TechTools CVASM16 Reference

Contents

CVASM16 Assembler

When you write programs for PIC microcontrollers, you'll use a text editor to create source code. Source code is the format that you're accustomed to looking at; it contains English-like labels, instructions, and data.

Before your source code can be used by a programmer, emulator, or other development tool, it must be converted into hex code. Hex code is the "machine-readable" version of source code; it contains instructions and data in the form of hexadecimal data, which can be executed by the PIC.

An assembler is a piece of software that converts source code into hex code. For instance, this line of source code:

```
        CALL    SENDBYTE          ;Call send routine
```

assembles into just two bytes of hex code:

```
        2420h
```

It's possible to write programs directly in hex code, using the individual machine codes that make up each instruction. However, most people find it preferable to use an assembler.

The TechTools assembler (CVASM16) is unique in its ability to accept two instruction sets: our own 8051-like instruction set, and the original Microchip instruction set. Many customers appreciate the TechTools instruction set, because it resembles other processors. However, there are certainly customers who prefer the Microchip instructions.

If you plan to use source code in the Microchip format, please note that while our assembler will accept their basic instructions, it will not accept various other aspects of their source code. The syntax of numbers is sometimes different, and our assembler does not recognize Microchip assembler directives and macros. If you plan to write a lot of code in the Microchip format, then you may wish to use the Microchip assembler (MPASM).

CVASM16 Assembler

If you already know how to use an assembler, you'll probably find our's to be quite simple. However, if you have little or no experience with assemblers, you should read the following pages before attempting to use the assembler.

Software Installation

If you installed our TDE or PICwriter software, CVASM16 was automatically installed into the TDE or PICwrite directory, and you can skip this step.

Insert the CD in your CDROM drive. If you have "auto run" enabled, the CD menu will launch automatically, or run the "Tech_cd.exe" program to launch the CD menu. After the CD menu has started, select the "Software" button from the selection on the right. Now choose the "CVASM16" button. This will install the assembler files, create a Program Group called "TechTools" and insert shortcuts to the CVASM16 documents and an "Uninstall" utility. The CVASM16 assembler is a DOS executable which requires command-line parameters and can be executed by opening a DOS prompt and typing CVASM16.

Running the Assembler

To assemble your source code into a hex file, type the following command at the DOS prompt:

CVASM16 filename	Assembles text file filename.src into hex file filename.obj.

As shown above, only the name of the source file is given. The assembler uses the same name for the hex file, but replaces the extension with .obj. If you do not specify an extension as part of the source file name, the assembler will assume that the extension is .src.

An example is shown below:

```
CVASM16  EXAMPLE
```

The assembler would produce a hex file called EXAMPLE.OBJ from the source file called EXAMPLE.SRC.

Generating Assembly Listings

It's possible to have the assembler create an "assembly listing" of your program. An assembly listing is a duplicate of the source code, but with hex code information (line number, address, opcode, & data) preceding each original line. To have the assembler create a listing file, simply add "/L" after the filename:

CVASM16 filename /L	Assembles text file filename.src into hex file filename.obj and creates a listing file called filename.lst.

Command-Line Options

The PIC assembler has several options which can be invoked when it is run. These command-line options are shown below:

CVASM16 filename	Assembles text file filename.src into hex file filename.obj.
CVASM16 filename.xxx	Assembles text file filename.xxx into hex file filename.obj.
CVASM16 filename /L	Assembles text file filename.src into hex file filename.obj and creates a listing file called filename.lst.
CVASM16 filename /D	This option, if used with the /L option, will add a list of the default PIC symbols in the generated .lst file.
CVASM16 filename /S	Assembles text file filename.src into hex file filename.obj, but suppresses TechTools device data normally included in the file. This may be useful with other tools that won't accept any extra data.

This will create a program LIST file that includes all default PIC symbols for the processor indicated in the DEVICE directive.

Assembler Basics

The purpose of the assembler is to convert assembly language source code into hex code. The assembler accomplishes its task in two passes:

Pass 0 - The source code is scanned in an attempt to resolve all symbols. This is possible if all *origin*, *define space*, and *equate* directives can be resolved (*equated* symbols may be referenced by *origin* or *define space*). All other symbols can be resolved by byte-offsets which are determined by the mnemonic/operand combinations. If Pass 0 is successful, the assembler will advance to Pass 1. If Pass 0 is unsuccessful, a list of errors will be shown and assembly will be aborted.

Pass 1 - The source code is scanned once more in order to assemble the hex code. Since all symbols were resolved in Pass 0, all instructions and miscellaneous directives can be fully resolved in Pass 1. If Pass 1 is successful, a hex file containing the assembled code will be created. If Pass 1 is not successful, a list of errors will be shown and assembly will be aborted.

In addition to a hex file, the assembler can be used to generate an assembly listing. An assembly listing shows line numbers, equated values, addresses, data, and original source code *(for more information, see the section Generating Assembly Listings on the previous page).*

Addressing Definitions

Throughout your programs, you'll refer to bits and bytes by their addresses. Depending on the instruction being used and the item being referred to, the address will be given in one of the following forms:

addr8	An 8-bit address (on 16C5x devices, the lower half of the current 512-word page).
addr9	A 9-bit address (on 16C5x devices, within the current 512-word page).
addr11	An 11-bit address (anywhere in program memory in a device with 2K of memory).

addr12	A 12-bit address (anywhere in program memory in a device with 4K of memory).
bit	An address for bitwise operations **Example: PortC.3 = bit 3 of port C**
fr	A file register (RAM) address.
rel	A relative address ranging from -7Fh to +80h.
literal	An immediate 8-bit value.

Data Types

Eight data types are allowed in the assembler. These data types are:

Symbol/label	Binary value
Local symbol/label	ASCII value
Decimal value	Assembly address (origin)
Hex value	EEPROM address (origin)

The examples below show various data types:

100	Decimal value *100*
18h	Hex value *18*
0A7h	Hex value *A7*
1011b	Binary value *1011*
‘A’	ASCII value for the letter *A* (65 decimal)
Start	Label called *Start*
:loop	Local label called *:loop*
$	Current assembly address
%	Current EEPROM address

Hex values with A-F as the first digit must have a leading zero. In the example shown, the value “A7” must be given as “0A7”.

In addition to single-character text, entire strings can be generated using the RETW instruction:

```
retw      ‘The fox jumped over the lazy dog’
```

Expressions

Mathematical expressions are used in many instructions. These expressions may be created using the following operators:

&	*Logical AND*	/	*Divide*
|	*Logical OR*	<<	*Shift left*
^	*Exclusive OR*	>>	*Shift right*
+	*Add*	<	*High byte*
-	*Subtract*	>	*Low byte*
*	*Multiply*	.	*Bit address*

Some example expressions:

setb	PortA.0	*Set bit 0 on port A.*
mov	Count+3,#88h	*Store 88h in location Count+3.*
ds	N*2	*Define empty space of Nx2 bytes.*

▶ All expressions are resolved strictly from left to right. Please make note of this, since it may affect the result of expressions. For instance, the expression 5+2*4 would normally be resolved as (2*4)+5, for a result of 13. Since our assemblers resolve expressions strictly from left to right, however, the example would be resolved as (5+2)*4, for a result of 28.

Symbols & Labels

Symbols are used to name locations and values within your program. Many people refer to address symbols as “labels”, but both have the same effect. For instance, by assigning a symbol to the start of an important routine, you can later call that routine by its name, rather than its address. And by giving a symbol to a common value, you can refer to it by its name. Rather than typing “212” many times in your program, you can type HOT = 212 at the beginning of your program, and then use HOT wherever you need it. This is also quite handy if your idea of what’s “hot” changes. By changing the symbol definition, you can easily redefine HOT to be a different temperature.

Symbols may be up to 32 character long. They must begin with a letter or underscore (_) and must contain only letters, numbers, underscores, and colons. Further, if you're labeling an address (such as the start of a routine), the label must start at the beginning of the line.

Here are some examples of valid symbols:

```
min_count        =        20h
maximum_count    =        21h
begin            mov      min_count,#05h
```

Local Labels

By default, labels are *global,* which means that they can be "seen" from anywhere in your program. Sometimes, however, you may want to use a *local* label, which can only be "seen" within a limited part of your program (the area in which the local label can be seen starts at the preceding global label, and continues up to the following global label).

Local labels have the same syntax rules as global labels, except they must begin with a colon (:), and must be referenced with a colon. Local labels can be referenced from from outside their normal area by referring to preceding global label name:local label name.

The following code demonstrates how to use the local label :loop for common looping purposes within two globally-labeled routines.

```
 1   Routine1    mov    count,#100      (global label Routine1)
 2   :loop       call   send_a          (local label :loop)
 3               djnz   count,:loop     (jump to line 2)
 4               ret

 5   Routine2    mov    count,#200      (global label Routine2)
 6   :loop       call   send_b          (local label :loop)
 7               djnz   count,:loop     (jump to line 6)
 8               ret

 9   Routine3    mov    count,#250      (global label Routine3)
10               jmp    Routine2:loop   (jump to line 6)
```

Areas of local labels are indicated by highlighting.

Local label can be called from outside its usual area.

In addition to the printed tables in the PICTOOLS manual, all predifined symbols for the device specfied can be generated by using the /L /D command line options. This will create a program LIST file that includes all default PIC symbols for the processor indicated in the DEVICE directive.

Default Symbol Tables

When the assembler is started, its symbol table is initialized with various PIC symbols, such as C for the Carry register, RA for Port A, etc. This saves you from having to define every register and bit in the PIC.

As the number of different PICs has grown, it has become necessary to have separate symbol tables for each PIC. At the beginning of your program, you must tell the assembler which PIC is being used. This is done with the device directive (explained later in this chapter).

If you'd like to see the symbol table for a particular PIC, please refer to the printed tables in Appendix C of our PICTOOLS Manual.

Comments

Comments can be placed anywhere in your source code, beginning at any point on a line and continuing to the end. A semicolon initiates a comment. The following are example comments:

```
;
;Move literal value 61h into w
;

Input           =       10h
Output          =       11h

                mov     Input,#61h      ;Load 61h into Input
                mov     Output,#10h     ;Load 10h into Output

                call    talk_host       ;Call routine to
                                        ;communicate with
                                        ;host
```

Blank lines can be used to provide space between lines and make the code more readable.

Assembler Directives

Assembler directives are instructions that direct the assembler to do something. Directives do many things; some tell the assembler to set aside space for variables, others tell the assembler to include additional source files, and others establish the start address for your program. The directives available are shown below:

=	Assigns a value to a symbol (same as EQU)
EQU	Assigns a value to a symbol (same as =)
ORG	Sets the current origin to a new value. This is used to set the program or register address during assembly. For example, ORG 0100h tells the assembler to assemble all subsequent code starting at address 0100h.
DS	Defines an amount of free space. No code is generated. This is sometimes used for allocating variable space.
ID	Sets the PIC's identification bytes. PIC16C5x chips have two ID bytes, which can be set to a 2-byte value. Newer PICs have four 7-bit ID locations, which can be filled with a 4-character text string
INCLUDE	Loads another source file during assembly. This allows you to insert an additional source file into your code during assembly. Included source files usually contain common routines or data. By using an INCLUDE directive at the beginning of your program, you can avoid re-typing common information. Included files may not contain other included files.

Note: **the DEVICE directive must be included at the beginning of your program. Otherwise, the assembler will not know which PIC to assemble for.**

When using the directive, you must include a minimum of the device type (listed to the right); all other settings are optional. If you do not specify a certain setting, such as oscillator type, then the default value for an erased PIC will be assumed.

DEVICE Establishes the device type, oscillator type, watchdog status, and code-protect status. The following options are used with DEVICE to define the PIC being used:

PIC16C52	PIC16C64	PIC12C508A	PIC16C622A
PIC16C54	PIC16C64A	PIC12C509	PIC16CE623
PIC16C54A	PIC16C65	PIC12C509A	PIC16CE624
PIC16C54B	PIC16C65A	PIC12CE518	PIC16CE625
PIC16C54C	PIC16C66	PIC12CE519	PIC16C641
PIC16C55	PIC16C67	PIC12C671	PIC16C642
PIC16C55A	PIC16C71	PIC12C672	PIC16C661
PIC16C56	PIC16C71A	PIC12CE673	PIC16C662
PIC16C56A	PIC16C72	PIC12CE674	PIC16C710
PIC16C57	PIC16C72A	PIC16C554	PIC16C711
PIC16C57A	PIC16C73	PIC16C554A	PIC16C715
PIC16C57C	PIC16C73A	PIC16C556	PIC16F873
PIC16C58	PIC16C74	PIC16C556A	PIC16F874
PIC16C58A	PIC16C74A	PIC16C558	PIC16F876
PIC16C58B	PIC16C76	PIC16C558A	PIC16F877
PIC16C505	PIC16C77	PIC16C620	PIC16C923
PIC16C61	PIC16F83	PIC16C620A	PIC16C924
PIC16C62	PIC16F84	PIC16C621	PIC14000
PIC16C62A	PIC16C84	PIC16C621A	
PIC16C63	PIC12C508	PIC16C622	

RC_OSC	*Use RC oscillator*
XT_OSC	*Use XT oscillator*
HS_OSC	*Use HS oscillator*
LP_OSC	*Use LP oscillator*
IRC_OSC *	*Use IRC oscillator*
XRC_OSC *	*Use XRC oscillator*
WDT_OFF	*Turn watchdog timer off*
WDT_ON	*Turn watchdog timer on*
PROTECT_OFF	*Code-protect off*
PROTECT_HALF *	*Code-protect half on*
PROTECT_3_QTR *	*Code-protect 3/4 on*
PROTECT_ON	*Code-protect fully on*
PWRT_OFF *	*Turn power-up timer off*
PWRT_ON *	*Turn power-up timer on*
BOD_OFF *	*Turn brownout detect off*
BOD_ON *	*Turn brownout detect on*

Note: **many newer PICs have permanent code-protect fuses. Once these fuses are programmed, you may not be able to "un-program" them (even on erasable PICs).**

MCLR_OFF *	*Turn brownout detect off*
MCLR_ON *	*Turn brownout detect on*

** These options are only available on some of the newer PICs.*

In addition to giving the assembler information, the DEVICE directive passes useful data to other tools down the line. For instance, the programmer software reads this data to preset options on the screen.

RESET Sets the reset start address. This address is where program execution will start following a reset.

A jump to the given address is inserted at the last location in memory. After the PIC is reset, it starts executing code at the last location, which holds the jump to the given address.

RESET is only available for PIC16C5x chips.

EEORG Sets the current data EEPROM origin to a new value. This is used to set the data EEPROM address during assembly. This directive usually precedes EEDATA.

EEORG is only available for PICs that have EEPROM memory .

EEDATA Loads data EEPROM with given values. This provides a means of automatically storing values in the data EEPROM when the PIC is programmed. This is handy for storing configuration or start-up information.

EEDATA is only available for PICs that have EEPROM memory.

Assembler Directive Examples

The following examples are valid assembler directives:

```
        DEVICE   PIC16C54,RC_OSC,WDT_OFF,PROTECT_OFF
        DEVICE   PIC16C64,PWRT_OFF,PROTECT_ON
        DEVICE   PIC16C71,XT_OSC,PWRT_ON
        DEVICE   PIC16C84,PROTECT_ON

Digit   =        43h            ;Assign value 43h to Digit
Max     EQU      1Ah            ;Assign value 1Ah to Max

        ORG      10h            ;Set assembly address to 10h

Count   DS       2              ;Define 2 bytes at 10h & 11h
                                ;Bytes can be referred to
                                ;later as Count and Count+1

        ID       1234h          ;Set 16C5x ID to 1234h
        ID       ‘ABCD’         ;Set newer PIC ID to 'ABCD'

        INCLUDE‘KEYS.SRC’       ;Include KEYS.SRC file at
                                ;point of insertion

        RESET    Start          ;Set 16C5x reset jump to
                                ;location at Start

Start   mov      Count,#00      ;This will be executed when PIC is reset

        EEORG    10h            ;Set EEPROM address to 10h
        EEDATA   02h,88h,34h    ;Store 3 bytes in EEPROM
```

Source Code Formatting

We recommend that you format your source code with evenly spaced tabs, preferably 8 spaces each, since this will lend consistency to assembly listings. The assemblers are not case sensitive (except in the instance of strings), so you may follow your own convention for using upper and lower case.

Error Messages

During assembly, any syntax errors will be brought to your attention. You can pause an error list by typing CTRL-S; you can also press ESC or CTRL-C to abort the error list.

The following error messages may occur during assembly. They will always be preceded by "ERROR ***filename*** **xxx**:", where ***filename*** is the file and **xxx** is the line number where the error occurred.

Address limit of xxxxh was exceeded: Data was assembled at an address which exceeded the limit for the given PIC.

Attempt to divide by 0: An attempt was made to divide a quantity by zero.

Bit number must be from 0 to 7: A bit-address expression attempted to use a bit number greater than 7.

Data was already entered at location xxxxh: An object code location which had already been assigned data, was written to again.

Equate directive must be preceded by a symbol: An "EQU" or "=" directive was not preceded by a necessary symbol.

Illegal mnemonic: The assembler encountered an unknown directive or instruction.

Include files cannot be nested: An INCLUDE directive was found in an included file.

Syntax error in operand: The operand contained an expression which did not follow proper syntax.

Invalid filename for include file: An invalid filename was given in an INCLUDE directive.

Line cannot exceed 256 characters: Line length exceeded the line limit of 256 characters.

Mnemonic field cannot exceed 7 characters: More than 7 characters were in the mnemonic (instruction) field.

Illegal mnemonic/operand combination: The operand structure did not match the instruction's or directive's possibilities.

Literal value must be from 0 to 0FFh: A value which needed to be from 0 to 255, was greater than 255.

Operand field cannot exceed 256 characters: More than 256 characters were in the operand field after expansion by the assembler.

Redefinition of symbol xxxx: An attempt was made to redefine a symbol that was already defined.

Symbol field cannot exceed 32 characters: More than 32 characters were in the symbol field.

(operand) symbol must contain only letters, numbers, '_', and ':': A symbol contained illegal characters.

(operand) symbol is a reserved word: A symbol was identical to a reserved word. The symbols you define must not be reserved words.

(operand) symbol is too long: A symbol referenced in the operand exceeded 32 characters.

(operand) symbol must begin with a letter or '_': A symbol started with an illegal character.

Symbol table full: The symbol table's 16K limit was exceeded.

Use of unknown symbol xxxx: A symbol was referenced in the operand, which was never declared earlier in your program.