UC-X Macro Assembler Manual

Version 0.7.0 DRAFT

Contents

Command Line Arguments

UCAsm is used from the command line, as follows.

```
ucasm [options] filename.asm
```

Command line arguments are case sensitive and must be specified before the filename. Multiple options that do not require an argument can be combined, for example `-qv` **is the same as specifying** `-q -v`**.**

The available options are shown in the following table.

Option	Purpose
`-q`	**Quiet mode**
`-v`	**Verbose mode**
`-N`	**Case insensitive labels**
`-S`	**Do not add system include path to the include search path**
`-I <path>`	**Add <path> to the include search path. May be used more then once to add multiple paths**
`-o <filename>`	**Set <filename> as the output filename**

Verbose Mode and Quiet Mode

Verbose, Quiet and Quiet Verbose mode allow you to control how much information is displayed while assembling.

The `-q` **and** `-v` **options are interpreted as follows.**

- **If only** `-q` **is specified, display nothing unless there is an error or warning**
- **If both** `-q` **and** `-v` **are specified, display only the filename unless there is an error or warning**
- **If -v is specified, display the filename plus current pass information**
- **If neither** `-q` **nor** `-v` **are specified, display only the number of errors / warnings**

The version and copyright information is only printed in non-quiet verbose mode, or if no arguments were provided.

Addresses

UCAsm was designed to give complete control to the programmer over where their code and data ends up in memory. The only restriction is the automatically generated Spin loader will initialize cog 0 from hub address $20, thus you must always have some code at $20.

As the Propeller has two separate memory spaces, the assembler maintains two address pointers that determine where code and data is assembled. The hub address pointer is the current byte address in hub memory where the data will be in the image. The cog base address pointer is the current base address used to calculate the correct cog address for the source and destination fields of instructions.

Although you will never need to do so manually, to convert from a hub address and cog base pointer, use the following formula: (Hub Address – Cog Base Address) / 4.

The assembler never makes any assumptions about which parts of your code form cogs. It simply uses the above formula to resolve symbols to the, hopefully, correct cog address when needed. This makes it possible to have overlapping data, an example of which can be found below.

Assembly Address Directives

COG [*hub address in bytes*]

If the address is specified, set both the cog base address and the hub address to the address. If the address is not specified, the cog base address will be set to the current hub address.

The `COG` **directive is the main directive you will use for setting the assembly address. It can be thought of as a hub memory** `ORG` **directive that also sets up the assembler state such that you can pass the address to a** `COGINIT` **directive or** `COGNEW` **macro call.**

Not specifying the address allows you to start a new "cog" continuing on from where the last cog left off. In this case, you should have a line with a label immediately *after* the `COG` **directive. You can then use that label in combination with the @ operator with** `COGINIT` **and** `COGNEW`**. Note that when using** `COGNEW`**, the @ operator is added automatically for the code address, but is not for the PAR value.**

COGBASE *hub address in bytes*

Set the cog base address independently of the hub address. The `COGBASE` **directive will not change the current hub address, only symbol resolution.**

HUB *hub address in bytes*

Set the hub address independently of the cog base address. The `HUB` **directive will not change the current cog base address.**

ORG [*cog address in longs*]

Start assembling at the long at the specified address in cog memory. The cog base address is not affected. If the address is not specified it defaults to zero.

`ORG` **allows you to change the assembly address from the point of view of the cog without any reliance on the actual location in hub memory.**

FIT [*cog address in longs*]

`FIT` **checks if the current cog address is below the specified address and throws an assembly time error if it isn't. If the address is not specified, it defaults to $1F0.**

Examples

Minimal Example

A minimal functional example follows. It simply stops the current cog.

```
            cog         $20

Entry       cogid       __COGADDR
            cogstop     __COGADDR – 1
            fit
```

Overlapped Data

Overlapped data in this instance is the ability to have common data situated at the end of one cog and the beginning of another. This is a useful technique to minimize wasted memory when you have two cogs that need to share data. The data can be referred to by label in either cog 1 or cog 2 and will resolve to the correct address for that cog.

```
            ; Need include file for cognew macro
            include     "p8x32a.inc"

            cog         $20

            ; Start cog1 and cog2
            cognew      cog1, 0
            cognew      cog2, 0

            ; Stop cog 0
            cogid       __COGADDR
            cogstop     __COGADDR – 1


            ; Definition for cog 1
            cog
cog1
            ; ... code for cog 1...

            ; $1ed should be $1f0 – size of data in longs - 1
            fit         $1ed
            org         $1ed

            ; Entry point for cog 2
            ; jmp required to skip data when starting cog 2
cog2        jmp         #cogEntry

            ; overlapped data
overlap1    long        $deadbeef
overlap2    long        $baaabaaa

            ; Definition for cog 2
            cogbase     @cog2

cogEntry
            ; ... code for cog 2...
            fit
```

Setting the Clock Mode and Frequency

Although you can change the clock mode or frequency at run time if you wish, UCAsm also allows you to specify the default mode and frequency in the binary header. If you try and change the clock mode or frequency after it has been set, the assembler will throw a warning. The thinking behind this is that if you are changing the frequency at run time, you probably shouldn't be relying on the __XINFREQ and __CLKFREQ constants since they are set at assembly time.

CLKMODE ***clock mode byte***

Specify the clock mode. A number of constants are provided in p8x32a.inc for this purpose.

UCAsm Constant	Propeller Tool Equivalent
RCFAST	RCFAST
RCSLOW	RCSLOW
XINPUT	XINPUT
XTAL1_PLL1X	XTAL1 + PLL1X
XTAL1_PLL2X	XTAL1 + PLL2X
XTAL1_PLL4X	XTAL1 + PLL4X
XTAL1_PLL8X	XTAL1 + PLL8X
XTAL1_PLL16X	XTAL1 + PLL16X
XTAL1	XTAL1
XTAL2	XTAL2
XTAL3	XTAL3
XTAL2_PLL1X	XTAL2 + PLL1X
XTAL2_PLL2X	XTAL2 + PLL2X
XTAL2_PLL4X	XTAL2 + PLL4X
XTAL2_PLL8X	XTAL2 + PLL8X
XTAL2_PLL16X	XTAL2 + PLL16X
XINPUT_PLL1X	XINPUT + PLL1X
XINPUT_PLL2X	XINPUT + PLL2X
XINPUT_PLL4X	XINPUT + PLL4X
XINPUT_PLL8X	XINPUT + PLL8X
XINPUT_PLL16X	XINPUT + PLL16X
XTAL3_PLL1X	XTAL3 + PLL1X
XTAL3_PLL2X	XTAL3 + PLL2X
XTAL3_PLL4X	XTAL3 + PLL4X
XTAL3_PLL8X	XTAL3 + PLL8X
XTAL3_PLL16X	XTAL3 + PLL16X

The clock mode must be set before either `CLKFREQ` **or** `XINFREQ` **are used.** `CLKMODE` **will set the** `__CLKMODE` **constant to the value specified.**

CLKFREQ ***frequency in hertz***

XINFREQ ***frequency in hertz***

Set the system clock frequency or input frequency respectively. Only one of `CLKFREQ` **or** `XINFREQ` **needs to be specified, not both. The missing frequency will be calculated based on the clock mode. Both directives set the** `__CLKFREQ` **and** `__XINFREQ` **constants to the relevant value.**

Include Files

INCLUDE "*filename*"

Include the contents of another file. Inclusion is done during pass 1. The filename may be relative and will be checked against all directories in the include search paths. In the case that there are multiple files with the same name in different directories, UCAsm will use the first file it finds. See the next section for more information on the search paths.

Path separator characters in the filename will be automatically converted to the OS' native representation. For example, on Mac OS X and Linux backslashes will be converted to forward slashes and on Windows forward slashes will be converted to backslashes.

Include Search Paths

The assembler maintains a list of paths it searches for include files. The current directory and the system include directory are always added to the beginning and end of the search paths respectively. This allows files in the current directory to always override other directories, and paths specified on the command line to override the system include directory. Search paths can be specified on the command line with the –I option, and will be added in the order they are provided; the first having higher priority.

The system include directory is always a directory called Include in the same directory as the Bin directory, which contains the ucasm executable. This ensures that the include directory can be found without forcing the user to setup environment variables. If you do not want the system include directory to be included, use the –S command line option.

Expressions

NOTE: The expression parsing/evaluation code has already changed significantly for version 0.8.0.

UCAsm supports both constant and variable expressions in arguments for most instructions and directives. The expression evaluator is currently temporary and will be improved in a future version.

Note that all expressions are evaluated by the assembler at assembly time. The Parallax expression syntax is not supported.

Operator Precedence

Due to the work-in-progress nature of the expression evaluator, this operator precedence table should be considered a guideline that is subject to change.

< **>** **< =** **> =** **= =** **! =**	**Less than** **Greater than** **Less than or equal** **Greater than or equal** **Equal** **Not equal**
-	**Unary minus**
!	**Unary not**
( )	
***** **/** **&** **< <** **> >**	**Multiply** **Divide** **Bitwise and** **Bitwise shift left** **Bitwise shift right**
+ **-** **%** **|** **^**	**Add** **Subtract** **Modulus** **Bitwise or** **Exclusive or**

Constants and Variables

Assembly time constants and variables are supported. It is important to point out that assembly time constants and variables affect only the operation of the assembler. They are used in expressions and for controlling conditional assembly, and will be evaluated by the assembler. No code will be generated by these directives.

Constants are set with the `EQU` **directive, and may only be set once. A constant's value is resolved the first time it is used, and so forward references are supported.**

Variables are set with the `SET` **directive or the synonym** `=`**. A variable's value is resolved immediately during Pass 1 and so forward references are not supported. Variables are most useful in combination with** `WHILE` **loops.**

When porting code written for the Propeller Tool, you will come across a lot of constants set with the = operator. It is tempting to leave them the way they are. However, in UCAsm they will be treated as variables, not constants. The correct solution is to define all constants using the `EQU` **directive. This ensures that they will be handled correctly as constants, and will support forward references.**

constant* EQU *expression

The `EQU` **directive declares an assembly time constant. Forward references of constants are permissible. If you use variables in the expression, their value will be that at the time the expression is resolved. Constants are only resolved the first time they are used. It is advised that you don't use variables in constant expressions as this will probably throw an error in a future version.**

Recursive definitions are not permitted and will currently cause the assembler to go into an infinite loop.

variable* SET *expression

variable = expression

The `SET` **directive and it's more natural synonym set an assembly time variable. Variable expressions do not support forward references. The expression is evaluated whenever the line is processed. Usually this is after it's tokenized, but when used with conditional assembly it won't be processed until the block is expanded. With loops, the expression will be evaluated every iteration of the loop.**

The `=` **form of the** `SET` **directive is simply a convenient synonym. It is not the same as an assignment operator, and code using it still need to obey the usual formatting rules for the line. The main implication of this is that there must be whitespace either side of the =.**

Conditional Assembly

All conditional assembly directives must be terminated by the `END` directive. Unlike some other assemblers, in UCAsm there is only one `END` directive.

IF *expression*

If *expression* evaluates to true, the assembler will output the following block of assembly code. If *expression* evaluates to false, no code will be output.

IFDEF *symbol*

If *symbol* is defined, the assembler will output the following block of assembly code. If *symbol* is not defined, no code will be output. *symbol* can be any symbol known to the assembler: a label, constant, variable, or macro.

IFNDEF *symbol*

If *symbol* is not defined, the assembler will output the following block of assembly code. If *symbol* is defined, no code will be output. *symbol* can be any symbol known to the assembler: a label, constant, variable, or macro.

ELSE

If the matching `IF`, `IFDEF` or `IFNDEF` did not output any code, output the following block of assembly code. Otherwise, no code will be output.

WHILE *expression*

While *expression* evaluates to true, output the following block of assembly code. `WHILE` is usually used in conjunction with assembler variables, as in the following example.

```
i     =         0
      while     i < 5
      mov       0, i
i     =         i + 1
      end
```

This would be equivalent to writing the following code.

```
      mov   0, 0
      mov   0, 1
      mov   0, 2
      mov   0, 3
      mov   0, 4
```

Note that currently it is possible to write code using the `WHILE` directive that causes an infinite loop.

Examples

Conditional assembly can be useful for generating lookup tables at assembly time. The following code snippet generates 256 longs forming a CRC32 lookup table for each possible byte value. This table is generated entirely by the assembler and will not have any affect on run time performance.

```
crcTable

i               =               0
                while           i < 256
val                     =               i

j                       =               0
```

```
                        while           j < 8

                                if              val & $01
val                             =               0xedb88320 ^ (val >> 1)
                                else
val                             =               val >> 1
                                end

j                               =               j + 1
                        end


                ; Store data
                long            val

i               =               i + 1
                end
```

The CRC32 value for a byte stream can be calculated with the following C code:

```
for(int i = 0; i < strlen(buffer); i++)
{
      crcVal = crcTable[(crcVal ^ buffer[i]) & 0xff] ^ (crcVal >> 8);
}
```

A functioning CRC32 example can be found in the Examples folder.

Macros

Unlike other conditional assembly features, macros do not generate code immediately. A macro must be instantiated before any code will be generated. Macros may also optionally take any number of arguments. Arguments are implemented as a text substitution; wherever a macro argument name appears in the operand of a directive or instruction, it will be replaced with the text supplied as the argument.

```
macro MACRO [arg1][, arg2][, …][, argN]
```

Declare a macro with an optional number of arguments. All instructions, directives or data between the macro directive and the corresponding end directive will form the body of the macro.

```
LOCAL [label1][, label2][, …][, labelN]
```

Declare labels as local to a macro. LOCAL **is only valid in a macro block. When the macro is instantiated, macro local labels will be changed to a unique name. This avoids duplicate label definitions when instantiating a macro multiple times.**

Examples

```
; Macro Definition
stopCurCog        macro
                  cogid       $
                  cogstop     $ - 1
                  end

; Macro Instantiation
                  cog         $20
SomeLabel         cognew      SomeWhere, @Wherever

                  ; ... code ...

                  stopCurCog
```

The stopCurCog **macro is defined in p8x32a.inc and is a useful way to stop a cog. The macro uses self modifying code to obtain the current cog ID and then stop it.**

Also defined in p8x32a.inc is the cognew **macro.**

```
cognew            macro       codeAddr, parValue
                  local       cogWord

                  coginit     cogWord

                  jmp         #cogWord + 1
cogWord           long   ((parValue & $fffc) << 16) | ((@codeAddr & $fffc) << 2) | %1_000
                  end
```

cognew **uses a macro local label to ensure that there are no duplicate labels when it is instantiated multiple times. The word for the** COGINIT **instruction is built automatically based on the supplied arguments.**

Miscellaneous Directives

These directives don't really fit anywhere else, but are useful none the less.

ERROR "*error message*"

WARN "*warning message*"

Throw an assembly time error or warning message. The message occurs during pass 1. These directives can be useful when combined with conditional assembly. For example:

```
            if    __UCASM < 700
            error "UCAsm Version 0.7.0 is required to assemble this example"
            end
```

READONLY *cog address*

The READONLY **directive marks a cog register as read only. This is used in p8x32a.inc to specify which registers are read only. Attempting to use a read only register in the destination field of an instruction will cause an assembly time error.**

Assembler Defined Symbols

The assembler defines a number of constants automatically.

Symbol	Description
__UCASM	**Version number of UCAsm encoded as (Major * 1000) + (Revision * 100) + Minor**
__ADDR	**Hub address of the line being assembled**
__COGADDR $	**Cog address of the line being assembled. The $ symbol may be used as a synonym**
_CLKMODE	**Current clock mode word. Defined after the** CLKMODE **directive is processed**
_XINFREQ	**Current input frequency. Defined after either of the** XINFREQ **or** CLKFREQ **directives are processed**
_CLKFREQ	**Current system clock frequency. Defined after either of the** XINFREQ **or** CLKFREQ **directives are processed**

Compatibility Notes and Caveats

These are assorted notes that we assembled during development and early testing. The main aim is to cover potential issues for users used to the Propeller Tool and any other general gotchas we thought of.

- **UCAsm is case sensitive by default. You can pass the –N command line option to make symbols case insensitive. This is generally required when porting code from the Parallax Object Exchange. Note that currently macro arguments are always case sensitive regardless of the –N option.**
- **Instructions, directives, preconditions and effects are always case insensitive.**
- **Reserved words cannot be used as labels, including local labels. Reserved words are anything that is an instruction, directive, precondition or effect.**
- **The res directive always initializes to zero.**
- **The cog, hub and cogbase directives take a hub memory address in bytes. The org and fit directives take a cog memory address in longs. This is covered in more detail in the Assembler Basics chapter.**
- **There is currently no floating point support. This will be addressed in a future version.**
- **Single quotes (') are not supported as a comment character, use a semi-colon (;) instead. Converting Propeller Tool code is simply a matter of search and replace. We intend to use single quotes for character literals in a future build, and so this will not change.**
- **C++ style // comments are also supported. C style /* */ block comments are not supported, however. This combined with an editor that supports syntax highlighting can be abused to highlight C++ style comments in a different color to normal assembler comments. I use // comments to disable code which will be colored grey, whilst normal comments are colored green. I find this provides a useful visual distinction when skimming code.**
- **The end directive ends a block of conditional assembly; it does *NOT* end assembly like it may do in other assemblers. There is currently no directive in ucasm to halt assembly.**
- **Parallax expressions are not supported. Allowances have been made to support multiple expression evaluators, and so there is a possibility of Parallax expression support being added in the future. However, this is not currently a priority so it may be a while before it is done, if at all.**