gputils 0.11.8

James Bowman

October 19, 2003

Contents

Chapter 1

Introduction

gputils is a collection of tools for Microchip (TM) PIC microcontrollers. It includes gpasm, gplink, and gplib. Each tool is intended to be an open source replacement for a corresponding Microchip (TM) tool. This manual covers the basics of running the tools. For more details on a microcontroller, consult the manual for the specific PICmicro product that you are using.

This document is part of gputils.

1.1 Tool Flows

gputils can be used in two different ways, absolute mode and relocatable mode.

In absolute mode, a source file is directly converted into a hex file by gpasm. This method is absolute because the final addresses are hard coded into the source file.

In relocatable mode, the microcontroller source code is divided into seperate modules. Each module is assembled into an object using gpasm. That object can be placed “anywhere” in microcontroller’s memory. Then gplink is used to resolve symbols references, assign final address, and to patch the machine code with the final addresses. The output from gplink is an absolute executable object.

1.2 Which Tool Flow is best?

Absolute mode is simple to understand and to use. It only requires one tool, gpasm. Most of the examples on Microchip’s website use absolute mode. So why use relocatable mode?

- Code can be written without regard to addresses. This makes it easier to write and reuse.
- The objects can be archived to create a library, which also simplifies reuse.
- Recompiling a project can be faster, because you only compile the portions that have changed.
- Files can have local name spaces. The user chooses what symbols are global.

Most develpment tools use relocatable objects for these reasons. The few that don’t are generally microcontroller tools. Their applications are so small that absolute mode isn’t impractical. For PICs, relocatable mode has one big disadvantage. The bank and page control is a challenge.

Chapter 2

gpasm

2.1 Running gpasm

The general syntax for running gpasm is

```
gpasm [options] asm-file
```

Where options can be one of:

Option	Meaning
a <format>	Produce hex file in one of four formats: inhx8m, inhx8s, inhx16, inhx32 (the default).
c	Output a relocatable object
d symbol[=value]	Equivalent to "#define <symbol> <value>".
e [ON|OFF]	Expand macros in listing file.
h	Display the help message.
i	Ignore case in source code. By default gpasms to treats "fooYa" and "FOOYA" as being different.
I <directory>	Specify an include directory.
l	List the supported processors.
L	Ignore nolist directives.
m	Memory dump.
n	Use DOS style newlines (CRLF) in hex file. This option is disabled on win32 systems.
o <file>	Alternate name of hex output file.
p<processor>	Select target processor.
q	Quiet
r <radix>	Set the radix, i.e. the number base that gpasm uses when interpreting numbers. <radix> can be one of "oct", "dec" and "hex" for bases eight, ten, and sixteen respectively. Default is "hex".
w [0 | 1 | 2]	Set the message level.
v	Print gpasm version information and exit.

Unless otherwise specified, gpasm removes the ".asm" suffix from its input file, replacing it with ".lst" and ".hex" for the list and hex output files respectively. On most modern operating systems case is significant in filenames. For this reason you should ensure that filenames are named consistently, and that the ".asm" suffix on any source file is in lower case.

gpasm always produces a ".lst" file. If it runs without errors, it also produces a ".hex" file or a ".o" file.

2.1.1 Using gpasm with "make"

On most operating systems, you can build a project using the make utility. To use gpasm with make, you might have a "makefile" like this:

```
tree.hex: tree.asm treedef.inc
        gpasm tree.asm
```

This will rebuild "tree.hex" whenever either of the "tree.asm" or "treedef.inc" files change. A more comprehensive example of using gpasm with makefiles is included as example1 in the gpasm source distribution.

2.1.2 Dealing with errors

gpasm doesn't specifically create an error file. This can be a problem if you want to keep a record of errors, or if your assembly produces so many errors that they scroll off the screen. To deal with this if your shell is "sh", "bash" or "ksh", you can do something like:

```
gpasm tree.asm 2>&1 | tee tree.err
```

This redirects standard error to standard output ("2>&1"), then pipes this output into "tee", which copies it input to "tree.err", and then displays it.

2.2 Syntax

2.2.1 File structure

gpasm source files consist of a series of lines. Lines can contain a label (starting in column 1) or an operation (starting in any column after 1), both, or neither. Comments follow a ";" character, and are treated as a newline. Labels may be any series of the letters A-z, digits 0-9, and the underscore ("_"); they may not begin with a digit. Labels may be followed by a colon (":").

An operation is a single identifier (the same rules as for a label above) followed by a space, and a comma-separated list of parameters. For example, the following are all legal source lines:

```
                            ; Blank line
loop    sleep               ; Label and operation
        incf    6,1         ; Operation with 2 parameters
        goto    loop        ; Operation with 1 parameter
```

2.2.2 Expressions

gpasm supports a full set of operators, based on the C operator set. The operators in the following table are arranged in groups of equal precedence, but the groups are arranged in order of increasing precedence. When gpasm encounters operators of equal precedence, it always evaluates from left to right.

Operator	Description
=	assignment
||	logical or
&&	logical and
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<	less than
>	greater than
==	equals
!=	not equals
>=	greater than or equal
<=	less than or equal
<<	left shift
>>	right shift
+	addition
-	subtraction
*	multiplication
/	division
%	modulo
HIGH	high byte
LOW	low byte
-	negation
!	logical not
~	bitwise no

Any symbol appearing in column 1 may be assigned a value using the assignment operator (=) in the previous table. Additionally, any value previously assigned may be modified using one of the operators in the table below. Each of these operators evaluates the current value of the symbol and then assigns a new value based on the operator.

Operator	Description
=	assignment
++	increment by 1
–	decrement by 1
+=	increment
-=	decrement
*=	multiply
/=	divide
%=	modulo
<<=	left shift
>>=	right shift
&=	bitwise and
|=	bitwise or
^=	bitwise exclusive-or

2.2.3 Numbers

gpasm gives you several ways of specifying numbers. You can use a syntax that uses an initial character to indicate the number's base. The following table summarizes the alternatives. Note the C-style option for specifying hexadecimal numbers.

base	general syntax	21 decimal written as
binary	B'[01]*'	B'10101'
octal	O'[0-7]*'	O'25'
decimal	D'[0-9]*'	D'21'
hex	H'[0-F]*'	H'15'
hex	0x[0-F]*	0x15

When you write a number without a specifying prefix such as "45", gpasm uses the current radix (base) to interpret the number. You can change this radix with the RADIX directive, or with the "-r" option on gpasm's command-line. If you do not start hexadecimal numbers with a digit, gpasm will attempt to interpret what you've written as an identifier. For example, instead of writing C2, write either 0C2, 0xC2 or H'C2'.

Case is not significant when interpreting numbers: 0ca, 0CA, h'CA' and H'ca' are all equivalent.

Several legacy mpasm number formats are also supported. These formats have various shortcomings, but are still supported. The table below summarizes them.

base	general syntax	21 decimal written as
binary	[01]*b	10101b
octal	q'[0-7]*'	q'25'
octal	[0-7]*o	25o
octal	[0-7]*q	25q
decimal	0-9]*d	21d
decimal	.[0-9]*	.21
hex	[0-F]*h	15h

You can write the ASCII code for a character X using 'X', or A'X'.

2.2.4 Preprocessor

A line such as:

```
include foo.inc
```

will make gpasm fetch source lines from the file "foo.inc" until the end of the file, and then return to the original source file at the line following the include.

Lines beginning with a "#" are preprocessor directives, and are treated differently by gpasm. They may contain a "#define", or a "#undefine" directive.

Once gpasm has processed a line such as:

```
#define X Y
```

every subsequent occurrence of X is replaced with Y, until the end of file or a line

```
#undefine X
```

appears.

The preprocessor will replace an occurance of #v(expression) in a symbol with the value of "expression" in decimal. In the following expression:

```
number equ 5
label_#v( (number +1) * 5 )_suffix equ 0x10
```

gpasm will place the symbol "label_30_suffix" with a value of 0x10 in the symbol table.

The preprocessor in gpasm is only *like* the C preprocessor; its syntax is rather different from that of the C preprocessor. gpasm uses a simple internal preprocessor to implement "include", "#define" and "#undefine".

2.2.5 Processor header files

gputils distributes the Microchip processor header files. These files contain processor specific data that is helpful in developing PIC applications. The location of these files is reported in the gpasm help message. Use the INCLUDE directive to utilize the appropriate file in your source code. Only the name of the file is required. gpasm will search the default path automatically.

2.3 Directives

2.3.1 Code generation

In absolute mode, use the ORG directive to set the PIC memory location where gpasm will start assembling code. If you don't specify an address with ORG, gpasm assumes 0x0000. In relocatable mode, use the CODE directive.

2.3.2 Configuration

You can choose the fuse settings for your PIC implementation using the __CONFIG directive, so that the hex file set the fuses explicitly. Naturally you should make sure that these settings match your PIC hardware design.

The __MAXRAM and __BADRAM directives specify which RAM locations are legal. These directives are mostly used in processor-specific configuration files.

2.3.3 Conditional assembly

The IF, IFNDEF, IFDEF, ELSE and ENDIF directives enable you to assemble certain sections of code only if a condition is met. In themselves, they do not cause gpasm to emit any PIC code. The example in section 2.3.4 for demonstrates conditional assembly.

2.3.4 Macros

gpasm supports a simple macro scheme; you can define and use macros like this:

```
any     macro parm
        movlw parm
        endm
...
        any   33
```

A more useful example of some macros in use is:

```
; Shift reg left, result (w or f) in 'dst'
slf     macro   reg,dst
        clrc
        rlf     reg,f
endm

; Scale W by "factor".  Result in "reg", W unchanged.
scale   macro   reg, factor
        if (factor == 1)
            movwf reg                ; 1 X is easy
        else
            scale   reg, (factor / 2) ; W * (factor / 2)
            slf     reg,f             ; double reg
            if ((factor & 1) == 1)    ; if lo-bit set ..
                addwf   reg,f         ; .. add W to reg
```

```
            endif
        endif
endm
```

This recursive macro generates code to multiply W by a constant “factor”, and stores the result in “reg”. So writing:

```
        scale   tmp,D'10'
```

is the same as writing:

```
        movwf   tmp      ; tmp = W
        clrc
        rlf     tmp,f    ; tmp = 2 * W
        clrc
        rlf     tmp,f    ; tmp = 4 * W
        addwf   tmp,f    ; tmp = (4 * W) + W = 5 * W
        clrc
        rlf     tmp,f    ; tmp = 10 * W
```

2.3.5 $

$ expands to the address of the instruction currently being assembled. If it’s used in a context other than an instruction, such as a conditional, it expands to the address the next instruction would occupy, since the assembler’s idea of current address is incremented after an instruction is assembled. $ may be manipulated just like any other number:

```
$
$ + 1
$ - 2
```

and can be used as a shortcut for writing loops without labels.

```
LOOP:  BTFSS flag,0x00
       GOTO LOOP
       BTFSS flag,0x00
       GOTO  $ - 1
```

2.3.6 Suggestions for structuring your code

Nested IF operations can quickly become confusing. Indentation is one way of making code clearer. Another way is to add braces on IF, ELSE and ENDIF, like this:

```
        IF (this) ; {
          ...
        ELSE      ; }{
          ...
        ENDIF     ; }
```

After you’ve done this, you can use your text editor’s show-matching-brace to check matching parts of the IF structure. In vi this command is “%”, in emacs it’s M-C-f and M-C-b.

2.3.7 Directive summary

__BADRAM

```
__BADRAM <expression> [, <expression]*
```

Instructs gpasm that it should generate an error if there is any use of the given RAM locations. Specify a range of addresses with <lo>-<hi>. See any processor-specific header file for an example.

See also: __MAXRAM

__CONFIG

```
__CONFIG <expression>
```

Sets the PIC processor's configuration fuses.

__IDLOCS

```
__IDLOCS <expression> or __IDLOCS <expression1>,<expression2>
```

Sets the PIC processor's identification locations. For 12 and 14 bit processors, the four id locations are set to the hexadecimal value of expression. For 18cxx devices idlocation expression1 is set to the hexadecimal value of expression2.

__MAXRAM

```
__MAXRAM <expression>
```

Instructs gpasm that an attempt to use any RAM location above the one specified should be treated as an error. See any processor specific header file for an example.

See also: __BADRAM

BANKISEL

```
BANKISEL <label>
```

This directive generates bank selecting code for indirect access of the address specified by <label>. The directive is not available for all devices. It is only available for 14 bit and 16 bit devices. For 14 bit devices, the bank selecting code will set/clear the IRP bit of the STATUS register. It will use MOVLB or MOVLR in 16 bit devices.

See also: BANKSEL, PAGESEL

BANKSEL

```
BANKSEL <label>
```

This directive generates bank selecting code to set the bank to the bank containing <label>. The bank selecting code will set/clear bits in the FSR for 12 bit devices. It will set/clear bits in the STATUS register for 14 bit devices. It will use MOVLB or MOVLR in 16 bit devices. MOVLB will be used for enhanced 16 bit devices.

See also: BANKISEL, PAGESEL

CBLOCK

```
CBLOCK [<expression>]
       <label>[:<increment>][,<label>[:<increment>]]
ENDC
```

Marks the beginning of a block of constants <label>. gpasm allocates values for symbols in the block starting at the value <expression> given to CBLOCK. An optional <increment> value leaves space after the <label> before the next <label>.

See also: EQU

CODE

```
<label> CODE <expression>
```

Only for relocatable mode. Creates a new machine code section in the output object file. <label> specifies the name of the section. If <label> is not specified the default name ".code" will be used. <expression> is optional and specifies the absolute address of the section.

See also: IDATA, UDATA

CONSTANT

```
CONSTANT <label>=<expression> [, <label>=<expression>]*
```

Permanently assigns the value obtained by evaluating <expression> to thesymbol <label>. Similar to SET and VARIABLE, except it can not be changed once assigned.

See also: EQU, SET, VARIABLE

DA

```
<label> DA <expression> [, <expression]*
```

Stores Strings in program memory. The data is stored as one 14 bit word representing two 7 bit ASCII characters.

See also: DT

DATA

```
DATA <expression> [, <expression]*
```

Generates the specified data.

See also: DA, DB, DE, DW

DB

```
<label> DB <expression> [, <expression]*
```

Declare data of one byte. The values are packed two per word.

See also: DA, DATA, DE, DW

DE

```
<label> DE <expression> [, <expression]*
```

Define EEPROM data. Each character in a string is stored in a separate word.

See also: DA, DATA, DB, DW

DT

```
DT <expression> [, <expression]*
```

Generates the specified data as bytes in a sequence of RETLW instructions.

See also: DATA

DW

```
<label> DW <expression> [, <expression]*
```

Declare data of one word.

See also: DA, DATA, DB, DW

ELSE

```
ELSE
```

Marks the alternate section of a conditional assembly block.
See also: IF, IFDEF, IFNDEF, ELSE, ENDIF

END

```
END
```

Marks the end of the source file.

ENDC

```
ENDC
```

Marks the end of a CBLOCK.
See also: CBLOCK

ENDIF

```
ENDIF
```

Ends a conditional assembly block.
See also: IFDEF, IFNDEF, ELSE, ENDIF

ENDM

```
ENDM
```

Ends a macro definition.
See also: MACRO

ENDW

```
ENDW
```

Ends a while loop.
See also: WHILE

EQU

```
<label> EQU <expression>
```

Permanently assigns the value obtained by evaluating <expression> to the symbol <label>. Similar to SET and VARIABLE, except it can not be changed once assigned.
See also: CONSTANT, SET

ERROR

```
ERROR <string>
```

Issues an error message.
See also: MESSG

ERRORLEVEL

```
ERRORLEVEL {0 | 1 | 2 | +<msgnum> | -<msgnum>}[, ...]
```

Sets the types of messages that are printed.

Setting	Affect
0	Messages, warnings and errors printed.
1	Warnings and error printed.
2	Errors printed.
-<msgnum>	Inhibits the printing of message <msgnum>.
+<msgnum>	Enables the printing of message <msgnum>.

See also: LIST

EXTERN

```
EXTERN <symbol> [ , <symbol> ]*
```

Only for relocatable mode. Delcare a new symbol that is defined in another object file.
See also: GLOBAL

EXITM

```
EXITM
```

Immediately return from macro expansion during assembly.
See also: ENDM

EXPAND

```
EXPAND
```

Expand the macro in the listing file.
See also: ENDM

FILL

```
<label> FILL <expression>,<count>
```

Generates <count> occurrences of the program word or byte <expression>. If expression is enclosed by parentheses, expression is a line of assembly.
See also: DATA DW ORG

GLOBAL

```
GLOBAL <symbol> [ , <symbol> ]*
```

Only for relocatable mode. Delcare a symbol as global.
See also: GLOBAL

IDATA

```
<label> IDATA <expression>
```

Only for relocatable mode. Creates a new initialized data section in the output object file. <label> specifies the name of the section. If <label> is not specified the default name ".idata" will be used. <expression> is optional and specifies the absolute address of the section. Data memory is allocated and the initialization data is placed in ROM. The user must provide the code to load the data into memory.
See also: CODE, UDATA

IF

```
IF <expression>
```

Begin a conditional assembly block. If the value obtained by evaluating <expression> is true (i.e. non-zero), code up to the following ELSE or ENDIF is assembled. If the value is false (i.e. zero), code is not assembled until the corresponding ELSE or ENDIF.

See also: IFDEF, IFNDEF, ELSE, ENDIF

IFDEF

```
IFDEF <symbol>
```

Begin a conditional assembly block. If <symbol> appears in the symbol table, gpasm assembles the following code.

See also: IF, IFNDEF, ELSE, ENDIF

IFNDEF

```
IFNDEF <symbol>
```

Begin a conditional assembly block. If <symbol>does not appear in the symbol table, gpasm assembles the following code.

See also: IF, IFNDEF, ELSE, ENDIF

LIST

```
LIST <expression> [ , <expression> ] *
```

Enables output to the list (".lst") file. All arguments are interpreted as decimal regardless of the current radix setting. "list n=0" may be used to prevent page breaks in the code section of the list file. Other options are listed in the table below:

option	description
b=nnn	Sets the tab spaces
f=<format>	Set the hex file format. Can be inhx8m, inhx8s, inhx16, or inhx32.
mm=[ON|OFF]	Memory Map on or off
n=nnn	Sets the number of lines per page
p = <symbol>	Sets the current processor
r= [oct | dec | hex]	Sets the radix
st = [ON | OFF]	Symbol table dump on or off
w=[0 | 1| 2]	Sets the message level.
x=[ON|OFF]	Macro expansion on or off

See also: NOLIST, RADIX, PROCESSOR

LOCAL

```
LOCAL <symbol>[[=<expression>], [<symbol>[=<expression>]]*]
```

Declares <symbol> as local to the macro that's currently being defined. This means that further occurrences of <symbol> in the macro definition refer to a local variable, with scope and lifetime limited to the execution of the macro.

See also: MACRO, ENDM

MACRO

```
<label> MACRO [ <symbol> [ , <symbol> ]* ]
```

Declares a macro with name <label>. gpasm replaces any occurrences of <symbol> in the macro definition with the parameters given at macro invocation.
See also: LOCAL, ENDM

MESSG

```
MESSG <string>
```

Writes <string> to the list file, and to the standard error output.
See also: ERROR

NOEXPAND

```
NOEXPAND
```

Turn off macro expansion in the list file.
See also: EXPAND

NOLIST

```
NOLIST
```

Disables list file output.
See also: LIST

ORG

```
ORG <expression>
```

Sets the location at which instructions will be placed. If the source file does not specify an address with ORG, gpasm assumes an ORG of zero.

PAGE

```
PAGE
```

Causes the list file to advance to the next page.
See also: LIST

PAGESEL

```
PAGESEL <label>
GOTO <label>
```

This directive will generate page selecting code to set the page bits to the page containing the designated <label>. The page selecting code will set/clear bits in the STATUS for 12 bit devices. For 14 bit and 16 bit devices, it will generate MOVLW and MOVWF to modify PCLATH. The directive is ignored for enhanced 16 bit devices.
See also: BANKISEL, BANKSEL

PROCESSOR

```
PROCESSOR <symbol>
```

Selects the target processor. See section **??** for more details.
See also: LIST

RADIX

```
RADIX <symbol>
```

Selects the default radix from “oct” for octal, “dec” for decimal or “hex” for hexadecimal. gpasm uses this radix to interpret numbers that don’t have an explicit radix.
See also: LIST

RES

```
RES <mem_units>
```

Causes the memory location pointer to be advanced <mem_units>. Can be used to reserve data storage.
See also: FILL, ORG

SET

```
<label> SET <expression>
```

Temporarily assigns the value obtained by evaluating <expression> to the symbol <label>.
See also: SET

SPACE

```
SPACE <expression>
```

Inserts <expression> number of blank lines into the listing file.
See also: LIST

SUBTITLE

```
SUBTITLE <string>
```

This directive establishes a second program header line for use as a subtitle in the listing output. <string> is an ASCII string enclosed by double quotes, no longer than 60 characters.
See also: TITLE

TITLE

```
TITLE <string>
```

This directive establishes a program header line for use as a title in the listing output. <string> is an ASCII string enclosed by double quotes, no longer than 60 characters.
See also: SUBTITLE

UDATA

```
<label> UDATA <expression>
```

Only for relocatable mode. Creates a new uninitialized data section in the output object file. <label> specifies the name of the section. If <label> is not specified the default name “.udata” will be used. <expression> is optional and specifies the absolute address of the section.
See also: CODE, IDATA, UDATA_ACS, UDATA_OVR, UDATA_SHR

UDATA_ACS

```
<label> UDATA_ACS <expression>
```

Only for relocatable mode. Creates a new uninitialized accessbank data section in the output object file. <label> specifies the name of the section. If <label> is not specified the default name ".udata_acs" will be used. <expression> is optional and specifies the absolute address of the section.

See also: CODE, IDATA, UDATA

UDATA_OVR

```
<label> UDATA_OVR <expression>
```

Only for relocatable mode. Creates a new uninitialized overlaid data section in the output object file. <label> specifies the name of the section. If <label> is not specified the default name ".udata_ovr" will be used. <expression> is optional and specifies the absolute address of the section.

See also: CODE, IDATA, UDATA

UDATA_SHR

```
<label> UDATA_SHR <expression>
```

Only for relocatable mode. Creates a new uninitialized sharebank data section in the output object file. <label> specifies the name of the section. If <label> is not specified the default name ".udata_shr" will be used. <expression> is optional and specifies the absolute address of the section.

See also: CODE, IDATA, UDATA

VARIABLE

```
VARIABLE <label>[=<expression>, <label>[=<expression>]]*
```

Delcares variable with the name <label>. The value of <label> may later be reassigned. The value of <label> does not have to be assigned at declaration.

See also: CONSTANT

WHILE

```
WHILE <expression>
```

Performs loop while <expression> is true.

See also: ENDW

2.4 Instructions

2.4.1 Supported processors

```
gpasm currently supports the following processors:
eeprom8    gen        p12c508    p12c508a   p12c509    p12c509a
p12c671    p12c672    p12ce518   p12ce519   p12ce673   p12ce674
p12cr509a  p12f629    p12f675    p14000     p16c5x     p16cxx
p16c432    p16c433    p16c505    p16c52     p16c54     p16c54a
p16c54b    p16c54c    p16c55     p16c55a    p16c554    p16c557
p16c558    p16c56     p16c56a    p16c57     p16c57c    p16c58a
p16c58b    p16c61     p16c62     p16c62a    p16c62b    p16c620
p16c620a   p16c621    p16c621a   p16c622    p16c622a   p16c63
p16c63a    p16c64     p16c64a    p16c642    p16c65     p16c65a
```

```
p16c65b    p16c66     p16c662    p16c67     p16c71     p16c710
p16c711    p16c712    p16c715    p16c716    p16c717    p16c72
p16c72a    p16c73     p16c73a    p16c73b    p16c74     p16c745
p16c747    p16c74a    p16c74b    p16c76     p16c765    p16c77
p16c770    p16c771    p16c773    p16c774    p16c781    p16c782
p16c84     p16c923    p16c924    p16c925    p16c926    p16ce623
p16ce624   p16ce625   p16cr54    p16cr54a   p16cr54b   p16cr54c
p16cr56a   p16cr57a   p16cr57b   p16cr57c   p16cr58a   p16cr58b
p16cr62    p16cr620a  p16cr63    p16cr64    p16cr65    p16cr72
p16cr83    p16cr84    p16f627    p16f627a   p16f628    p16f628a
p16f630    p16f648a   p16f676    p16f684    p16f716    p16f72
p16f73     p16f737    p16f74     p16f76     p16f767    p16f77
p16f777    p16f818    p16f819    p16f83     p16f84     p16f84a
p16f87     p16f870    p16f871    p16f872    p16f873    p16f873a
p16f874    p16f874a   p16f876    p16f876a   p16f877    p16f877a
p16f88     p16hv540   p17cxx     p17c42     p17c42a    p17c43
p17c44     p17c752    p17c756    p17c756a   p17c762    p17c766
p17cr42    p17cr43    p18cxx     p18c242    p18c252    p18c442
p18c452    p18c601    p18c658    p18c801    p18c858    p18f1220
p18f1320   p18f2220   p18f2320   p18f2331   p18f242    p18f2431
p18f2439   p18f248    p18f252    p18f2539   p18f258    p18f2620
p18f4220   p18f4320   p18f4331   p18f442    p18f4431   p18f4439
p18f448    p18f452    p18f4539   p18f458    p18f4620   p18f6520
p18f6525   p18f6585   p18f6620   p18f6621   p18f6680   p18f6720
p18f8520   p18f8525   p18f8585   p18f8620   p18f8621   p18f8680
p18f8720   rf509af    rf509ag    rf675f     rf675h     rf675k
sx18       sx20       sx28
```

2.4.2 Instruction set summary

12 bit Devices (PIC12C5XX)

Syntax	Description
ADDWF <f>,<dst>	Add W to <f>, result in <dst>
ANDLW <f>,<dst>	And W and literal, result in W
ANDWF <f>,<dst>	And W and <f>, result in <dst>
BCF <f>,<bit>	Clear <bit> of <f>
BSF <f>,<bit>	Set <bit> of <f>
BTFSC <f>,<bit>	Skip next instruction if <bit> of <f> is clear
BTFSS <f>,<bit>	Skip next instruction if <bit> of <f> is set
CALL <addr>	Call subroutine
CLRF <f>,<dst>	Write zero to <dst>
CLRW	Write zero to W
CLRWDT	Reset watchdog timer
COMF <f>,<dst>	Complement <f>, result in <dst>
DECF <f>,<dst>	Decrement <f>, result in <dst>
DECFSZ <f>,<dst>	Decrement <f>, result in <dst>, skip if zero
GOTO <addr>	Go to <addr>
INCF <f>,<dst>	Increment <f>, result in <dst>
INCFSZ <f>,<dst>	Increment <f>, result in <dst>, skip if zero
IORLW <f>,<dst>	Or W and <f>, result in <dst>
MOVF <f>,<dst>	Move <f> to <dst>
MOVLW <imm8>	Move literal to W
MOVWF <f>	Move W to <f>
NOP	No operation
OPTION	
RETLW <imm8>	Load W with immediate and return
RLF <f>,<dst>	Rotate <f> left, result in <dst>
RRF <f>,<dst>	Rotate <f> right, result in <dst>
SLEEP	Enter sleep mode
SUBWF <f>,<dst>	Subtract W from <f>, result in <dst>
SWAPF <f>,<dst>	Swap nibbles of <f>, result in <dst>
TRIS	
XORLW	Xor W and <f>, result in <dst>
XORWF	Xor W and immediate

14 Bit Devices (PIC16CXX)

Syntax	Description
ADDLW <imm8>	Add immediate to W
ADDWF <f>,<dst>	Add W to <f>, result in <dst>
ANDLW <f>,<dst>	And W and <f>, result in <dst>
BCF <f>,<bit>	Clear <bit> of <f>
BSF <f>,<bit>	Set <bit> of <f>
BTFSC <f>,<bit>	Skip next instruction if <bit> of <f> is clear
BTFSS <f>,<bit>	Skip next instruction if <bit> of <f> is set
CALL <addr>	Call subroutine
CLRF <f>,<dst>	Write zero to <dst>
CLRW	Write zero to W
CLRWDT	Reset watchdog timer
COMF <f>,<dst>	Complement <f>, result in <dst>
DECF <f>,<dst>	Decrement <f>, result in <dst>
DECFSZ <f>,<dst>	Decrement <f>, result in <dst>, skip if zero
GOTO <addr>	Go to <addr>
INCF <f>,<dst>	Increment <f>, result in <dst>
INCFSZ <f>,<dst>	Increment <f>, result in <dst>, skip if zero
IORLW <f>,<dst>	Or W and <f>, result in <dst>
MOVF <f>,<dst>	Move <f> to <dst>
MOVLW <imm8>	Move literal to W
MOVWF <f>	Move W to <f>
NOP	No operation
OPTION	
RETFIE	Return from interrupt
RETLW <imm8>	Load W with immediate and return
RETURN	Return from subroutine
RLF <f>,<dst>	Rotate <f> left, result in <dst>
RRF <f>,<dst>	Rotate <f> right, result in <dst>
SLEEP	Enter sleep mode
SUBLW	Subtract W from literal
SUBWF <f>,<dst>	Subtract W from <f>, result in <dst>
SWAPF <f>,<dst>	Swap nibbles of <f>, result in <dst>
TRIS	
XORLW	Xor W and <f>, result in <dst>
XORWF	Xor W and immediate

Ubicom Processors

For Ubicom (Scenix) processors, the assembler supports the following instructions, in addition to those listed under “12 Bit Devices” above.

Syntax	Description
BANK <imm3>	
IREAD	
MODE <imm4>	
MOVMW	
MOVWM	
PAGE <imm3>	
RETI	
RETIW	
RETP	
RETURN	

Special macros

There are also a number of standard additional macros. These macros are:

Syntax	Description
ADDCF <f>,<dst>	Add carry to <f>, result in <dst>
B <addr>	Branch
BC <addr>	Branch on carry
BZ <addr>	Branch on zero
BNC <addr>	Branch on no carry
BNZ <addr>	Branch on not zero
CLRC	Clear carry
CLRZ	Clear zero
SETC	Set carry
SETZ	Set zero
MOVFW <f>	Move file to W
NEGF <f>	Negate <f>
SKPC	Skip on carry
SKPZ	Skip on zero
SKPNC	Skip on no carry
SKPNZ	Skip on not zero
SUBCF <f>,<dst>	Subtract carry from <f>, result in <dst>
TSTF <f>	Test <f>

2.5 Errors/Warnings/Messages

gpasm writes every error message to two locations:

- the standard error output
- the list file (".lst")

The format of error messages is:

```
Error <src-file> <line> : <code> <description>
```

where:

<src-file> is the source file where gpasm encountered the error

<line> is the line number

<code> is the 3-digit code for the error, given in the list below

<description> is a short description of the error. In some cases this contains further information about the error.

Error messages are suitable for parsing by emacs' "compilation mode". This chapter lists the error messages that gpasm produces.

2.5.1 Errors

101 ERROR directive

A user-generated error. See the ERROR directive for more details.

114 Divide by zero

gpasm encountered a divide by zero.

115 Duplicate Label

Duplicate label or redefining a symbol that can not be redefined.

124 Illegal Argument

gpasm encountered an illegal argument in an expression.

125 Illegal Condition

An illegal condition like a missing ENDIF or ENDW has been encountered.

126 Argument out of Range

The expression has an argument that was out of range.

127 Too many arguments

gpasm encountered an expression with too many arguments.

128 Missing argument(s)

gpasm encountered an expression with at least one missing argument.

129 Expected

Expected a certain type of argument.

130 Processor type previously defined

The processor is being redefined.

131 Undefined processor

The processor type has not been defined.

132 Unknown processor

The selected processor is not valid. Check the processors listed in section **??**.

133 Hex file format INHX32 required

An address above 32K was specified.

135 Macro name missing

A macro was defined without a name.

136 Duplicate macro name

A macro name was duplicated.

145 Unmatched ENDM

ENDM found without a macro definition.

159 Odd number of FILL bytes

In PIC18CXX devices the number of bytes must be even.

2.5.2 Warnings

201 Symbol not previously defined.

The symbol being #undefined was not previously defined.

202 Argument out of range

The argument does not fit in the allocated space.

211 Extraneous arguments

Extra arguments were found on the line.

215 Processor superseded by command line

The processor was specified on the command line and in the source file. The command line has precedence.

216 Radix superseded by command line

The radix was specified on the command line and in the source file. The command line has precedence.

217 Hex format superseded by command line

The hex file format was specified on the command line and in the source file. The command line has precedence.

218 Expected DEC, OCT, HEX. Will use HEX.

gpasm encountered an invalid radix.

219 Invalid RAM location specified.

gpasm encountered an invalid RAM location as specified by the __MAXRAM and __BADRAM directives.

222 Error messages can not be disabled

Error messages can not be disabled using the ERRORLEVEL directive.

223 Redefining processor

The processor is being reselected by the LIST or PROCESSOR directive.

224 Use of this instruction is not recommanded

Use of the TRIS and OPTION instructions is not recommended for a PIC16CXX device.

2.5.3 Messages

301 User Message

User message, invoked with the MESSG directive.

303 Program word too large. Truncated to core size.

gpasm has encounter a program word larger than the core size of the selected device.

304 ID Locations value too large. Last four hex digits used.

The ID locations value specified is too large.

305 Using default destination of 1 (file).

No destination was specified so the default location was used.

308 Warning level superseded by command line

The warning level was specified on the command line and in the source file. The command line has precedence.

309 Macro expansion superseded by command line

Macro expansion was specified on the command line and in the source file. The command line has precedence.

Chapter 3

gplink

gplink relocates and links gpasm COFF objects and generates an absolute executable COFF.

3.1 Running gplink

The general syntax for running gplink is

```
gplink [options] [objects] [libraries]
```

Where options can be one of:

Option	Meaning
a	Produce hex file in one of four formats: inhx8m, inhx8s, inhx16, inhx32 (the default).
c	Output an executable object.
d	Display debug messages
f <value>	Fill unused unprotected program memory with <value>.
h	Show the help message
I <directory>	Specify an include directory.
m	Output a map file.
o <file>	Alternate name of hex output file.
q	Quiet.
s <file>	Specify linker script.
v	Print gplib version information and exit

3.2 gplink outputs

gplink creates an absolute executable COFF. From this COFF a hex file and cod file are created. The executable COFF is only written when the "-c" option is added. This file is useful for simulating the design with mpsim. The cod file is used for simulating with gpsim.

gplink can also create a map file. The map file reports the final addresses gplink has assigned to the COFF sections. This is the same data that can be viewed in the executable COFF with gpvo.

3.3 Linker scripts

gplink requires a linker script. This script tells gplink what memory is available in the target processor. A set of Microchip generated scripts are installed with gputils. These scripts were intended as a starting point, but for many applications they will work as is.

If the user does not specify a linker script, gplink will attempt to use the default script for the processor reported in the object file. The default location of the scripts is reported in the gplink help message.

Chapter 4

gplib

gplib creates, modifies and extracts COFF archives. This allows a related group of objects to be combined into one file. Then this one file is passed to gplink.

4.1 Running gplib

The general syntax for running gplib is

```
gplib [options] library [member]
```

Where options can be one of:

Option	Meaning
c	Create a new library
d	Delete member from library
h	Show the help message
n	Don't add the symbol index
q	Quiet mode.
r	Add or replace member from library.
s	List global symbols in libary.
t	List member in library
v	Print gplib version information and exit
x	Extract member from library

4.2 Creating an archive

The most common operation is to create a new archive:

```
gplib -c math.a mult.o add.o sub.o
```

This command will create a new archive "math.a" that contains "mult.o add.o sub.o".

The name of the archive "math.a" is arbitrary. The tools do not use the file extension to determine file type. It could just as easily been "math.lib" or "math".

When you use the library, simply add it to the list of object passed to gplink. gplink will scan the library and only extract the archive members that are required to resolve external references. So the application won't necessarily contain the code of all the archive members.

4.3 Other gplib operations

Most of the other are useful , but will be used much less often. For example you can replace individual archive members, but most people elect to delete the old archive and create a new one.

4.4 Archive format

The file format is a standard COFF archive. A header is added to each member and the unmodified object is copied into the archive.

Being a standard archive they do include a symbol index. It provides a simple why to determine which member should be extract to resolve external references. This index is not included in mplib archives. So using gplib archives with Microchip Tools will probably cause problems unles the “-n” option is added when the archive is created.

Chapter 5

Utilities

5.1 gpdasm

gpdasm is a disassembler for gputils. It converts hex files generated by gpasm and gplink into disassembled instructions.

5.1.1 Running gpdasm

The general syntax for running gpdasm is

```
gpdasm [options] hex-file
```

Where options can be one of:

Option	Meaning
h	Display the help message.
i	Display hex file information
l	List supported processors.
m	Memory dump hex file.
p<processor>	Select processor.
s	Print short form output
v	Print gpasm version information and exit.

gpdasm doesn't specifically create an output file. It dumps its output to the screen. This helps to reduce the risk that a good source file will be unintentionally overwritten. If you want to create an output file and your shell is "sh", "bash" or "ksh", you can do something like:

```
gpdasm test.hex > test.dis
```

This redirects standard output to the file "test.dis".

5.1.2 Comments on Disassembling

- The gpdasm only uses a hex file as an input. Because of this it has no way to distinguish between instructions and data in program memory.
- If gpdasm encounters an unknown instruction it uses the DW directive and treats it as raw data.
- There are DON'T CARE bits in the instruction words. Normally, this isn't a problem. It could be, however, if a file with data in the program memory space is disassembled and then reassembled. Example: gpdasm will treat 0x0060 in a 14 bit device as a NOP. If the output is then reassembled, gpasm will assign a 0x0000 value. The value has changed and both tools are behaving correctly.

5.2 gpvc

gpvc is cod file viewer for gputils. It provides an easy way to view the contents of the cod files generated by gpasm and gplink.

5.2.1 Running gpvc

The general syntax for running gpvc is

```
gpvc [options] cod-file
```

Where options can be one of:

Option	Meaning
a	Display all information
d	Display directory header
s	Display symbols
h	Show the help message.
r	Display ROM
l	Display source listing
m	Display debug message area
v	Print gpvc version information and exit.

gpvc doesn't specifically create an output file. It dumps its output to the screen. If you want to create an output file and your shell is "sh", "bash" or "ksh", you can do something like:

```
gpvc test.cod > test.dump
```

This redirects standard output to the file "test.dump".

5.3 gpvo

gpvo is COFF object file viewer for gputils. It provides an easy way to view the contents of objects generated by gpasm and gplink.

5.3.1 Running gpvo

The general syntax for running gpvo is

```
gpvo [options] object-file
```

Where options can be one of:

Option	Meaning
b	Binary data
f	File header
h	Show the help message
n	Suppress filenames
s	Section data
t	Symbol data
v	Print gpvo version information and exit

gpvo doesn't specifically create an output file. It dumps its output to the screen. If you want to create an output file and your shell is "sh", "bash" or "ksh", you can do something like:

```
gpvo test.obj > test.dump
```

This redirects standard output to the file "test.dump".

Index