gpasm 0.9.1

James Bowman

February 11 2001

Contents

Introduction

gpasm is meant to be an open source replacement for the Microchip (TM) product MPASM, an assembler for Microchip's popular PICmicro (TM) line of microcontrollers. This manual covers the basics of running gpasm: for more details on a microprocessor, consult the manual for the specific PICmicro product that you are using.

This document is part of gpasm.

Chapter 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 three formats: inhx8m (the default), inhx8s, inhx32.
c	Do not ignore case in source code. By default gpasms to treat “fooYa” and “FOOYA”as equal.
d symbol[=value]	Equivalent to “#define <symbol> <value>”.
e [ON|OFF]	Expand macros in listing file.
h	Display the help message.
I	Specify an include directory.
l	List the supported processors.
m	Memory dump.
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.

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.

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.

Chapter 2

Syntax

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 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 and
||	logical or
&	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 not

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.

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

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 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”.

Chapter 3

Directives

3.1 Code generation

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

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.

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 3.4 for demonstrates conditional assembly.

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
```

3.5 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.

3.6 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 hexidecimal value of expression. For 18cxx devices idlocation expression1 is set to the hexidecimal 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

CBLOCK

```
CBLOCK <expression>
```

Marks the beginning of a block of constants. gpasm allocates values for symbols in the block starting at the value given to CBLOCK.

See also: EQU

CONSTANT

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

Delcares <label> equal to <expression>. Similar to SET and VARIABLE, except it can not be changed once assigned.

See also: 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 seperate 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>.
See also: 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

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> occurances of the program word or byte <expression>.
See also: DATA DW ORG

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> appeared previously in a '#define' directive, gpasm assembles the following code.

See also: IF, IFNDEF, ELSE, ENDIF

IFNDEF

```
IFNDEF <symbol>
```

Begin a conditional assembly block. If <symbol>has not appeared previously in a '#define' directive, gpasm assembles the following code.

See also: IF, IFNDEF, ELSE, ENDIF

LIST

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

Enables output to the list (".lst") file. It may also change the state of gpasm in various surprising ways:

option	description
b=nnn	Sets the tab spaces
f=<format>	Set the hex file format. Can be inhx8m, inhx8s, or inhx32.
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.

See also: NOLIST, RADIX, PROCESSOR

LOCAL

```
LOCAL <symbol> [ , <symbol> ]
```

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

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_unitsl>. 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. <stringt> is an ASCII string enclosed by double quotes, no longer than 60 characters.
See also: SUBTITLE

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

Chapter 4

Instructions

4.1 Supported processors

gpasm currently supports the following processors:

- PIC12C508
- PIC12C509
- PIC16C54
- PIC16C55
- PIC16C56
- PIC16C57
- PIC16C58
- PIC16C62
- PIC16C63
- PIC16C64
- PIC16C65
- PIC16C71
- PIC16C74
- PIC16C83
- PIC16F83
- PIC16C84
- PIC16F84
- PIC16F874
- PIC16F876
- PIC16F877
- PIC18C242
- PIC18C252

- PIC18C442
- PIC18C452
- SX18
- SX20
- SX28

4.2 Instruction set summary

4.2.1 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

4.2.2 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

4.2.3 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	

4.2.4 Special macros

There are also a number of standard additional macros that gpasm reads from the file “special.inc”. Consult that file for more details. 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>

Chapter 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.

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.

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 superceded by command line

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

217 Hex format superceded 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.

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 6

Utilities

6.1 gpdasm

gpdasm is open source disassembler for Microchip's popular PICmicro (TM) line of microcontrollers. gpdasm is part of gpasm.

6.1.1 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.
m	Memory dump hex file.
p [pic12|pic14]	Select processor family.
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".

6.1.2 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.

6.2 gpvc

gpvc is open source cod file viewer for Microchip's popular PICmicro (TM) line of microcontrollers. gpvc is part of gpasm.

6.2.1 Running gpvc

The general syntax for running gpdasm 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 gpasm 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".

Index