2 ASSEMBLER

Overview

The `easm21k` assembler runs from an operating system command line or from the VisualDSP++ environment. The assembler processes assembly source, data, and header files, and produces an object file. Assembler operations depend on two types of controls: assembler directives and assembler switches.

Assembler directives are coded in your assembly source file. The directives allow you to define variables in your program, set up some hardware features, and identify your program's sections* for placement within DSP memory. The assembler uses directives for guidance as it translates your source into object code.

Assembler switches are specified on the operating system's command line or in the Assemble tab of the VisualDSP++ environment's Project Options dialog box. These switches allow you to control how the assembler processes your programs. Using these switches, you select features, such as search paths, output file names, and macro preprocessing, among others.

This chapter provides assembler information that you need to know when developing and assembling programs for the ADSP-21xxx DSPs.

---

* The assembler section (or `.SECTION`) declaration referred to here corresponds to a linker input section.

This chapter contains the following information on the assembler:

Assembler Guide

The guide section describes the process of developing programs in the ADSP-21xxx DSP assembly language. The discussion covers conventions you should follow when assembling source programs from the operating system's command line.

Software developers using the assembler should be familiar with the following operations:

- "Writing Assembly Programs" on page 2-3
- "Preprocessing a Program" on page 2-11
- "Reading a Listing File" on page 2-12
- "Setting Assembler Options" on page 2-14

For information about the DSP architecture, including the DSP instruction set that you use when writing assembly programs, see the hardware and instruction set manuals.

Writing Assembly Programs

Write your assembly language programs using the VisualDSP++ editor or any editor that produces text files. Do not use a word processor that embeds special control codes in the text. Append an `.ASM` extension to your source files to identify them as the SHARC DSP assembly files.

Assemble your source files, either using the assembler's command line or within the VisualDSP++ environment. In the default mode of operation, the assembler processes an input file through the listed stages to produce a binary object file (`.DOJ`) and an optional listing file (`.LST`).

Object files serve as input to the linker when you link your executable program. These files are in Executable and Linkable Format (ELF), an industry-standard format for object files. In addition, the assembler can embed binary information in Debugging Information Format (DWARF-2) for source level debugging.

Listing files are text files that you can read for information on the results of the assembly process.

Figure 2-1 on page 2-5 shows a graphical overview of the assembly process. The figure shows the preprocessor processing the assembly source (`.ASM`) and initialization data (`.DAT`) files. The assembly source file often contains preprocessor commands, such as `#include`, that causes the preprocessor to include header files (`.H`) into your source program. The preprocessor's only output, an intermediate preprocessed file (`.IS`), is the assembler's primary input.

A binary object (.DOJ) and an optional listing (.LST) files are final results of the successful assembly.

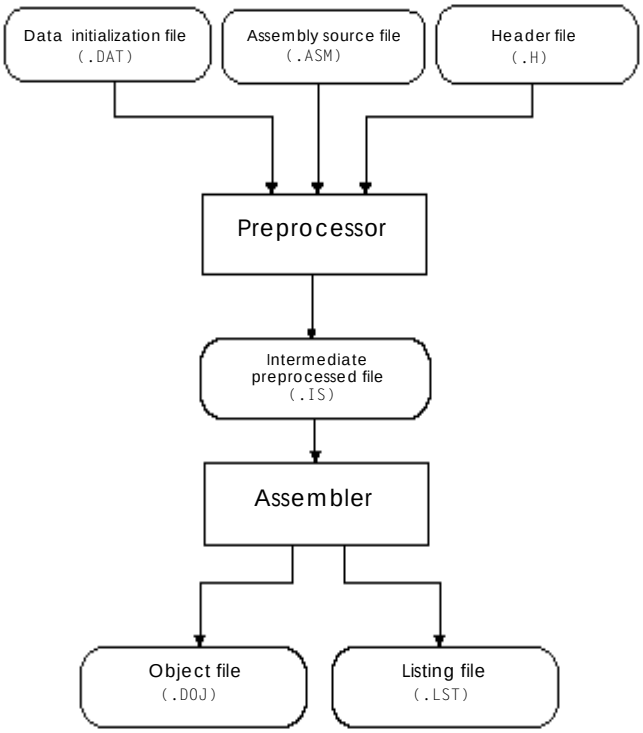


Figure 2-1. Assembler Input & Output Files

Program Content

Statements within an assembly source file are comprised of assembly instructions, assembler directives, and preprocessor commands. Instructions assemble to executable code, while directives and commands modify the assembly process. The syntax of these statement types is as follows:

- Assembly instructions

  Instructions follow the DSP's instruction set syntax documented in the DSP Instruction Set manuals. Each instruction begins with a keyword and ends with a semicolon (`;`). To mark the location of an instruction, place an address label at the beginning of an instruction line or on the preceding line. End the label with a colon (`:`) before beginning the instruction. You can refer to this memory location in your program using the label instead of an absolute address. Although there is no length restriction when defining labels, it is convenient to limit them to the length of a screen line, typically eighty characters.

  Labels are sensitive to case. So, `easm21k` treats "`outer`" and "`Outer`" as unique labels.

  Examples:

  ```
  outer: DM(I1,M1)=F8;
  start: r0=source;
  ```

- Assembler directives

  Directives begin with a period (`.`) and end with a semicolon (`;`). The period must be the first character on the line containing the directive. The assembler does not differentiate between directives in lowercase or uppercase characters.

  Note that this manual prints directives in uppercase to distinguish them from other assembly statements.

Examples:

```
.PRECISION 40;
.ROUND_ZERO;
```

For a description of the ADSP-21xxx DSP directive set, see "Assembler Directives" on page 2-36.

- Preprocessor commands

  Preprocessor commands begin with a pound sign (#) and end with a carriage return. The pound sign must be the first character on the line containing the command. If the command is longer than one line, use a backslash (\) and a carriage return to continue the command on the next line. Do not put any characters between the backslash and the carriage return. Unlike other assembly statements, preprocessor commands are case-sensitive and must be lowercase. For a list of these commands, see "Preprocessing a Program" on page 2-11.

  Examples:

```
#include "const.h"
#define PI 3.14159
```

Figure 2-2 on page 2-9 contains an example assembly source file.

Program Structure

An assembly source file must describe how code and data are mapped into the memory on your target DSP. There are two types of memory: data memory, which typically contains data and memory-mapped ports, and program memory, which typically contains code (and can also store data). The way you structure your code and data into memory should follow from the memory architecture of the target DSP.

The mapping of code and data is accomplished using the `.SECTION` directive (formerly `.SEGMENT` and `.ENDSEG`). The `.SECTION` directive defines groupings of instructions and data that are set as contiguous memory

addresses in the DSP. Each .SECTION name corresponds to an input section names in the Linker Description File (.LDF).

Some suggested section names that you could use in your assembly source appear in Table 2-1. Using these predefined names in your sources makes it easier to take advantage of the default Linker Description File included in your DSP system. For more information on the LDF, see the *VisualDSP++ 2.0 Linker & Utilities Manual for ADSP-21xxx DSPs.*

Table 2-1. Suggested Section Names

.SECTION Name	Description
seg_pmco	A section in Program Memory that holds code.
seg_dmda	A section in Data Memory that holds data.
seg_pmda	A section in Program Memory that holds data.
seg_init	A section in Program Memory that holds system initialization data.
seg_rth	A section in Program Memory that holds system initialization code and interrupt service routines.
seg_stak	A section in Data Memory that holds run-time stack. Required by the C run-time environment.
seg_heap	A section in Data Memory that holds run-time heap. Required by the C run-time environment.

You may create sections in a program by grouping elements to meet hardware constraints. To group code that reside in off-chip memory, declare a section for that code and place that section in the selected memory with the linker. Figure 2-2 on page 2-9 shows how a program divides into sections that match the program and data memory segmentation of a DSP system.

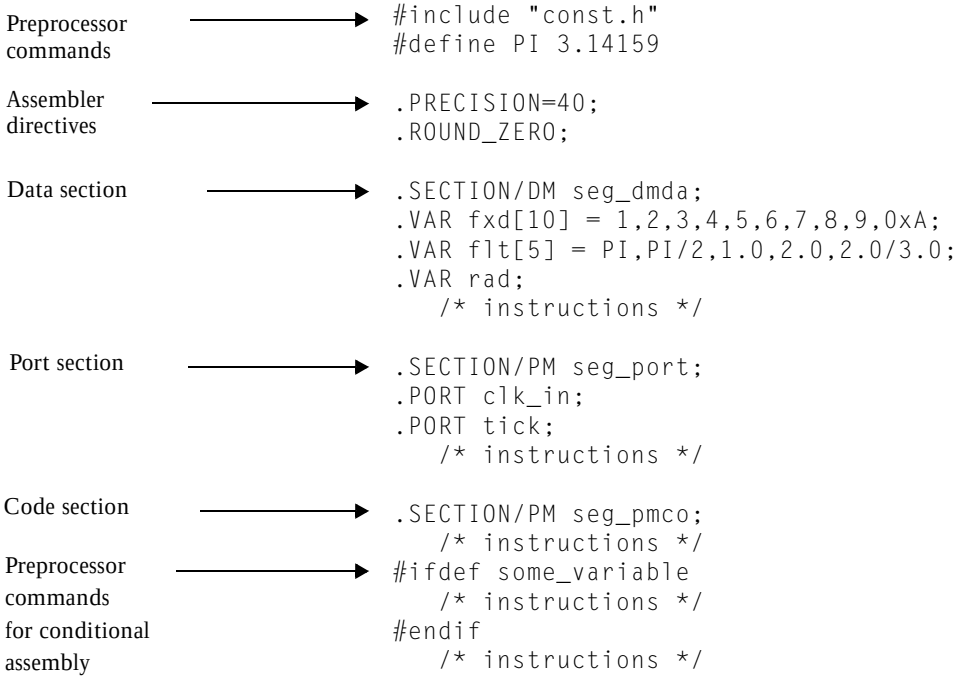


Figure 2-2. Example Assembly Source File

The sample assembly program divides into sections; each section begins with a `.SECTION` directive and ends with the occurrence of the next `.SECTION` directive or end-of-file.

The source program contains the following sections:

- Data Section — `seg_dmda`. Variables and buffers are declared and can be initialized.
- Port Section — `seg_port`. I/O ports mapped to program memory; each port has a unique name to identify it for reading and writing.
- Program Section — `seg_pmco`. Data, instructions, and possibly other types of statements are in this section, including statements that are needed for conditional assembly.

Looking at Figure 2-2, notice that an assembly source file often begins with one or more statements, such as `#include` to include other files in your source code or `#define` to define macros. The `.PRECISION` and `.ROUND` directives tell the assembler to store floating-point data with 40-bit precision and to round a floating-point value to a closer-to-zero value if it does not fit in the 40-bit format.

Program Interfacing Requirements

At some point, you may want to interface your assembly program with a C or C++ program. The C/C++ compiler supports two methods for mixing C/C++ and assembly language: embedding of assembly code in C or C++ programs and linking C and assembly routines.

To embed (inline) assembly code in your C or C++ program, use the `asm()` extension. To link together programs that contain C/C++ and assembly routines, use assembly interface macros. These macros facilitate the assembly of mixed routines. For more information about these methods, see the *VisualDSP++ 2.0 C/C++ Compiler & Library Manual for ADSP-21xxx DSPs.*

Preprocessing a Program

The assembler includes a preprocessor that allows you to use C-style preprocessor commands in your assembly source files. Table 3-3 on page 3-17 lists preprocessor commands and provides a brief description of each command. The preprocessor automatically runs before the assembler unless you use the assembler's `-sp` (skip preprocessor) switch.

Preprocessor commands are useful for modifying assembly code. For example, you can use the `#include` command to fill memory, load configuration registers, and set up DSP parameters. You can use the `#define` command to define constants and aliases for frequently used instruction sequences. The preprocessor replaces each occurrence of the macro reference with a corresponding value or a series of instructions. For example, the macro `PI` in Figure 2-2 on page 2-9 is replaced with the characters `3.14159` during preprocessing.

Note that the listing files keep any comments.

For information on the ADSP-21xxx DSPs' preprocessor command set, see "Preprocessor Directives/Commands" on page 3-16.

Reading a Listing File

A listing file (`.LST`) is an optional output text file that lists the results of the assembly process. A sample listing file appears in Listing 2-1. The source file, `DFT.ASM`, is available in the `…/21k/Examples/DASM_Examples/DFT` directory.

Each listing file provides the following information:

- Address — The first column contains the offset from the `.SECTION`'s base address.
- Opcode — The second column contains the hexadecimal opcode that the assembler generates for the line of assembly source.
- Line — The third column contains the line number in the assembly source file.
- Assembly Source — The fourth column contains the line of assembly source from the file including any comments.

Listing 2-1. Listing File Example

```
Page 1  dft.asm
ADI EASM21k (2.1.2.0) 05 Sep 2001 14:02:21

offset       opcode  line
======       ======  ====
                        1                                          /*
                        2 DFT.ASM         ADSP-2106x Discrete Fourier
                          Transform
                        3 This routine performs an N point real DFT
                          according to the following equation:
                        4
                        5             N-1
                        6 real(k)+j*imag(k) = SUM input(n)[C - j*S];
                          k=0 to N-1
                        7             n=0
                        8
                        9 where: C=cos(2*pi*k*n/N), S=sin(2*pi*k*n/N),
                          j=sqrt(-1)
                       10                                           */
```

```
                     11
                     12 #include "def21060.h"          /* Memory Mapped
                        IOP register definitions */
                     13 #define N 64     /* Constant for number of
                        points in input */
                     14
                     15 .SEGMENT/DM     dm_data;   /* Declare
                        variables in data memory */
 0   4180000000      16 .VAR input[N]= "test64.dat";
                     16
40   0000000000      17 .VAR real[N];
80   0000000000      18 .VAR imag[N];
                     19 .ENDSEG;
                     20
                     21 .SEGMENT/PM     pm_data;        /* Declare
                        variables in program memory */
 0 000000000000      22 .VAR sine[N]= "sin64.dat";      /* Cosine is
                        derived using a shifted */
                     22
                     23 .ENDSEG;   /* pointer to this circular buffer.*/
                     24
                     25 .SEGMENT/PM     pm_rsti;  /* The reset vector
                                  resides in this space */
 0 000000000000      26        NOP;
 1 0f7100108421      27        USTAT2= 0x108421;  /* 1st instr. to be
                        executed after reset */
 2 117100000002      28        DM(WAIT)=USTAT2;   /* Set external memory
                        waitstates to 0 */
 3 063e00000000      29         JUMP start;
                     30
                     31 .ENDSEG;
                     32
                     33 .SEGMENT/PM     pm_code;    /* Example setup for
                        DFT routine */
 0                   34 start:          M1=1;
 0 0f2100000001      34
 1 0f2900000001      35         M9=1;
 2 0f4000000000      36         B0=input;
 3 0f3000000040      37         L0=@input;     /* Input buffer is
                        circular */
 4 0f1100000080      38         I1=imag;
 5 0f3100000000      39         L1=0;
 6 06be0400000a  40     CALL dft (DB);    /* Example delayed call
                        instruction */
 7 0f1200000040      41          I2=real;   /* In delay field of call */
 8 0f3200000000      42           L2=0;    /*           ''          */
 9                   43 end:            IDLE;
 9 008000000000      43
```

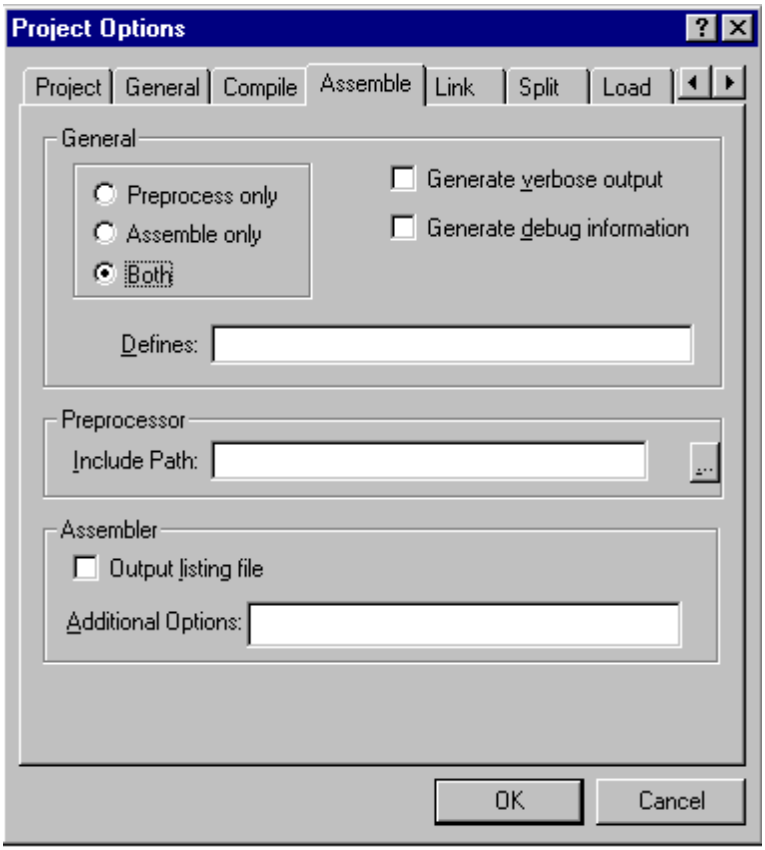
```
                    44
                    45 /*    DFT Subroutine    */
 a                 46 dft: B8=sine;                    /* Sine pointer */
 a 0f4800000000     46
 b 0f3800000040     47          L8=@sine;
 c 0f4900000000     48      B9=sine;  /* Derive cosine from sine by */
 d 0f1900000010     49      I9=sine+N/4; /* shifting pointer
                        over 2pi/4 */
 e 0f3900000040     50      L9=@sine;               /* and using a
                        circular buffer.*/
 f 0f1a00000000     51      I10=0;   /* I10 is used to increment the */
10 0f3a00000000     52      L10=0;   /* frequency of sine lookup.*/
11 0f0f00000000     53      F15=0;   /* Zero to clear accumulators */
12 0c004000001c     54      LCNTR=N, DO outer UNTIL LCE;
13 71af940a18f0     55      F8=PASS F15, M8=I10  /* Update frequency */
14 2040428a19f0     56      F9=PASS F15, F0=DM(I0,M1), F5=PM(I9,M8);
15 503f02130c05     57      F12=F0*F5, F4=PM(I8,M8);
16 0c003f000018     58      LCNTR=N-1, DO inner UNTIL LCE;
17 204042d8d904     59     F13=F0*F4, F9=F9+F12, F0=DM(I0,M1),
                            F5=PM(I9,M8);
18                  60 inner: F12=F0*F5, F8=F8-F13, F4=PM(I8,M8);
18 503f0259c811     60
19 013e0058d904     61      F13=F0*F4, F9=F9+F12;
1a 547e8488288d     62     F8=F8-F13, DM(I2,M1)=F9;    /* Write real
                        result */
1b 047e88000000     63      MODIFY(I10,M9); /* Increment frequency */
1c                  64 outer: DM(I1,M1)=F8;     /* Write imaginary
                        result */
1c 527e84000000     64
1d 0a3e00000000     65          RTS;
                    66 .ENDSEG;
                    67
```

Setting Assembler Options

When developing a DSP project, you may find it useful to modify the assembler's default options. The way you set the assembler options depends on the environment used to run your DSP development software:

- From the operating system command line, you select the assembler's command-line switches. For more information on these switches, see the "Assembler Command-Line Interface" on page 2-16.

- From the VisualDSP++ environment, you choose the assembler's options on the Assemble tab in the Project Options dialog box selected via the Project menu. For more information on how to set these project options, see the *VisualDSP++ User's Guide for ADSP-21xxx DSPs* and online Help. For example:



Assembler Command-Line Reference

The ADSP-21xxx SHARC DSP assembler (`easm21k`) processes data from assembly source (`.ASM`), data (`.DAT`), header (`.H`), and preprocessed (`.IS`) files and generates object files in Executable and Linkable Format (ELF), an industry-standard format for binary object files. The assembler's primary output file has a `.DOJ` extension. The assembler's secondary outputs are an optional listing (`.LST`) file and information in binary Debugging Information Format (DWARF-2), which is embedded in the object file. By linking together separately assembled object files, the linker produces executable programs (`.DXE`).

You can archive the output of an assembly process into a library file (`.DLB`), which can then be linked with other objects into an executable. Because the archive process performs a partial link, placing frequently used objects in a library reduces the time required for subsequent links. For more information on the archiver, see the *Linker & Utilities Manual for ADSP-21xxx DSPs*.

This section provides reference information on the assembler's command-line switches and their use. The ADSP-21xxx DSP assembler switches are accessible either from the operating system's command line or via the VisualDSP++ project environment.

Assembler Command-Line Interface

This section describe the `easm21k` assembler command-line interface and switch (option) set.

Switches control certain aspects of the assembly process, including library searching, listing, and preprocessing. Because the assembler automatically runs the preprocessor as your program is assembled (unless you use the `-sp` switch), the assembler's command line can receive input for the preprocessor program and direct its operation. For more information on the preprocessor, see "Preprocessor" on page 3-1.

Running the Assembler

To run the assembler from the command line, type the name of the assembler program followed by arguments in any order:

```
easm21k [-switch1 [-switch2 …]] sourceFile
```

where:

- `easm21k` — Name of the assembler program for the ADSP-21xxx DSPs.

- *`-switch1,-switch2`* — The switches to process.

  The assembler offers many optional switches that select operations and modes for the assembler and preprocessor. Some assembler switches take a file name as a required parameter.

- *`sourceFile`* — Name of the source file to assemble.

The assembler outputs an error message when run without arguments.

The assembler supports relative and absolute path names. When you provide an input or output file name as a parameter, use the following guidelines for naming files:

- Include the drive letter and path string if the file is not in the current directory.
- Enclose long file names in double quotation marks, for example, `"long file name"`.
- Append the appropriate file name extension to each file.

The assembler uses each file's extension to determine what operations to perform. Table 2-2 on page 2-18 lists the valid file extensions that the assembler accepts.

The assembler handles file name extensions as follows:

- Files with an `.ASM` or no extension are treated as assembly source files to be assembled.
- Files with an `.H` extension named in an `#include` command are treated as header files to be preprocessed.
- Files with a `.DAT` extension named with an `-I` switch are treated as data initialization files to be searched.
- File name typed in lower or uppercase defines the same file.

Table 2-2 summarizes file extension conventions that the assembler follows.

Table 2-2. File Name Extension Conventions

Extension	Description
`.asm`	Assembly source file. Note that the assembler treats all files with unrecognized extensions as assembly source files with an `.asm` extension.
`.is`	Preprocessed assembly source file.
`.h`	Header file.
`.lst`	Listing file.
`.doj`	Assembled object file.
`.dat`	Data initialization file.

The following command line, for example:

```
easm21k -l p1Listing.lst -Dfilter_taps=100 -v -o bin\p1.doj p1.asm
```

runs the assembler with:

`-l p1Listing.lst` — Directs the assembler to output the listing file.

`-Dfilter_taps=100` — Defines the macro `filter_taps` as equal to 100.

`-v` — Displays verbose information on each phase of the assembly.

`-o bin\p1.doj` — Specifies the name and directory for the object output.

`p1.asm` — Specifies the assembly source file to assemble.

Assembler Command-Line Switch Summary

This section describes the `easm21K` command-line switches (options) in ASCII collation order. A summary of the assembler switches appears in Table 2-3, and a brief description for each switch starts on page 2-21.

Table 2-3. Assembler Command-Line Switch Summary

Switch Name	Description
`-21[020|060|061|062|065L|160|161]`	Generate code for ADSP-21xxx processors. If omitted, defaults to `-21060`.
`-D`*`macro`*`[=`*`definition`*`]`	Define *`macro`*.
`-g`	Generate debug information (DWARF-2 format).
`-h[elp]`	Output a list of assembler switches.
`-i` *`directory`*	Search *`directory`* for included files.
`-l` *`filename`*	Produces a listing file named <filename>. Nested source files ***are not*** included in the listing.
`-li` *`filename`*	Produces a listing file named <filename>. Nested source files ***are*** included in the listing.

Table 2-3. Assembler Command-Line Switch Summary (Cont'd)

`-M`	Make dependencies only, does not assemble.
`-MM`	Make dependencies and assemble.
`-Mo` *`filename`*	Specify *`filename`* for the make dependencies output file.
`-Mt filename`	Make dependencies for the specified source file.
`-o` *`filename`*	Output named object file.
`-pp`	Run preprocessor only.
`-r`	Remove preprocessor information from a listing file.
`-sp`	Skip reprocessing.
`-v[erbose]`	Display information about each assembly phase.
`-version`	Display version info about the assembler and preprocessor programs.
`-w`	Remove all assembler-generated warnings.

Assembler Command-Line Switch Descriptions

A description of each switch includes information about case-sensitivity, equivalent switches, switches overridden/contradicted by the one described, and naming and spacing constraints on parameters.

-21[020|060|061|062|065L|160|161]

The `-21[060|065L|061|062|160|161]` switch directs the assembler to generate code suitable for one of the following processors: ADSP-21020, ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, ADSP-21160, or ADSP-21161.

-D*macro*[=*def*]

The `-D` (define macro) switch directs the assembler to define a macro. If you do not include the optional definition string (`=def`), the assembler defines the macro as value 1.

Some examples of this switch are as follows:

```
-Dinput             // defines input as 1
-Dsamples=10        // defines samples as 10
-Dpoint="Start"     // defines point as the string “Start”
```

-g

The `-g` (generate debug information) switch directs the assembler to generate line number and symbol information in DWARF-2 binary format, allowing you to debug the assembly source files.

-h[elp]

The `-h` or `-help` switch directs the assembler to output to standard output a list of command line-switches with a syntax summary.

-i *directory*

The `-i directory` or `-I directory` (include directory) switch directs the assembler to append the specified directory or a list of directories separated by semicolons (;) to the search path for included files. These files are:

- header files (`.h`) included with the `#include` command
- data initialization files (`.dat`) specified with the `.VAR` directive

The assembler passes this information to the preprocessor; the preprocessor searches for included files in the following order:

- current project (`.dpj`) directory
- `\include` subdirectory of the VisualDSP installation directory
- specified directory (a list of directories). The order of the list defines the order of multiple searches.

Current directory is your `*.dpj` project directory, not the directory of the assembler program. Usage of full path names for the `-I` switch on the command line (omitting the disk partition) is recommended. For example,

```
easem21k -I "\bin\inlclude\buffer1.dat"
```

-l *filename*

The `-l` (listing) switch directs the assembler to generate the named listing file. Each listing file shows the relationship between your source code and the instruction opcodes that the assembler produces. If you omit the *filename*, the assembler outputs an error message. For more information about listing files, see "Reading a Listing File" on page 2-12.

-M

The `-M` (generate make rule only) assembler switch directs the preprocessor to output a rule, which is suitable for the make utility, describing the dependencies of the source file. After preprocessing, the assembler stops without assembling the source into an object file. The output, an assembly make dependencies list, is written to `stdout` in the standard command-line format:

```
source_file.doj: dependency_file.ext
```

where *`dependency_file.ext`* may be an assembly source file, a header file included with the `#include` preprocessor command, or a data file.

When the `-o` *`filename`* option is used with `-M`, the assembler outputs the make dependencies list to the named file.

-MM

The `-MM` (generate make rule and assemble) assembler switch directs the preprocessor to output a rule, which is suitable for the make utility, describing the dependencies of the source file. After preprocessing, the assembly of the source into an object file proceeds normally. The output, an assembly make dependencies list, is written to `stdout` in the standard command-line format:

```
source_file.doj: dependency_file.ext
```

where *`dependency_file.ext`* may be an assembly source file, a header file included with the `#include` preprocessor command, or a data file.

For example, the source `vectAdd.asm` includes the "`MakeDepend.h`" and `inits.dat` files. When assembling with

```
easm21k -MM vectAdd.asm
```

the assembler appends the .DOJ extension to the source file name for the list of dependencies:

```
vectAdd.doj: MakeDepend.h
vectAdd.doj: inits.dat
```

When the -o *filename* option is used with -MM, the assembler outputs the make dependencies list to stdout.

-Mo *filename*

The -Mo (output make rule) assembler switch specifies the name of the make dependencies file, which the assembler generates when you use the -M or -MM switch. If the named file is not in the current directory, you must provide the path name in the double quotation marks ("").

The -Mo *filename* option takes precedence over the -o *filename* option.

-Mt *filename*

The -Mt (output make rule for the named source) assembler switch specifies the name of the source file for which the assembler generates the make rule when you use the -M or -MM switch. If the named file is not in the current directory, you must provide the path name in the double quotation marks ("").

-o [*filename*]

The -o (output) switch directs the assembler to use the specified *filename* argument for the object file. If you omit the switch or its argument, the assembler uses the input file name for the output and appends a .DOJ extension.

You also can use this switch to specify *filename* for the preprocessed assembly file (.IS) — the only file that the preprocessor outputs when the -pp switch is selected.

Some examples of this switch are as follows:

```
easm21k -pp -o test1.is test.asm
        // specify filename for the preprocessed file

easm21k -o "C:\bin\prog3.doj" prog3.asm
        // specify directory for the object file
```

-pp

The `-pp` (proceed with preprocessing only) switch directs the assembler to run the preprocessor, but stop without assembling the source into an object file.

By default, the preprocessor generates an intermediate preprocessed assembly file using the name of the source program and attaching an `.IS` extension to it. When assembling with the `-pp` switch, the `.IS` file is the final result of the assembly.

-r

The `-r` (remove preprocessor output) switch directs the assembler to remove lines that contain the preprocessor output information from the listing file.

-sp

The `-sp` (skip preprocessing) switch directs the assembler to assemble the source into an object file without running the preprocessor. When the assembly skips preprocessing entirely (the `-sp` switch), no preprocessed assembly file (`.IS`) is created.

-v[erbose]

The `-v` or `-verbose` (verbose) switch directs the assembler to display both version and command-line information for each phase of assembly.

-version

The `-version` (display version) switch directs the assembler to display version information for the assembler and preprocessor programs.

-w

The `-w` (disable all warnings) switch directs the assembler not to display warning messages generated during the assembly.

Assembler Syntax Reference

When you develop source program in assembly language, you include preprocessor commands and assembler directives to control the program's processing and assembly. You must follow the assembler rules and conventions of syntax to define symbols (identifiers), expressions, and use different numeric and comment formats.

Software developers who write assembly programs should be familiar with the following topics:

- "Assembler Keywords & Symbols" on page 2-27
- "Assembler Expressions" on page 2-31
- "Assembler Operators" on page 2-32
- "Comment Conventions" on page 2-36
- "Assembler Directives" on page 2-36

Assembler Keywords & Symbols

The assembler supports a set of predefined keywords that includes register and bit-field names, assembly instructions, and assembler directives.

Listing 2-2 lists the assembler keywords. Although the keywords in this listing appear in uppercase, the keywords are case insensitive in the assembler's syntax. For example, the assembler does not differentiate between "`if`" and "`IF`".

Listing 2-2. Assembler Keywords

```
ABS          EX           L9           PC           SUF
AC           EXP          L10          PCSTK        SUFR
ACT          EXP2         L11          PCSTKP       SUI
ADDRESS      EXTERN       L12          PM           SUIR
ALIGN        F0           L13          PMADR        SV
AND          F1           L14          PMBANK1      SZ
ASHIFT       F2           L15          PMDAE        TAG
ASTAT        F3           LA           PMDAS        TCOUNT
AV           F4           LADDR        MPWAIT       TF
B0           F5           LCE          POP          TGL
B1           F6           LCNTR        PORT         TPERIOD
B2           F7           LE           POVLO        TRUE
B3           F8           LEFTMARGIN   POVL1        TRUNC
B4           F9           LEFTO        PRECISION    TST
B5           F10          LEFTZ        PREVIOUS     TYPE
B6           F11          LENGTH       PSA1E        UF
B7           F12          LINE         PSA1S        UI
B8           F13          LN           PSA2E        UNPACK
B9           F14          LOAD         PSA2S        UNTIL
B10          F15          LOG2         PSA3E        UR
B11          FADDR        LOGB         PSA3S        USF
B12          FDEP         LOOP         PSA4E        USFR
B13          FEXT         LR           PSA4S        USI
B14          FILE         LSHIFT       PUSH         USIR
B15          FIX          LT           PX           USTAT1
BB           FLAG0_IN     M0           PX1          USTAT2
BCLR         FLAG1_IN     M1           PX2          UUF
BF           FLAG2_IN     M2           R0           UUFR
BIT          FLAG3_IN     M3           RI           UUI
BITREV       FLOAT        M4           R2           UUIR
BM           FLUSH        M5           R3           VAL
BSET         FMERG        M6           R4           VAR
BTGL         FOREVER      M7           R5           WITH
BTST         FPACK        M8           R6           XOR
BY           FRACTIONAL   M9           R7           SIZE
CA           FTA          M10          R8
CACHE        FTB          M11          R9
CALL         FTC          M12          R10
CH           FUNPACK      M13          R11
CI           GCC_COMPILED M14          R12
CJUMP        GE           M15          R13
```

```
CL          GLOBAL      MANT        R14
CLIP        GT          MAX         R15
CLR         I0          MIN         READ
COMP        I1          MOD         RECIPS
COPYSIGN    I2          MODE1       RFRAME
COS         I3          MODE2       RND
CURLCNTR    I4          MODIFY      ROT
DADDR       I5          MROB        ROUND_MINUS
DB          I6          MROF        ROUND_NEAREST
DEC         I7          MR1B        ROUND_PLUS
DEF         I8          MR1F        ROUND_ZERO
DIM         I9          MR2B        RS
DM          I10         MR2F        RSQRTS
DMA1E       I11         MRB         RTI
DMAIS       I12         MRF         RTS
DMA2E       I13         MS          SAT
DMA2S       I14         MV          SCALB
DMADR       I15         MBM         SCL
DMBANK1     IDLEI15     NE          SE
DMBANK2     IDLE16      NEWPAGE     SECTION
DMBANK3     IF          NOFO        SEGMENT
DMWAIT      IMASK       NOFZ        SET
DO          IMASKP      NOP         SF
DOVL        INC         NOPSPECIAL  SI
EB          IRPTL       NOT         SIN
ECE         JUMP        NU          SIZE
EF          L0          OR          SQR
ELSE        L1          P20         SR
EMUCLK      L2          P24         SSF
EMUCLK2     L3          P32         SSFR
EMUIDLE     L4          PASS        SSI
EMUN        L5          P40         SSIR
ENDEF       L6          PACK        ST
ENDSEG      L7          PAGE        STEP
EOS         L8          PAGELENGTH  STKY
EQ          L9          PAGEWIDTH   STS
WEAK
```

Although the keywords in this list appear in uppercase, the keywords are case-insensitive in the assembler's syntax. For example, the assembler does not differentiate between "`DM`" and "`dm`".

You extend this set of keywords with symbols that declare sections, variables, constants, and address labels. When defining symbols in assembly source code, follow these conventions:

- Define symbols that are unique within the file in which they are declared.

  If you use a symbol in more than one file, use the `.GLOBAL` directive to export the symbol from the file in which it is defined. Then use the `.EXTERN` directive to import the symbol into the other files.

- Begin symbols with alphabetic characters.

  Symbols can use the alphabetic characters (`A–Z` and `a–z`), digits (`0–9`), and special characters `$` and `_` (dollar sign and underscore).

  Symbols are case-sensitive, so `input_addr` and `INPUT_ADDR` define unique variables.

- Do not use a reserved keyword to define a symbol.

  The ADSP-21xxx DSPs assembler's reserved keywords are shown in Listing 2-2.

- Match source and LDF sections' symbols.

  Ensure that .`SECTION`s' name symbols do not conflict with the linker's keywords in the Linker Description File (`.LDF`). The linker uses sections' name symbols to place code in DSP memory. For more details, see the *VisualDSP++ Linker & Utilities Manual for ADSP-21xxx DSPs*.

  Ensure that .`SECTION`s' name symbols do not begin with the "`.rela.`" string reserved by the assembler to form relocatable sections.

- Use a colon (:) to terminate address label symbols.

  Address label symbols may appear at the beginning of an instruction line or stand alone on the preceding line.

The following disassociated lines of code demonstrate symbol usage:

```
.VAR xoperand;          // xoperand is a data variable
.VAR input_array[10];   // input_array is a data buffer
 sub_routine_1:         // sub_routine_1 is a label
.SECTION/PM seg_pmco;   // seg_pmco is a pm section name
```

Assembler Expressions

The assembler can evaluate simple expressions in source code. The assembler supports two types of expressions:

- Constant expressions

  A constant expression is acceptable wherever a numeric value is expected in an assembly instruction or in a preprocessor command syntax. Constant expressions contain an arithmetic or logical operation on two or more numeric constants, for example:

  ```
  2.9e-5 + 1.29
  (128 - 48) / 3
  0x55 & 0x0f
  ```

- Address expressions

  Address expressions contain a symbol + or - an integer constant, for example:

  ```
  data - 8
  data_buffer + 15
  strtup + 2
  ```

Symbols in this type of expression are data variables, data buffers, and program labels. Adding or subtracting an integer from a symbol defines an offset from the address the symbol represents.

Assembler Operators

Table 2-4 lists the assembler's numeric and bitwise operators used in constant expressions and address expressions. These operators are listed in the order they are processed while the assembler evaluates your expressions. Note that assembler limits operators in address expressions to addition and subtraction.

Table 2-4. Operator Precedence Chart

Operator	Description
(*expression*)	*expression* in parenthesis evaluates first
~ -	Ones complement Unary minus
* / %	Multiply Divide Modulus
+ -	Addition Subtraction
<< >>	Shift left Shift right

Table 2-4. Operator Precedence Chart (Cont'd)

Operator	Description
< <= > >=	Less than. Less than or equal Greater than Greater than or equal
== !=	Equal Not equal
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
&&	Logical AND
||	Logical OR

The assembler also supports special "length of" and "page of" operators. Table 2-5 lists and describes these operators used in constant and address expressions.

Table 2-5. Special Assembler Operators

Operator	Description
symbol	Address pointer to *symbol*.
LENGTH(*symbol*)	Length of *symbol* in words.
PAGE(*symbol*)	Most significant 8 address bits associated with *symbol*.

The “length of” and “page of” expressions can be used with external symbols — apply these special operators to symbols that are defined in other sections as .GLOBAL symbols.

The following code determines the base address and length of the circular buffer real_data. The buffer’s length value (contained in L5) determines when addressing wraps around to the top of the buffer.

```
.SECTION/DM seg_dmda;         // data segment
.VAR real_data[n];            // n=number of input samples
…

.SECTION/PM seg_pmco;         // code segment
   B5=real_data;              // buffer base address
                              // I5 loads automatically
   L5=length(real_data);      // buffer’s length
   M6=1;                      // post-modify I5 by 1
   LCNTR=length(real_data), DO loop UNTIL LCE;
                              // loop counter=buffer’s length
    F0=DM(I5,M6);             // get next sample
    …
loop:  …
```

Although the ADSP-21xxx assembler accepts the source code written with the legacy @ operator, we recommend to use LENGTH() in place of @.

Numeric Formats

The assembler supports binary, octal, decimal, and hexadecimal numeric formats (bases) within expressions and assembly instructions.

Table 2-6 describes the conventions of notation the assembler uses to distinguish between numeric formats.

Table 2-6. Numeric Formats

Convention	Description
0x*number* H#*number* h#*number*	A "0x", "H#", or "h#" prefix indicates a hexadecimal *number*.
B#*number* b#*number*	A "B#" or "b#" prefix indicates a binary *number*.
D#*number* d#*number* *number*	A "#D", "#d", or no prefix indicates a decimal *number*.
O#*number* o#*number*	A "#O" or "#o" prefix indicates an octal *number*.

If any operand in an expression is a floating-point value, the assembler stores the value of the expression in floating-point format.

Comment Conventions

The assembler supports C- and C++-style formats for inserting comments in assembly code. Table 2-7 lists and describes assembler comment formats. Note that the assembler does not allow nested comments.

Table 2-7. Comment Conventions

Convention	Description
`/* comment */`	A "`/* */`" string encloses a multiple-line *comment*.
`// comment`	A pair of slashes "`//`" begin a single-line *comment*.

Assembler Directives

Directives in an assembly source file control the assembly process. Unlike instructions, directives do not produce opcodes during assembly. Use the following general syntax for the assembler directives:

```
.directive [/qualifier|arguments];
```

Each assembler directive starts with a period (`.`) and ends with a semicolon (`;`). Some directives take qualifiers and arguments. A directive's qualifier immediately follows the directive and is separated by a slash (`/`); arguments follow qualifiers. Comments may follow the directive's terminating semicolon.

Assembler directives can be uppercase or lowercase. Using uppercase distinguishes directives from other symbols in your source code.

The ADSP-21xxx assembler supports directives listed in Table 2-8. A description of each directive appears in the following sections.

Table 2-8. Assembler Directives

Directive	Description
.ALIGN (see page 2-39)	Specifies a word alignment requirement.
.ENDSEG (see page 2-59)	Marks the end of a section. This correspond to the legacy directive .SEGMENT.
.EXTERN (see page 2-42)	Allows reference to a global symbol.
.FILE (see page 2-43)	Overrides *filename* given on the command line. Used by C compiler.
.GLOBAL (see page 2-44)	Changes a symbol's scope from local to global.
.LEFTMARGIN (see page 2-45)	Defines the width of the left margin of a listing.
.LIST (see page 2-46)	Starts listing of source lines. Default is ON.
.LIST_DATFILE (see page 2-47)	Starts listing of data initialization files. Default is OFF.
.LIST_DEFTAB (see page 2-48)	Sets the default tab width for listings. Default is 4.
.LIST_LOCTAB (see page 2-49)	Sets the local tab width for listings. Default is 0 (set to LIST_DEFTAB).
.LIST_WRAPDATA (see page 2-50)	Starts wrapping opcodes that don't fit listing column.
.NEWPAGE (see page 2-51)	Inserts a page break in a listing. Default is OFF.
.NOLIST (see page 2-46)	Stops listing of source lines.
.NOLIST_DATFILE (see page 2-47)	Stops listing of data initialization files.
.NOLIST_WRAPDATA (see page 2-50)	Stops wrapping opcodes that don't fit listing column.

Table 2-8. Assembler Directives (Cont'd)

Directive	Description
.PAGELENGTH (see page 2-52)	Defines the length of a listing.
.PAGEWIDTH (see page 2-53)	Defines the width of a listing.
.PORT (see page 2-54)	Legacy directive. Declares a memory-mapped I/O port.
.PRECISION (see page 2-55)	Defines the number of significant bits in a floating-point value.
.PREVIOUS (see page 2-56)	Reverts to a previously described .SECTION.
.ROUND_NEAREST (see page 2-57)	Specifies the Round-to-Nearest mode.
.ROUND_MINUS (see page 2-57)	Specifies the Round-to-Negative Infinity mode.
.ROUND_PLUS (see page 2-57)	Specifies the Round-to-Positive Infinity mode.
.ROUND_ZERO (see page 2-57)	Specifies the Round-to-Zero mode.
.SECTION (see page 2-59)	Marks the beginning of a section.
.SEGMENT (see page 2-61)	Legacy directive. Replaced with the .SECTION directive.
.TYPE (see page 2-62)	Changes the default data type of a symbol. Used by C compiler.
.VAR (see page 2-63)	Defines and initializes data objects.

.ALIGN, Specify an Address Alignment

The .ALIGN directive forces the address alignment of an instruction or data item within the .SECTION it is used.

The .ALIGN directive uses the following syntax:

```
.ALIGN expression;
```

where:

- *expression* evaluates to an integer. The *expression* specifies the word alignment requirement; its value must be a power of 2. When aligning a data item or instruction, the assembler adjusts the address of the current location counter to the next address so it can be evenly divided by the value of *expression*, or aligned. The expression set to 0 or 1 signifies no address alignment requirement.

In the absence of the .ALIGN directive, the default address alignment is set to 1, i.e. single-word alignment.

Example:

```
…
.ALIGN 1;    // no alignment requirement
…
.SECTION/DM seg_dmda;
.ALIGN 2;
.VAR single;
           /* aligns the data item in DM on the word boundary,
           at the location with the address value that can be
           evenly divided by 2 */
.ALIGN 4;
.VAR samples1[100]=”data1.dat”;
           /* aligns the first data item in DM on the double-
           word boundary, at the location with the address
           value that can be evenly divided by 4; advances
           other data items consequently */
```

INPUT_SECTION_ALIGN() Linker Instruction

The INPUT_SECTION_ALIGN(*#number*) instruction is used by the linker in the Linker Description File to align the input sections (instructions or data) specified within an output section.

The INPUT_SECTION_ALIGN() operator uses the following syntax:

```
INPUT_SECTION_ALIGN(address_boundary_expession)

INPUT_SECTIONS(filename(input section name)
```

address_boundary_expession aligns the input section to the next word boundary. The expression must be a power of 2. Legal values for this expression depend on the word size of the segment that receive the output section being aligned.

The following example shows how to align input sections using the Linker Description File:

```
SECTIONS
{
dxe_pmco
{                                                                {
       INPUT_SECTION_ALIGN(4)

          INPUT_SECTIONS( a.doj(seg_pmco))
          INPUT_SECTIONS( b.doj(seg_pmco))
          INPUT_SECTIONS( c.doj(seg_pmco))

       INPUT_SECTION_ALIGN(1)
       // end of alignment directive for input sections

          // The following sections will not be aligned
          INPUT_SECTIONS( d.doj(seg_pmco))
          INPUT_SECTIONS( e.doj(seg_pmco))

   } >mem_pmco
}
```

Here the input sections specified after `INPUT_SECTION_ALIGN(4)` but before the `INPUT_SECTION_ALIGN(1)` instructions are aligned. However, the input sections from `d.doj` and `e.doj` are not aligned as the `INPUT_SECTION_ALIGN(1)` instruction indicates the end of the alignment directive.

The `INPUT_SECTION_ALIGN()` operator is valid only within the scope of an output section the Linker Description File. This command gives you the flexibility to align the input sections as needed.

Refer to the *VisualDSP++ Linker & Utilities Manual for ADSP-21xxx DSPs* for more information.

.EXTERN, Refer to a Globally Available Symbol

The .EXTERN directive imports symbols that have been declared as .GLOBAL in other files. For information on the .GLOBAL directive, see page 2-44.

The .EXTERN directive uses the following syntax:

```
.EXTERN symbolName1[, symbolName2, …];
```

where:

- *symbolName* is the name of a global symbol to import. A single .EXTERN directive can reference any number of separated by commas symbols on one line.

Example:

```
.EXTERN coeffs; // This code declares “coeffs” as external,
                // meaning that it was declared as .GLOBAL in
                // another file and it is referenced in this
                // file.
```

.FILE, Override the Name of an Object File

The `.FILE` directive overrides the name of an object file specified with the `-o` *`filename`* command-line switch. This directive may appear in the C/C++ compiler-generated assembly source file (`.S`). The `.FILE` directive is used to ensure that the debugger has the correct file name for a symbol table. This directive is added in connection with overlay linking to enable overriding of the filename given on the command line.

This directive uses the following syntax:

```
.FILE "filename.ext";
```

where:

- *`filename`* is the name the assembler applies to the object file. The argument is enclosed in double quotes.

Example:

```
.FILE "vect.c";    // the argument may be a *.c file
.SECTION/DM seg_dmda;
   …
   …
```

.GLOBAL, Make a Symbol Globally Available

The .GLOBAL directive changes the scope of a symbol from local to global, making the symbol available for reference in object files that are linked with the current one.

By default, a symbol is valid only in the file in which it is declared. Local symbols in different files can have the same name, and the assembler considers them to be independent entities. Global symbols are recognizable in other files and refer to the same address and value. You change the scope of a symbol with the .GLOBAL directive. Once the symbol is declared global, other files may refer to it with .EXTERN. For more information on the .EXTERN directive, see page 2-42.

The .GLOBAL directive uses the following syntax:

```
.GLOBAL symbolName1[, symbolName2,…];
```

where:

- *symbolName* is the name of a global symbol. A single .GLOBAL directive may define the global scope of any number of symbols, separated by commas, on one line.

Example:

```
.VAR coeffs[10];       // declares a buffer
.VAR taps=100;         // declares a variable
.GLOBAL coeffs, taps;  // makes the buffer and the variable
                       // visible in other files
```

LEFTMARGIN, Set the Margin Width of a Listing File

The .LEFTMARGIN directive sets the margin width of the listing page. It specifies the number of empty spaces at the left margin of the listing file (.LST), which the assembler produces when you use the -l switch. In the absence of the .LEFTMARGIN directive, the printer advances 5 empty spaces for the left margin.

The .LEFTMARGIN directive uses the following syntax:

```
.LEFTMARGIN expression;
```

where:

- *expression* evaluates to an integer from 1 to 72. The *expression* value cannot exceed 72, the maximum number of columns per printed page. To change the default setting for the entire listing, place the .LEFTMARGIN directive at the beginning of your assembly source file.

Example:

```
.LEFTMARGIN 9;  /* the listing line begins at column 10. */
```

You can set the margin width only once per source file. If the assembler encounters multiple occurrences of the .LEFTMARGIN directive, it ignores all of them except the last.

.LIST/.NOLIST, Listing Source Lines and Opcodes

The .LIST/.NOLIST directives (ON by default) turn the listing of source lines and opcodes on and off.

If .NOLIST is in effect, no lines in the current source, or any nested source, will be listed until a .LIST directive is encountered in the same source, at the same nesting level. The .NOLIST directive operates on the next source line, so that the line containing ".NOLIST" will appear in the listing (and thus account for the missing lines).

The .LIST/.NOLIST directives use the following syntax:

```
.LIST;

.NOLIST;
```

These directives can appear multiple times anywhere in a source file, and their effect depends on their location in the source file.

.LIST_DATFILE/.NOLIST_DATFILE, Listing Data Initialization Files

The `.LIST_DATFILE/.NOLIST_DATFILE` directives turn the listing of data initialization files on or off. Default setting is OFF.

Nested source files inherit the current setting of this directive pair, but a change to the setting made in a nested source file will not affect the parent source file.

The `.LIST_DATFILE/.NOLIST_DATFILE` directives use the following syntax:

```
.LIST_DATFILE;

.NOLIST_DATFILE;
```

These directives can appear multiple times anywhere in a source file, and their effect depends on their location in the source file. They are used in assembly source files, and not in data initialization files.

.LIST_DEFTAB, Set the Default Tab Width for Listings

Tab characters in source files are expanded to blanks in listing files under the control two internal assembler parameters that set the tab expansion width. The default tab width is normally in control, but it can be overridden if the local tab width is explicitly set with a directive.

The `.LIST_DEFTAB` directive sets the default tab width, and the `.LIST_LOCTAB` directive sets the local tab width (see page 2-49).

Both the default tab width and the local tab width can be changed any number of times via the `.LIST_DEFTAB` and `.LIST_LOCTAB` directives. The default tab width is inherited by nested source files, but the local tab width only affects the current source file.

The `.LIST_DEFTAB` directive uses the following syntax:

```
.LIST_DEFTAB expression;
```

where:

- *`expression`* evaluates to an integer greater than or equal to 0. A value of 0 sets the default tab width to the default tab width.

In the absence of a `.LIST_DEFTAB` directive, the default tab width defaults to 4.

Example:

```
// Tabs here are expanded to the default of 4 columns
.LIST_DEFTAB 8;
// Tabs here are expanded to 8 columns
.LIST_LOCTAB 2;
// Tabs here are expanded to 2 columns
// But tabs in "include_1.h" will be expanded to 8 columns
#include "include_1.h"
.LIST_DEFTAB 4;
// Tabs here are still expanded to 2 columns
// But tabs in "include_2.h" will be expanded to 4 columns
#include "include_2.h"
```

.LIST_LOCTAB, Set the Local Tab Width for Listings

Tab characters in source files are expanded to blanks in listing files under the control of two internal assembler parameters that set the tab expansion width. The default tab width is normally in control, but it can be overridden if the local tab width is explicitly set with a directive.

The `.LIST_LOCTAB` directive sets the local tab width, and the `.LIST_DEFTAB` directive sets the default tab width (see page 2-48).

Both the default tab width and the local tab width can be changed any number of times via the `.LIST_DEFTAB` and `.LIST_LOCTAB` directives. The default tab width is inherited by nested source files, but the local tab width only affects the current source file.

The `.LIST_LOCTAB` directive uses the following syntax:

```
.LIST_LOCTAB expression;
```

where:

- *`expression`* evaluates to an integer greater than or equal to 0. A value of 0 sets the local tab width to the current setting of the default tab width.

In the absence of a `.LIST_LOCTAB` directive, the local tab width defaults to the current setting for the default tab width.

Example: See the `.LIST_DEFTAB` example on page 2-48.

.LIST_WRAPDATA/.NOLIST_WRAPDATA

The .LIST_WRAPDATA/.NOLIST_WRAPDATA directives control the listing of opcodes that are too big to fit in the opcode column. These directives are off by default.

This directive pair actually applies to any opcode that won't fit, but in practice such a value will almost always be data (alignment directives can also result in large opcodes).

- If .LIST_WRAPDATA is in effect, the opcode value is wrapped so that it fits in the opcode column (resulting in multiple listing lines).
- If .NOLIST_WRAPDATA is in effect, the printout has only as much as fits in the opcode column. Nested source files inherit the current setting of this directive pair, but a change to the setting made in a nested source file will not affect the parent source file.

The .LIST_WRAPDATA/.NOLIST_WRAPDATA directives use the following syntax:

```
.LIST_WRAPDATA;
.NOLIST_WRAPDATA;
```

These directives can appear multiple times anywhere in a source file, and their effect depends on their location in the source file.

.NEWPAGE, Insert a Page Break in a Listing File

The `.NEWPAGE` directive inserts a page break in the printed listing file, which the assembler produces when you use the `-l` switch. The assembler inserts a page break at the location of the `.NEWPAGE` directive.

The `.NEWPAGE` directive uses the following syntax:

```
.NEWPAGE;
```

This directive may appear anywhere in your source file. In the absence of the `.NEWPAGE` directive, a page is ejected after listing 66 lines.

.PAGELENGTH, Set the Page Length of a Listing File

The .PAGELENGTH directive controls the page length of the listing file (.LST), which the assembler produces when you use the -l switch.

The .PAGELENGTH directive uses the following syntax:

```
.PAGELENGTH expression;
```

where:

- *expression* evaluates to an integer from 1 to 66. It specifies the number of text lines per printed page. In the absence of the .PAGELENGTH directive, the listing file prints 66 lines per page. To format the entire listing, place the .PAGELENGTH directive at the beginning of your assembly source file.

Example:

```
.PAGELENGTH 50;   // starts a new page
                  // after printing 50 lines
```

You can set the page length only once per source file. If the assembler encounters multiple occurrences of the .PAGELENGTH directive, it ignores all of them except the last.

.PAGEWIDTH, Set the Page Width for a Listing File

The .PAGEWIDTH directive sets the page width of the listing file (.LST), which the assembler produces when you use the -l switch.

The .PAGEWIDTH directive uses the following syntax:

```
.PAGEWIDTH expression;
```

where:

- *expression* evaluates to an integer from 1 to 72. It specifies the maximum number of characters per row in the printed output. In the absence of the .PAGEWIDTH directive, a new line begins after 72 characters are printed on the preceding line. To change the default number of characters per line in the entire listing, place the .PAGEWIDTH directive at the beginning of your assembly source file.

Example:

```
.PAGEWIDTH 36; // starts a new line after 36
               // characters are printed on one line
```

You can set the page width only once per source file. If the assembler encounters multiple occurrences of the .PAGEWIDTH directive, it ignores all of them except the last.

.PORT, Legacy Directive

The .PORT legacy directive assigns port name symbols to I/O ports. Port name symbols are global symbols; they correspond to memory-mapped I/O ports defined in the Linker Description File (.LDF).

The .PORT directive uses the following syntax:

```
.PORT portName;
```

where:

- *portName* is a globally available port symbol.

Example:

```
.PORT p1;    // declares I/O port p1
.PORT p2;    // declares I/O port p2
```

To declare a port using the ADSP-21xxx assembler syntax, use the .VAR directive (for port-identifying symbols) and the Linker Description File (for corresponding I/O sections). The linker resolves port symbols in the LDF. For more information on the LDF, see the *Linker & Utilities Manual for ADSP-21xxx DSPs.*

PRECISION, Select Floating-Point Precision

The .PRECISION directive controls how the assembler interprets floating-point numeric values in constant declarations and variable initializations. Note that you configure the floating-point precision of the target DSP system by setting up control registers with instructions that specific to the processor core.

Use one of the following options:

```
.PRECISION [=] 32;

.PRECISION [=] 40;
```

where:

- The precision of 32 or 40 specifies the number of significant bits for floating-point data. The equal sign (=) following the .PRECISION keyword is optional.

Example:

```
.PRECISION=32; /* Selects standard IEEE 32-bit
                  single-precision format; this is the default
                  setting */

.PRECISION 40; /* Selects standard IEEE 40-bit format with
                  extended mantissa */
```


.PRECISION applies only to floating-point data. Precision of fixed-point data is determined by the number of digits specified.

.PREVIOUS, Revert to Previously Defined Section

The `.PREVIOUS` directive instructs the assembler to set the current section in program memory or data memory to the section that has been described directly before the current one.

This directive uses the following syntax:

```
.PREVIOUS;
```

Example:

```
.SECTION/PM sec_one;
 …     // data & instructions

.SECTION/DM sec_two;
 …     // data

.PREVIOUS;
…      // data & instructions
```

directs the assembler to revert back to `sec_one` and has the same effect as:

```
.SECTION/PM sec_one;
 …     // data & instructions

.SECTION/DM sec_two;
 …     // data

.SECTION/PM sec_one;
…      // data & instructions
```

.ROUND, Select Floating-Point Rounding

The .ROUND_ directive controls how the assembler interprets literal floating-point numeric data after .PRECISION is defined. The .PRECISION directive determines the number of bits to be truncated to match the number of significant bits.

The .ROUND_ directive determines how the assembler handles floating-point values in constant declarations and variable initializations. You configure floating-point rounding modes of the DSP system by setting up control registers with the instructions specific to the target processor. The .ROUND_ directive uses the following syntax:

```
.ROUND_mode;
```

where:

- The *mode* string specifies the rounding scheme used to fit a value in the destination format. Use one of the following IEEE standard modes:

  ```
  .ROUND_NEAREST;
  .ROUND_PLUS;
  .ROUND_MINUS;
  .ROUND_ZERO;
  ```

In the following examples, the numbers with four decimal places are reduced to three decimal places and are rounded accordingly:

```
.ROUND_NEAREST;
     /* Selects Round-to-Nearest scheme; this is the default
        setting.
        A 5 is added to the digit that follows the third
        decimal digit (the least significant bit - LSB). The
        result is truncated after the third decimal digit (LSB).

        1.2581 rounds to 1.258
        8.5996 rounds to 8.600
```

```
        -5.3298 rounds to -5.329
        -6.4974 rounds to -6.496
    */
.ROUND_ZERO;
    /* Selects Round-to-Zero. The closer to zero value is
       taken.
       The number is truncated after the third decimal digit
       (LSB).

       1.2581 rounds to 1.258
       8.5996 rounds to 8.599
       -5.3298 rounds to -5.329
       -6.4974 rounds to -6.497
    */

.ROUND_PLUS;
    /* Selects Round-to-Positive Infinity. The number rounds
       to the next larger.
       For positive numbers, a 1 is added to the third decimal
       digit (the least significant bit). Then the result is
       truncated after the LSB.
       For negative numbers, the mantissa is truncated after
       the third decimal digit (LSB).

       1.2581 rounds to 1.259
       8.5996 rounds to 8.600
       -5.3298 rounds to -5.329
       -6.4974 rounds to -6.497
    */

.ROUND_MINUS;
    /* Selects Round-to-Negative Infinity. The value
       rounds to the next smaller.
       For negative numbers, a 1 is subtracted from the
       third decimal digit (the least significant bit).
       Then the result is truncated after the LSB.
       For positive numbers, the mantissa is truncated
       after the third decimal digit (LSB).

       1.2581 rounds to 1.258
       8.5996 rounds to 8.599
       -5.3298 rounds to -5.330
       -6.4974 rounds to -6.498
    */
```

.SECTION, Declare a Memory Section

The .SECTION directive marks the beginning of a program memory section or data memory section, which is an array of contiguous locations in your target DSP program memory or data memory. Statements between .SECTION and the following .SECTION directive or the end-of-file specify the contents of the section.

This directive uses the following syntax:

```
.SECTION/type sectionName [sectionType];
```

where:

- The /*type* keyword maps a section into the DSP memory. This mapping should follow from the chip's memory architecture.The *type* must match the memory type of the input section of the same name used by the LDF to place the section. One of the following types is required for each .SECTION directive:

Table 2-9. Memory and Section Types

Memory/Section Type	Description
PM	Memory that contains instructions and possibly data.
DM	Memory that contains data.
RAM	Random access memory.
ROM	Read only memory.

- The section name symbol, *sectionName*, must contain eight or fewer characters and is case-sensitive. Section names must match the corresponding input section names used by the Linker Description File

to place the section. You can take advantage of the default LDF included in .\21k\ldf subdirectory of the VisualDSP++ installation directory, or you can write your own LDF.

The assembler generates relocatable sections for the linker to fill in the addresses of symbols at link time. The ADSP-21xxx assembler implicitly pre-fix the name of the section with the ".rela." string to form a relocatable section. For example, for the sections named rela.seg_dmda and rela.seg_pmco, the relocation section are .rela.rela.seg_dmda and .rela.rela.seg_pmco respectively. To avoid such an ambiguity, ensure that your sections' names do not begin with ".rela.".

- The *sectionType* parameter is an optional ELF symbol type STT_*. Valid *sectionTypes* are described in the ELF.h header file, which is available from third-party companies.

Example:

```
/* Data section and program section declared below correspond
   to the default LDF's input sections. */

.SECTION/DM seg_dmda;    // data section
…

.SECTION/PM seg_pmco;    // program section
…
```

If you select an invalid qualifier or disregard it entirely, the assembler exits with an error message.

.SEGMENT & .ENDSEG, Legacy Directives

Previous releases of the ADSP-210xx DSP development software used the `.SEGMENT` and `.ENDSEG` directives to define the beginning and end of a section of contiguous memory addresses. Although these directives have been replaced with the `.SECTION` directive, the source code written with `.SEGMENT`/`.ENDSEG` legacy directives is accepted by the ADSP-21xxx DSP assembler.

.TYPE, Change Default Symbol Type

The `.TYPE` directive enables the compiler to change the default symbol type of an object. This directive may appear in the compiler-generated assembly source file (`.S`).

This directive uses the following syntax:

```
.TYPE symbolName, symbolType;
```

where:

- *`symbolName`* is the name of the object, which symbol type the compiler has changed.
- *`symbolType`* is the ELF symbol type `STT_*`. The valid ELF symbol types are listed in the `ELF.h` header file. By default, a label in a code section has the `STT_FUNC` symbol type and a label in a data section has the `STT_OBJECT` symbol type.

Example:

```
.SECTION/PM seg_pmco;
_main:
     .TYPE _main, STT_FUNC;
        // a label in PM section has STT_FUNC type
```

.VAR, Declare a Data Variable or Buffer

The .VAR directive declares and optionally initializes variables and data buffers. A variable uses a single memory location, and a data buffer uses an array of sequential memory locations.

When declaring or initializing variables, be aware of the following:

- A .VAR directive is valid only if it appears within a section.The assembler associates the variable with the memory type of the section in which the .VAR appears.
- Referring to variables and buffers in code before declaring them, leads to syntax errors.
- A single .VAR directive can declare any number of variables or buffers, separated by commas, on one line.
- The size of a variable's initialization value or the size of a buffer's largest initialization value defines the word size for the variable or buffer. The available word sizes are 32-, 40-, and 48-bits. These word sizes equate respectively to 10-, 12-, and 14-digit hexadecimal constant strings.
- The .VAR directive can list initial values in the directive statement or read them from an external file.
- The number of initial values can not exceed the number of variables or buffer locations that you declare.
- The .VAR directive may declare an implicit-size buffer. The number of initialization elements defines *length* of the implicit-size buffer.

The .VAR directive takes one of the following forms:

```
.VAR varName1[,varName2,…];
.VAR varName1,varName2,… = initExpression1, initExpression2,…;
.VAR bufferName[] = initExpression1, initExpression2,…;
.VAR bufferName[] = "fileName";
.VAR bufferName[length] = "fileName";
.VAR bufferName1[length][, bufferName2[length],…];
.VAR bufferName[length] = initExpression1, initExpression2,…;
```

where:

- The user-defined *varName* and *bufferName* symbols identify variables and buffers.
- The *fileName* parameter indicates that the elements of a buffer get their initial values from the *fileName* data file (.DAT). If the initialization file is in the current directory of your operating system, only the filename need be given inside brackets. Otherwise, you specify the directory and the name of the initialization file with the -I switch.
- The ellipsis (…) represents a comma-delimited list of parameters.
- The optional [*length*] parameter defines the length of the associated buffer in words. The number of initialization elements defines *length* of an implicit-size buffer.
- The *initExpressions* parameters set initial values for variables and buffer elements.

The following lines of code demonstrate some `.VAR` directives:

```
.VAR samples[] = 10, 11, 12, 13, 14;
            // declare and initialize an implicit-length buffer

.VAR twiddles[] = "phase.dat";
            // declare an implicit-length buffer and load the
            // buffer with the contents of the phase.dat file

.VAR Ins, Outs, Remains;
            // declare three uninitialized variables

.VAR samples[100]= "inits.dat";
            // declare a 100-location buffer and initialize it
            // with the contents of the inits.dat file;
            // data file is in current directory

.VAR taps=100;
            // declare a variable and initialize the variable
            // to 100

.VAR myPMdata = 0x157001;
            // declare a variable in program memory
```

Initializing from files is useful for loading buffers with data, such as filter coefficients or FFT phase rotation factors that are generated by other programs. The assembler determines how the values are stored in memory when it reads the data files.

.VAR and ASCII String Initialization Support

The assembler supports ASCII string initialization. This allows the full use of the ASCII character set, including digits, and special characters.

String initialization takes one of the following forms:

```
.VAR symbolString[length] = 'initString', 0;

.VAR symbolString[] = 'initString', 0;
```

Note that the number of initialization characters defines *length* of a string (implicit-size initialization).

Example:

```
.var x[13] = ‘Hello world!’, 0;
.var x[] = ‘Hello world!’, 0;
```

The assembler also accepts ASCII characters within comments.

.WEAK, Support a Weak Symbol Definition and Reference

The `.WEAK` directive supports weak binding for a symbol defined as `WEAK`. You use this directive where the symbol is defined, replacing `.GLOBAL`, and instead of `.EXTERN` to make a weak reference.

This directive uses the following syntax:

```
.WEAK symbol;
```

where:

- `symbol` is the user-defined symbol

When reading in object files, the linker does not resolve a reference to a `.WEAK` symbol, leaving it undefined (defined as 0). If in a later step the linker reads in a definition for a global-bound symbol of the same name (`GLOBAL symbol;`), the linker updates the weak entry to a defined address in memory.

When linking library files, the linker does not resolve weak references, instead all unresolved relocations are defined as 0.

Assembler Glossary

Assembler directives — tell the assembler how to process your source code and set up DSP features. Directives let you structure your program into logical section(s) that mirror the memory layout of your target DSP system.

Instruction set — is the set of assembly instructions that work on a specific DSP family. The `easm21k` assembler supports the ADSP-21xxx instruction set, including the ADSP-21160 extensions.

Linker Description File — controls how the linker processes the assembler's output object files into executable programs. For more information, see the *VisualDSP++ 2.0 Linker and Utilities Manual for the ADSP-21xxx DSPs.*

Preprocessor commands — directs the preprocessor to include files, perform macro substitutions, and control conditional assembly. For more information, see "Preprocessor Directives/Commands" on page 3-16.