
  assemble
  prefix_debug_loop

procedure prefix0_enter
  arguments_none
  returns_nothing
  return_suppress

  # This procedure is called when a procedure
  # is entered. We know we are in code bank 0,
  # but we do not know our data bank:

prefix_status := $status
  assemble
  # Goto the shared routine:
  prefix0_enter

# The code for prefix0_breakpoint and
# prefix0_exit is the same except that
# the goto goes to prefix_breakpoint
# and prefix_exit respectively.

procedure prefix_enter
  arguments_none
  returns_nothing
  return_suppress

  # This procedure is called when a procedure is
  # entered.

  # Force into code bank 0:
  $status := 0

  # See whether we need to do anything:
prefix_enter_stop
prefix_why := prefix_why_enter
  assemble
  prefix_debug_loop

  # Otherwise, we can just return without doing anything:
  assemble
  prefix_done

procedure prefix_exit
  arguments_none
  returns_nothing
  return_suppress

  # This procedure is called when a procedure is
  # exited. At this point we know the code
  # bank is 0 and the data bank is 0.
```

The µCL Language Specification

```

    $status := 0
prefixexit_stop
    prefix_why := prefix_why_exit
        assemble
    prefix_debug_loop

    # Otherwise, we can just return without doing anything:
    assemble
prefixdone

procedure prefix_breakpoint
    arguments_none
    returns_nothing
    return_suppress

    # This procedure is called when a procedure
    # breakpoint is encounter. At this point we
    # know tha the code bank is 0 and the data
    # bank is 0.

prefix_why := prefix_why_breakpoint

    # Run into the next procedure:

procedure prefix_debug_loop
    arguments_none
    returns_nothing
    return_suppress

    # This procedure is called to get some attention
    # from the debugger. At this point we know that
    # both the code bank and data bank are 0.
prefixwhy variable contains a code
    # explains why the debugger attention is needed.

    local done bit
    local echo_eat byte
    local send bit
    local send_value byte
    local state byte
    local receive bit
    local receive_value byte

    # Get attention of debugger:

    state := 0
    done := $false
    while !done
        switch state
            case 0
                # Send the attention address byte:
                $tx9d := $true
                send := $true
prefixdebug_address
                state := 1
            case 1
                # Send the "why" byte:
                $tx9d := $false
                send := $true
prefixwhy :=

```

The μ CL Language Specification

```
        receive := $true
        state := 2
    case 2
        # We've got a command:
        state := receive_value

        #...
    lastcase
        done := $true

    # Send a byte to the debugger:
    if send
        while !$txif
            do_nothing
        $txreg := send_value
        echo_eat := $true
        send := $false

    # Receive a byte:
    while echo_eat || receive
        while !$rcif
            do_nothing
        receive_value := $rcreg
        if $oerr
            $scren := $false
        if $ferr
            $scren := $false
        $scren := $true
        if echo_eat
            $echo_eat := false
        else_if receive
            $receive := false

    # Run into the next procedure:

procedure prefix_done
    arguments_none
    returns_nothing

    assemble
        # Restore status:
        prefix $status, w
        movwf $status
    # Implicit Return instruction executed next will
    # return to the proper code bank.
```

The code for a debugger entry point in one of the other code banks looks as follows:

```
procedure prefix1_enter
    arguments_none
    returns_nothing
    return_suppress

    # This procedure is called when a procedure
    # is entered. We know we are in code bank 0,
    # but we do not know our data bank:

    # Save the status:
    prefix_status := $status
    # Force into code bank 0:
```


The μ CL Language Specification

```
$pclath := 0
assemble
  # Goto the shared routine:
  prefixenter
```

Copyright © 1999-2007 by Wayne C. Gramlich. All rights reserved.