

2 ASSEMBLER

Overview

The `easm218x.exe` assembler runs from an operating system command line or from VisualDSP++. The assembler processes assembly source, data, 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 let you define and initialize variables, set up hardware features, or identify program's sections* for placement within DSP memory. The assembler uses directives for guidance as it translates a source program into object code.

Assembler switches are specified on the operating system's command line or in the `Assemble` tab of the VisualDSP++'s `Project Options` dialog box. These switches allow you to control the assembly process of source, data, and header files. You also use these switches to select assembler's features, such as search paths, output file names, and macro preprocessing, among others.

This chapter provides information that you need to know when developing and assembling programs for the ADSP-218x family of DSPs.

The chapter contains the following information about the assembler:

- [“Assembler Guide” on page 2-3](#)

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

Overview

- [“Assembler Command-Line Reference” on page 2-15](#)
- [“Assembler Syntax Reference” on page 2-30](#)
- [“Assembler Glossary” on page 2-68](#)

Assembler Guide

The guide section describes the process of developing new programs in the ADSP-218x family DSPs assembly language. The discussion also covers conventions you should follow when upgrading source programs developed under Release 5.1 or Release 6.1.

This section provides information that you need to know when assembling your 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 2-13](#)
- [“Reading a Listing File” on page 2-13](#)
- [“Setting Assembler Options” on page 2-14](#)

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 source file names to identify them as the VisualDSP++ 2.0 assembly source files.

Assemble your source files, either using the assembler's command line or from VisualDSP++. In the default mode of operation, the assembler processes an input file through the listed stages ([Figure 2-1 on page 2-5](#)) 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.



VisualDSP++ 2.0 assembler can process your source programs developed under Release 5.1 or Release 6.1. The assembly of these programs requires an additional processing step described in [“Assembler Enhancements and Legacy Support”](#) on page 3-1.

[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. An assembly source file often contains preprocessor commands, such as `#include`, that cause the preprocessor to include header files (.H) into your source program. The preprocessor's only output, an intermediate source file (.IS), is the assem-

bler's primary input. A binary object (.OBJ) file and an optional listing (.LST) files are final results of the successful assembly.

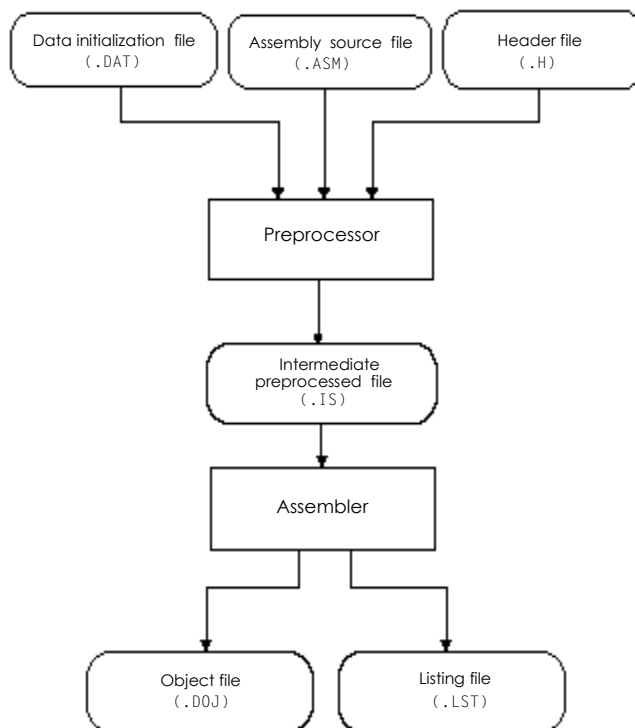


Figure 2-1. Assembler Input and Output Files

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 are as follows:

- Assembly instructions

Instructions follow the DSP's instruction set syntax documented in the DSP's user manuals. Each instruction begins with a keyword and ends it 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 then 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, the assembler treats “outer” and “Outer” as unique labels.

Example:

```
outer: DM(I1,M1)=AR;  
start: AX0=0X1234;
```

- Assembler directives

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

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

Example:

```
.SECTION/DM data1;
.VAR sqrt_coeff[2] = 0x5D1D, 0xA9ED;
```

For complete description of the directive set for the ADSP-218x assembler, see [“Assembler Directives” on page 2-43](#).

- 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 assembly directives, preprocessor commands are case-sensitive and must be lowercase. For a list of the preprocessor commands, see [Table 4-3 on page 4-17](#).

Example:

```
#include "string.h"
#define MAX 100
```

[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 `.DMSEG` and `.PMSEG`). 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 name 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 source programs makes it easier to take advantage of the default Linker Description File included in your VisualDSP++ 2.0 development kit. For more information about the LDF, see the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs*.

Table 2-1. Suggested Input Section Names

<code>.SECTION</code> Name	Description
<code>data1</code>	A section in Data Memory that holds data.
<code>data2</code>	A section in Program Memory that holds data.
<code>program</code>	A section in Program Memory that holds code.

You can 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 live external overlay memory region in your Linker Description File (LDF) for the linker. [Figure 2-2 on page 2-9](#) shows how a program divides into sections that match the program memory and data memory segmentation of a DSP system.

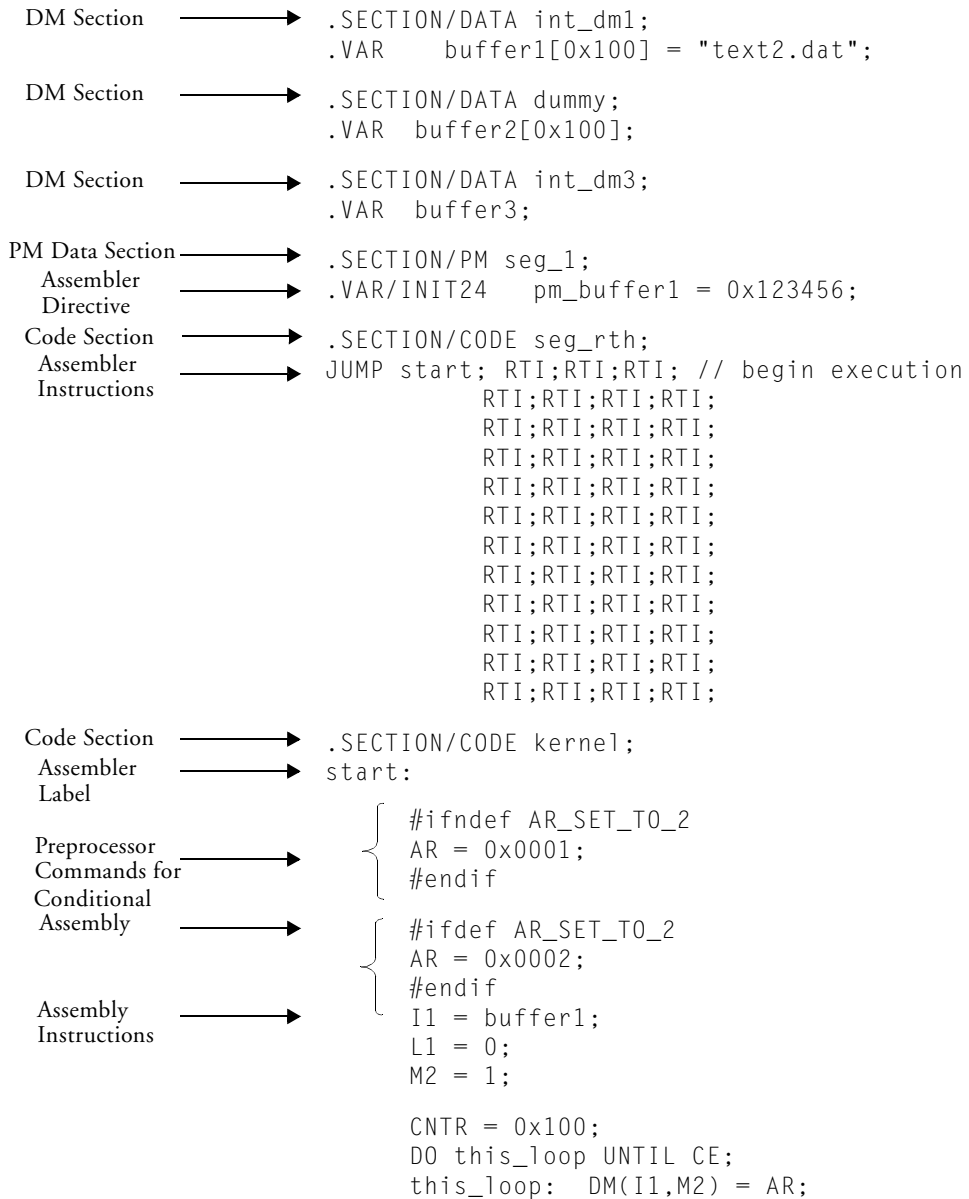


Figure 2-2. Example Assembly Source File

Assembler Guide

The sample assembly program splits into sections; each section begins with a `.SECTION` directive and ends with the occurrence of the next `.SECTION` directive or end-of-file. Legacy tools users should note that there is no `.ENDSECTION` directive.

The source program contains three data and three program sections:

- Data Sections — `int_dm1`, `dummy`, and `int_dm3`. Variables and buffers are declared and can be initialized.
- Program Sections — `seg_1`, `seg_rth`, and `kernel`. Data and instructions, including statements that are needed for conditional assembly, are coded in program memory sections.

Looking at [Figure 2-2 on page 2-9](#), notice that an assembly source often contains preprocessor commands, such as `#include` for inclusions of other files in your source code, `#ifdef` for conditional assembly, or `#define` for macro definitions. The assembler directives, such as `.VAR`, appear within a section to declare and initialize variables.

[Figure 2-3](#) shows a sample user-defined Linker Description File. Looking at the LDF's `SECTIONS{}` command, notice that the input sections' names match the program memory and data memory sections' names used in the assembly program ([Figure 2-3 on page 2-12](#)). The LDF's `SECTIONS{}` command defines the `.SECTION` placements in the ADSP-218x system's physical memory as defined by the LDF `Memory{}` command.

```
ARCHITECTURE(ADSP-2181)
SEARCH_DIR( $ADI_DSP\218x\lib )

//Libraries from the command line are included in COMMAND_LINE_OBJECTS
$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY
{
    seg_rth {TYPE(PM RAM) START(0x00000000) END(0x000000ff) WIDTH(24)}
    kernel {TYPE(PM RAM) START(0x000100) END(0x00fff) WIDTH(24)}
    seg_1  {TYPE(PM RAM) START(0x001100) END(0x03fff) WIDTH(24)}
    int_dm1 {TYPE(DM RAM) START(0x00000) END(0x01fff) WIDTH(16)}
    int_dm3 {TYPE(DM RAM) START(0x02000) END(0x03fff) WIDTH(16)}
}

PROCESSOR p0{

    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
        seg_rth
        {
            INPUT_SECTIONS( $OBJECTS(seg_rth) )
        } >seg_rth

        seg_code
        {
            INPUT_SECTIONS( $OBJECTS(kernel) )
        } >kernel

        seg_code2
        {
            INPUT_SECTIONS( $OBJECTS(seg_1) )
        } >seg_1

        seg_data1
        {
            INPUT_SECTIONS( $OBJECTS(int_dm1) )
        } >int_dm1
    }
}
```

```
    seg_data11
    {
        INPUT_SECTIONS( $OBJECTS(dummy) )
    } >int_dm1

    seg_data3
    {
        INPUT_SECTIONS( $OBJECTS(int_dm3) )
    } >int_dm3
}
}
```

Figure 2-3. Sample Linker Description File

Program Interfacing Requirements

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

To embed (inline) assembly code in your C program, use the `asm()` extension. To link together programs that contain C and assembly routines, use assembly interface macros. These macros facilitate the assembly of mixed routines. For more information about these methods, refer to [“Using Pre-defined Macros” on page 4-6](#).

When you write a C program that interfaces with assembly, you must observe rules that the C compiler follows as it produces code to run on the DSP. These rules for compiled code are called the compiler’s runtime environment. Complying with a runtime environment means following rules for memory usage, register usage, commenting, and variable names.

For information on a particular compiler’s run-time environment, see the compiler’s user manual. The definition of the run-time environment for the ADSP-218x C compiler is provided in the *VisualDSP++ 2.0 C Com-*

piller and Library Manual for ADSP-218x DSPs, which also includes a series of examples to demonstrate how to mix C and assembly code.

Preprocessing a Program

The assembler includes a preprocessor that allows you to use C-style preprocessor commands in your assembly source files. The preprocessor automatically runs before the assembler unless you use the assembler's `-sp` (skip preprocessor) switch. [Table 4-3 on page 4-17](#) lists preprocessor commands and provides a brief description of each command.

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 series of instructions. For example, the macro `MAX` in the example on [page 2-7](#) is replaced with the number `100` during preprocessing.

Note that the preprocessor also automatically removes all comments from assembler-generated optional listing files.

For more information about the preprocessor command set, see [“Preprocessor Commands and Operators” on page 4-17](#).

Reading a Listing File

A listing file (`.LST`) is an optional output text file that lists the results of the assembly process. Listing files provide 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.

Assembler Guide

- **Line** — The third column contains the line number in the assembly source file.
- **Assembly Source** — The fourth column contains the assembly source line from the file.

Note that no comments appear in the listing file. Building your source file with the preprocessor automatically removes all comments. To output comments in the listing output, you must skip preprocessing entirely using the `-sp` switch.

Setting Assembler Options

When developing a DSP project, you may find it useful to modify the assembler's default option settings. The way you set the assembler's options depends on the environment used to run the 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 VisualDSP++, you set the assembler's options in the **Project Options** dialog box of the **Project** menu. For more information on these option settings, see the *VisualDSP++ 2.0 User's Guide for ADSP-21xx DSPs* and online Help.

Assembler Command-Line Reference

The ADSP-218x DSP assembler processes data from assembly source (.ASM, .DSP), data (.DAT), header (.H), and preprocessed (.IS, .APP) files to generate 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 file (.LST) and information in binary Debugging Information Format (DWARF-2) 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 *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs*.

This section provides reference information on the assembler's switches, directives, expressions, and conventions. The reference information available on these topics is as follows:

- [“Assembler Command-Line Interface” on page 2-16](#)

This section describes the assembler's switches, which are accessible from the operating system's command line or from the VisualDSP++.

- [“Listing 2-1 on page 2-32 lists the ADSP-218x assembler keywords.” on page 2-30](#)

This section describes the assembler's predefined keywords and provides rules for user-defined symbols.

Assembler Command-Line Reference

- [“Assembler Expressions” on page 2-35](#)

This section describes the assembler’s numeric and relational operators and other conventions of syntax.

- [“Assembler Directives” on page 2-43](#)

This section describes the assembler’s directives, including syntax and usage examples.

Assembler Command-Line Interface

This section describes VisualDSP++ 2.0 assembler command-line interface and option (switch) 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 4-1](#).

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

```
easm218x [-switch1 [-switch2 ...]] [sourceFile]
```

Where:

- `easm218x` — Name of the assembler program for the ADSP-218x family processors.

- `-switch1, -switch2` — 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 a list of command-line options when run without arguments (same as `-h[elp]`).

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 project 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 a file's extension to determine what operations to perform on the file. [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`, `.DSP`, 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.

Assembler Command-Line Reference

[Table 2-2](#) summarizes file extension conventions that the assembler follows.

Table 2-2. File Name Extension Conventions

Extension	File Description
.asm	Assembly language source file.
.app	Release 5.x or Release 6.x preprocessed assembly source file.
.dsp	Release 5.x or Release 6.x assembly language source file.
.is	Preprocessed assembly source file.
.h	Header file.
.lst	Listing file.
.doj	Assembled object file in ELF/DWARF-2 format.
.dat	Data initialization file.

The assembler command-line switches are case-insensitive.

The following command line, for example:

```
easm218x -2181 -l p1List.lst -DMAX=100 -v -o bin\p1.doj p1.asm
```

runs the ADSP-218x DSP family assembler with:

- 2181 — Assembles instructions unique to ADSP-2181 DSPs.
- l p1List.lst — Directs the assembler to output the listing file.
- DMAX=100 — Defines the constant MAX 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 assembled object file.

`p1.asm` — Names the assembly source file.

The following command line, for example:

```
easm218x -2187 -legacy -c -o test.doj program2.dsp
```

runs the ADSP-218x family DSP assembler with:

`-2187` — Specifies the ADSP-2187L DSPs.

`-legacy` — Processes source code developed under Release 6.1.

`-c` — Preserves original case of assembly symbols.

`-o test.doj` — Renames the assembled object file.

`program2.dsp` — Names the assembly source file of the 6.1 Release.

Assembler Command-Line Options

This section describes the `easm218x` command-line options (switches) in ASCII collation order. A summary of the assembler switches appears in [Table 2-3](#); a brief description of each switch starts on [page 2-21](#).

Table 2-3. `easm218x` Command-Line Switches

Switch Name	Usage Description
<code>-218{1 3 4 5 6 7 8 9}</code>	Assembles instructions unique to the ADSP-218x processors. This switch is required by the assembler.
<code>-Ao filename</code>	Writes RESOVLE() LDF commands for absolute placement to the specified .ldf header file. Same as the absolute placement qualifier /ABS= in Release 5x/6x.
<code>-c</code>	Directs the assembler to preserve the case- sensitive mode.
<code>-dmacro[=definition]</code>	Defines <i>macro</i> .
<code>-flags-pp -switch</code>	Passes the specified switch(s) to the preprocessor.

Assembler Command-Line Reference

Table 2-3. easm218x Command-Line Switches (Cont'd)

Switch Name	Usage Description
-g	Generates debug information (DWARF-2 format).
-h[elp]	Outputs a list of assembler switches.
-i <i>directory</i>	Searches <i>directory</i> for included files.
-ip	Marks data objects for individual placement to reduce memory fragmentation at link time. Overrides consecutive placement. This switch is turned on by default.
-l <i>filename</i>	Outputs named listing file.
-li <i>filename</i>	Outputs named listing, embeds included files.
-legacy	Accepts legacy code.
-M	Makes dependencies only, does not assemble.
-MM	Makes dependencies and assembles.
-Mo <i>filename</i>	Specifies <i>filename</i> for the make dependencies output file.
-Mt <i>filename</i>	Makes dependencies for the specified source file.
-noip	Turns off the -ip switch.
-o <i>filename</i>	Outputs named object file.
-pp	Runs preprocessor only.
-r	Removes preprocessor information from a listing file.
-sp	Assembles without preprocessing.
-v[erbose]	Displays information on each assembly phase.
-version	Displays version info for assembler and preprocessor programs.
-w	Removes all assembler-generated warnings.

Command-Line Reference

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

-218{1|3|4|5|6|7|8|9}

The `-218{1|3|4|5|6|7|8|9}` (ADSP-218x variant) switch directs the assembler to process instructions unique to the specific variant of ADSP-218x DSPs: ADSP-2181, ADSP-2183, ADSP-2184/84L/84N, ADSP-2185/85L/85M/85N, ADSP-2186/2186L/86M/86N, ADSP-2187L/87N, ADSP-2188M/88N, or ADSP-2189M/89N.



`eam218x` requires the processor-specific switch on the command line.

-Ao *filename*

The `-Ao` (absolute placement) switch directs the assembler to write resolve commands to the specified Linker Description File (`.ldf`). This switch is optional for legacy assembly of source programs with absolute placement directives.

The assembler generates an LDF `RESOLVE()` command for each `.VAR/ABS=address` directive in a legacy source program, a program developed under Release 5x/6x. The assembler outputs a resolve `.ldf` header file. The linker inputs the file (if manually referenced with the `INCLUDE()` command) in your project's Linker Description File. You can use a default resolve `.ldf` file name, composed of the 'resolve_' prefix and the `.ldf` extension, or override it with the *filename* argument.

Assembler Command-Line Reference

For example,

```
easm218x -legacy -c cmn.dsp
// generates cmn.doj and resolve_cmn.ldf.

easm218x -legacy -c cmn.dsp -Ao resolve1.ldf
// generates cmn.doj and resolve1.ldf.
```

Note that the `RESOLVE()` commands must be used for global symbols. For more information about the `RESOLVE()` command, see the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs* and [“.VAR/ABS, Place a Variable at the Specified Address” on page 3-26](#).

-c

The `-c` (preserve case) switch directs the assembler to preserve case-sensitive mode.

Previous versions of the assembler software upper-case program symbols by default, whereas the `easm218x` assembler does not. Use the `-c` switch in combination with `-legacy` to preserve the original case of program symbols when re-assembling or linking legacy (coded in the previous version of the assembler language) and/or `easm218x`-written routines.

For example, Release 6.1 assembler processes the `CALL Start;` statement coded in your `p1.dsp` program:

```
asm21 p1.dsp      // produces a relocation against START
asm21 p1.dsp -c   // produces a relocation against Start
```

VisualDSP++ 2.0 assembler re-assembles `p1.dsp`:

```
easm218x p1.dsp -legacy -2181      /* produces a relocation
                                   against START */
```

```
easm218x p1.dsp -legacy -c -2181 /* produces a relocation
                                against Start */
```



VisualDSP++2.0 assembler differentiates between lowercase and uppercase characters by default; Release 6.1 assembler does not.

-dmacro[=def]

The `-D` or `-d` (define macro) switch directs the assembler to define a macro. If you omit 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"
```

-flags-pp -switch [, ...]

The `-flags-pp` (flag the preprocessor) switch directs the assembler to pass each comma-separated switch to the preprocessor. For example, to enclose strings in the double quotation marks, making them compatible with C-style strings, pass the `-cprefix` switch to the preprocessor as follows:

```
easm218x -flags-pp -cprefix -pp CstyleStrings.asm
```

and the preprocessor runs with the `-cprefix` option. For a list of the preprocessor's options, see [Table 4-2 on page 4-9](#).

-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.

Assembler Command-Line Reference

-h[elp]

The `-h` or `-help` switch directs the assembler to output to standard out 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 pathnames for the `-I` switch on the command line (omitting the disk partition) is recommended. For example:

```
easm218x -I "\bin\include\buffer1.dat"
```

-ip

The `-ip` (individual placement) directs the assembler to mark data objects that are eligible for individual placement by the linker. When the `-ip` switch is specified on the linker's command line or in the IDDE, the default behavior of the linker — placing data blocks in consecutive memory addresses — is overridden. `-ip` allows individual placement of a grouping of data in DSP memory, thus reducing memory space fragmentation.

Absolute (`/ABS`) and circular buffer (`/CIRC`) placements take precedence over data/program section placements in contiguous memory locations. When remaining memory space is not sufficient for the entire section placement, the link fails. With `-ip`, you allow the linker to extract a block of data for individual placement and fill in fragmented memory spaces.

For more information about the individual placement option, see the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs*.

-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 instruction opcodes that the assembler produces. If you omit the *filename* argument, the assembler outputs an error message. For more information about listing files, see [“Reading a Listing File” on page 2-13](#).

-li filename

The `-li` (listing with include files) switch directs the assembler to generate the named listing file. Note that `-l` uses the source file to produce the listing, while `-li` uses both the source file and `#include` files. For more information about listing files, see [“Reading a Listing File” on page 2-13](#).

Assembler Command-Line Reference

-legacy

The `-legacy` (accept legacy code) switch directs the assembler to process source programs developed using Release 6.x (and older) assembler software.

The assembler accepts legacy directives listed in [Table 2-9 on page 2-45](#). Note that Release 5x/6x assembler automatically uppercases symbols. To preserve the original case of program symbols, use the `-legacy -c` combination. For more information about `-c`, see [“-c” on page 2-22](#).



You may need to revise source code programs when re-assembling with `eam218x`. Please review any diagnostic or error messages issued during the assembly of Release 6.1 source programs.

For information on how to revise Release 5.x/6x programs to comply with VisualDSP++ assembler syntax, refer to [“Assembler Enhancements and Legacy Support” on page 3-1](#).

-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: 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 `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:

```
easm218x -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 pathname in the double quotation marks (“”).



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

Assembler Command-Line Reference

-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 pathname in the double quotation marks (“”).

-noip

The `-noip` (no individual placement) directs the assembler to disable the default `-ip` option, which directs the assembler to mark data objects eligible for individual placement by the linker.

-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:

```
easm218x -pp -o test1.is test.asm
           // specify filename for the preprocessed file

easm218x -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 file 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 write source program in assembly language, you include pre-processor 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 and Symbols” on page 2-30](#)
- [“Assembler Expressions” on page 2-35](#)
- [“Assembler and Preprocessor Operators” on page 2-36](#)
- [“Numeric Formats” on page 2-39](#)
- [“Comment Conventions” on page 2-42](#)
- [“Assembler Directives” on page 2-43](#)

Assembler Keywords and Symbols

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

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 “DM” and “dm”.

[Listing 2-1 on page 2-32](#) lists the ADSP-218x assembler keywords.

__ADI__ __TIME__	__DATE__	__FILE__	__LINE__	__STDC__
ABS	AC	ADDRESS	AF	ALIGN
AND	AR	AR_SAT	AS	ASHIFT
ASM	ASM_ASSERT	ASTAT	AV	AV_LATCH
AX0	AX1	AY0	AY1	
BIT_REV	BOOT	BR	BY	BYTE
C	CALL	CE	CIRC	CLRBIT
CNTR	CODE	CONST		
DATA	DEFINE	DIS	DIVQ	DIVS
DM	DMSEG	DO	DW	DX
ELIF	ELSE	ENA	END_REPEAT	ENDIF
ENDMACRO	ENDMOD	ENTRY	EQ	ERROR
EXP	EXPADJ	EXPORT	EXTERN	EXTERNAL
FILE	FLO	FL1	FL2	FLAG_OUT
FOREVER				
GE	GLOBAL	GT		
HI				
I0	I1	I2	I3	I4
I5	I6	I7	ICNTL	IDLE
IF	IFDEF	IFNDEF	IMASK	INCLUDE
INDENT	INIT	INIT24	INT	IO
JUMP				
L0	L1	L2	L3	L4
L5	L6	L7	LE	LEFTMARGIN
LENGTH	LINE	LO	LOOP	LSHIFT
LT				
M0	M1	M2	M3	M4
M5	M6	M7	MACRO	M_MODE
MF	MODIFY	MODULE	MR	MRO
MR1	MR2	MSTAT	MV	MX0
MX1	MY0	MY1		
NE	NEG	NEWPAGE	NONE	NOP
NORM	NOT			
OFFSETOF	OWRCNTR			
PAGE	PAGEID	PAGELength	PAGEWIDTH	PASS

Assembler Syntax Reference

PC	PM	PMCODE	PMDATA	PMSEG
POP	PORT	POS	PRAGMA	PREVIOUS
PUSH	PX			
RAM	REPEAT	RESET	RND	ROM
ROUND_MINUS	ROUND_NEAREST	ROUND_PLUS	ROUND_ZERO	RTI
RTS				
SAT	SB	SD	SE	SEG_REG
SEGMENT	SECTION	SEG	SET	SETBIT
SETDATA	SETINT	SHT_DEBUGINFO	SHT_DM	SHT_DYNAMIC
SHT_HASH	SHT_DYNSYM	SHT_LDF	SHT_NOBITS	SHT_NOTE
SHT_NULL	SHT_PMCODE	SHT_PMDATA	SHT_PROSORTYPE	
SHT_PROGBITS	SHT_REL	SHT_RELA	SHT_SEGMENTS	
SHT_SHLIB	SHT_STRTAB	SHT_SYMTAB	SI	SIMIDLE
SIZE	SR	SRO	SR1	SR2
SS	SSTAT	STATIC	STEP	STS
STT_FUNK	STT_OBJECT	SU	SV	
TGLBIT	TIMER	TOGGLE	TRAP	TRUE
TSTBIT	TX0	TX1	TYPE	
UNDEFINE	UNTIL	US	UU	
VAR				
WARNING	WEAK			
XOR				

Listing 2-1. ADSP-218x Assembler Keywords

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.

`'_'` as the first character of an symbol is considered *reserved* by the assembler.

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

The ADSP-218x assembler's reserved keywords are shown in Listing 2-1.

- Match source and LDF section symbols.

Ensure that `.SECTIONS'` 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++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs*.

Ensure that `.SECTIONS'` name symbols do not begin with the `'rel.'` string *reserved* by the assembler to form relocatable sections.

Assembler Syntax Reference

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

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 some symbol usage:

```
.SECTION/DM data1;
.VAR xoperand;
    // xoperand is a 16-bit variable in DM

.SECTION/PM data2;
.VAR/INIT24 input_array[10];
    // input_array is a 24-bit wide data buffer in pm

sub_routine_1:
    // sub_routine_1 is a label

.SECTION/PM kernel;
    // kernel is a section in program memory
```

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. Constant expressions contain an arithmetic or logical operation on two or more numeric constants, including constants of the fractional type:

```
2.9e-5 + 1.29
```

```
(128 - 48) / 3
```

```
0x55 & 0x0f
```

```
7.6r - .8r
```

For information about fraction type support, refer to [“Fractional Type Support” on page 2-39](#).

- Address expressions

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

```
data - 8
```

```
data_buffer + 15
```

```
strdup + 2
```

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

Assembler and Preprocessor Operators

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

Table 2-4. Operator Precedence Chart

Operator	Usage Description
<i>(expression)</i>	<i>expression</i> in parenthesis evaluates first.
<i>~</i>	One's complement.
<i>-</i>	Unary minus.
<i>*</i>	Multiply.
<i>/</i>	Divide (preprocessor only). Ⓢ “/” has replaced “\” of the previous release.
<i>%</i>	Modulus (preprocessor only).
<i>+</i>	Addition.
<i>-</i>	Subtraction.
<i><<</i>	Shift left.
<i>>></i>	Shift right.
<i><</i>	Less than (preprocessor only).
<i><=</i>	Less than or equal (preprocessor only).
<i>></i>	Greater than (preprocessor only).
<i>>=</i>	Greater than or equal (preprocessor only).
<i>==</i>	Equal (preprocessor only).
<i>!=</i>	Not equal (preprocessor only).

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

Operator	Usage Description
&	Bitwise AND (preprocessor only).
	Bitwise inclusive OR.
^	Bitwise exclusive OR (preprocessor only).
&&	Logical AND (preprocessor only).
	Logical OR (preprocessor only).

The assembler also supports special “address of”, “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	Usage Description
ADDRESS(<i>symbol</i>)	Least significant 16 address bits of <i>symbol</i> .
<i>symbol</i>	Address pointer to <i>symbol</i> .
LENGTH(<i>symbol</i>)	Length of <i>symbol</i> in words.
PAGE(<i>symbol</i>)	Most significant 8 address bits associated with <i>symbol</i> .

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

Assembler Syntax Reference

The following example demonstrates how the assembler operators are used to load **L** (length) and **I** (index) registers when setting up circular buffers:

```
.SECTION/DM data1;           // data section
    .VAR real_data[n];       // n = number of input samples
    ...

.SECTION/PM program;         // code section
    I5=real_data;            // buffer's base address

    L5=length(real_data);     // buffer's length
    AR=I5;                    // load address to data register
    M4=1;                     // post-modify I5 by 1
    CNTR=length(real_data);   // set loop counter for n instructions
    DO loop1 UNTIL CE;
    AX0=DM(I5,M4);            // get next sample
    ...
loop1: ...
```

This code fragment initializes **I5** (index) and **L5** to the base address and length, respectively, of the circular buffer `real_data`. The buffer length value contained in **L5** determines when addressing wraps around the top of the buffer. For further information on circular buffers, refer to the *ADSP-218x DSP Hardware Reference Manual*.

Numeric Formats

The assembler supports binary, decimal, hexadecimal, and fractional 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
<i>0xnumber</i>	An “0x” prefix indicates a hexadecimal <i>number</i> .
<i>B#number</i> <i>b#number</i>	A “B#” or “b#” prefix indicates a binary <i>number</i> .
<i>number</i>	No prefix indicates a decimal <i>number</i> .
<i>numberr</i>	An “r” suffix indicates a fractional <i>number</i> .

Fractional Type Support

Fractional (fract) constants are specially marked floating-point constants to be represented in fixed-point. A fract constant uses the floating-point representation with a trailing “r”, where “r” stands for “fract”. The legal range is [-1...1). Fracts are represented as signed values, which means the values must be greater than or equal -1 and less than 1.

Example:

```
.VAR myFracts[] = 0.5r, -0.5e-4r, -0.25e-3r, 0.875r;
/* constants are examples of legal fracts */

.VAR OutOfRangeFract = 1.5r;
/* [Error E37] ...Fract constant '1.5r' is out of range. Fract
constants must be greater than or equal to -1 and less than 1.
*/
```

Assembler Syntax Reference

1.15 Fracts

ADSP-218x supported fracts use 1.15 representation, meaning a sign bit and "15 bits of fraction". This is -1 to $+1-2^{-15}$. For example, 1.15 maps the constant `0.5r` to 2^{-14} .

The conversion formula used for ADSP-218x DSP to convert from the floating-point to fixed-point uses a scale factor of 15:

```
fractValue = (short) (doubleValue * (1 << 15))
```

Example:

```
// Fract output for 0.5r is 0x4000
// sign bit + 15 bits
// 0100 0000 0000 0000
//      4      0      0      0 = 0x4000 = .5r

.var myFract = .5r;

// Fract output for -1.0r is 0x8000
// sign bit + 15 bits
// 1000 0000 0000 0000
//      8      0      0      0 = 0x8000 = -1.0r

.var myFract = -1.0r;
```

1.0r Special Case

`1.0r` is out of the fract range. Specify `0x7FFF` for the closest approximation of `1.0r` within the 1.15 representation.

Fractional Arithmetic

The assembler supports arithmetic operations on fractional constants. As implemented, constant `fract` expressions are consistent with what is provided for other numeric types in the assembler. Doing `fract` constant arithmetic is sometimes necessary when one receives constants and wants to derive others from them.

The internal (intermediate) representation for expression evaluation is a double floating-point value. Fract range checking is deferred until the expression is evaluated.

```
#define some_val 0.875r
.SECTION data1;
.VAR localOne = some_val + 0.005r;
                        // Result .88r is within the legal range
.VAR xyz = 1.5r - .9r; // Result .6r is within the legal range
.VAR abc = 1.5r;      // Error: 1.5r is out of the legal range
```

Mixed Type Arithmetic

The assembler implementation currently supports arithmetic between fracts, but not fracts and integers.

```
.var myFract = 1 - 0.5r;
// Assembler Error: Illegal mixing of types in expression.
```

Comment Conventions

The assembler supports C- and C++-style formats for inserting comments in assembly code. [Table 2-7](#) lists and describes easm218x comment formats. Note that the assembler does not allow nested comments.

Table 2-7. Comment Conventions

Convention	Description
<i>/* comment */</i>	A “/* */” string encloses a multiple-line <i>comment</i> .
<i>// comment</i>	A pair of slashes “//” denote a single-line <i>comment</i> .

The comment conversion utility included in this release provides support for converting legacy code developed under Release 6.1 (or earlier). For information about the comment converter, refer to [“Comment Converter” on page A-1](#).

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 [/qualifiers |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. The ADSP-218x assembler supports directives shown in [Table 2-8](#). A description of each directive appears in the following sections.

The ADSP-218x assembler also supports Release 6.1 (or older) directives shown in [Table 2-9 on page 2-45](#). To re-assemble a program that uses any of the directives listed in [Table 2-9](#) with `eam218x`, use the `-legacy` switch. For more information about the legacy directives and syntax conventions, see [“Assembler Enhancements and Legacy Support” on page 3-1](#).



Future releases of the DSP development tools may not support legacy directives or conventions of syntax. We recommend to write new programs using VisualDSP++ 2.0 directives and conventions of syntax.

Assembler Syntax Reference

Table 2-8. Assembler Directives

Directive	Description
<code>.ALIGN</code> (see page 2-47)	Specifies a byte alignment requirement.
<code>.EXTERN</code> (see page 2-48)	Allows to reference a global symbol.
<code>.FILE</code> (see page 2-49)	Overrides <i>filename</i> given on the command line. Used by C compiler.
<code>.GLOBAL</code> (see page 2-50)	Changes a symbol's scope from local to global.
<code>.LEFTMARGIN</code> (see page 2-51)	Defines the width of the left margin of a listing.
<code>.NEWPAGE</code> (see page 2-52)	Inserts a page break in a listing.
<code>.PAGELength</code> (see page 2-53)	Defines the length of a listing.
<code>.PAGEWIDTH</code> (see page 2-54)	Defines the width of a listing.
<code>.PREVIOUS</code> (see page 2-55)	Reverts to a previously described <code>.SECTION</code> .
<code>.SECTION</code> (see page 2-56)	Marks the beginning of a section.
<code>.TYPE</code> (see page 2-58)	Changes the default <code>STT_*</code> type of a symbol. Used by C compiler.
<code>.VAR</code> (see page 2-59)	Declares and initializes data objects.
<code>.VAR/CIRC</code> (see page 2-64)	Declares and initializes circular buffers.
<code>.VAR/INIT24</code> (see page 2-63)	Declares and initializes 24-bit wide data objects.
<code>.WEAK</code> (see page 2-67)	Supports weak symbol definition and reference

Table 2-9. Release 6.1 Legacy Directives

Release 6.1 Directive	Change for VisualDSP++ 2.0
.CONST (see page 3-8)	#define (see page 4-19)
.DMSEG (see page 3-9)	.SECTION/DM or .SECTION/DATA (page 2-56)
.ENTRY (see page 3-11)	.GLOBAL (see page 2-50)
.EXTERNAL (see page 3-12)	.EXTERN (see page 2-48)
.GLOBAL (see page 2-50)	Modified. You can make any symbol globally available with .GLOBAL.
.INCLUDE (see page 3-13)	#include (see page 4-28)
.INDENT (see page 3-15)	Removed
.INIT (see page 3-16)	.VAR (see page 2-59)
.INIT24 (see page 3-16)	.VAR/INIT24 (see page 2-59)
.LOCAL (see page 3-19)	? (macro label generation) (see page 4-33)
.MACRO/.ENDMACRO (see page 3-21)	#define (see page 4-19)
.MODULE/.ENDMOD (see page 3-23)	.SECTION/PM or .SECTION/CODE (see page 2-56)
.PAGE	Removed. The ADSP-218x processors do not support paging.
.PMSEG (see page 3-9)	.SECTION/PM or .SECTION/CODE (see page 2-56)
.PORT	Removed. You can declare an I/O port using the .VAR and .GLOBAL directive, then create the corresponding section in the Linker Description File.

Assembler Syntax Reference

Table 2-9. Release 6.1 Legacy Directives (Cont'd)

Release 6.1 Directive	Change for VisualDSP++ 2.0
.SEGMENT	.SECTION (see page 2-56)
.SETDATA	.VAR (see page 2-59)

.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 byte 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.

Example:

```
.ALIGN 0;      /* no alignment requirement */
...
.ALIGN 1;      /* no alignment requirement */
...
.SECTION/DM data1;
.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 */
```

.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 2-50](#).

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 an external symbol
                // to reference the global symbol
                // declared in the example code in the .GLOBAL
                // directive description (page 2-50).
```

.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 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 data1;
...
...
```

.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 default scope 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 2-48](#).

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 a 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.

.NEWPAGE, Insert a Page Break in a Listing File

The `.NEWPAGE` directive inserts a page break in the printed listing file (`.LST`), 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 of 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 the 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.

.PREVIOUS, Revert to the 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 and instructions

.SECTION/DM sec_two;
...      // data

.PREVIOUS;
...      // data and instructions
```

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

```
.SECTION/PM sec_one;
...      // data and instructions

.SECTION/DM sec_two;
...      // data

.SECTION/PM sec_one;
...      // data and instructions
```

.SECTION, Declare a Memory Section

The `.SECTION` directive marks the beginning of a logical section that mirrors an array of contiguous locations in program memory or data memory of the ADSP-218x system. Statements between `.SECTION` and the following `.SECTION` directive or the end-of-file comprise the contents of the section.

With `.SECTION`, you have advantage of having multiple code sections in a source file and control over data placement. The Linker Description File defines placement of sections in the DSP memory. In addition, you can interpret the results of the assembly process via the `elfdump` utility. This utility is included in the DSP development kit and allows you to view the section's size and variable placement.

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 Linker Description File to place the section. One of the following types is required for each `.SECTION` directive:

Memory Type	Description
PM or CODE	Memory that contains instructions and possibly data.
DM or DATA	Memory that contains data.

- The section name symbol, `sectionName`, is not limited in length 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 Linker Description File included in the `\218x\ldf` subdirectory of the VisualDSP++ installation directory, or you can write your own LDF.

Unless you specify absolute section placement (via `/ABS=address`), the assembler generates relocatable sections for the linker to fill in the addresses of symbols at link time. The ADSP-218x assembler implicitly pre-fixes the name of the section with the `".rel."` string to form a relocatable section. For example, for the sections named `rel_seg_dmda` and `rel_seg_pmco`, the relocation section are `.rel.rel_seg_dmda` and `.rel.rel_seg_pmco` respectively. To avoid such an ambiguity, ensure that your sections' names do not begin with `".rel."`.

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

For more information on the ELF file format, see the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs*.

Example:

```
/* Declared below data and program sections correspond to the
   default LDF's input sections. */

.SECTION/DM data1;           // data memory section
...

.SECTION/CODE program;       // program memory section
...
```



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

.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_*`. Valid ELF symbol types are listed in the table below.

ELF Symbol Type	Description
<code>STT_FILE</code>	Filename
<code>STT_FUNC</code>	Code label
<code>STT_OBJECT</code>	Data label
<code>STT_SECTION</code>	Data Memory or Program Memory section

.VAR, Declare a Data Variable or Buffer

A single `.VAR` directive can declare and optionally initialize one or more variables and data buffers. A variable uses a single memory location, and a data buffer uses an array of memory locations.

The `.VAR` directive containing a single declaration takes one of the following forms:

```
.VAR varName;

.VAR = {initExpression1, initExpression2,...}*;

.VAR varName = {initExpression};

.VAR bufName[[length]]† = {initExpression1, initExpression2,...};

.VAR bufName[[length]] = {"fileName.dat"};
```

The assembler requires "{}" for each initializer list on the `.VAR` directive containing more than one symbol declaration:

```
.VAR = {initExpression}, varName2 = {initExpression},...;

.VAR varName1 = {initExpression}, varName2 = {initExpression},...;

.VAR bufName1[[length]] = {initExpression, ...};
    bufName2[[length]] = {initExpression, ...},...;

.VAR bufName1[[length]] = {"fileName.dat"},
    bufName2[[length]] = {"fileName.dat"},...;
```

* For compatibility with previous assembler versions, braces can be omitted on a `.VAR` statement containing one symbol (buffer) declaration.

[†] The inner brackets denote an optional *length* parameter; the outer brackets are required for buffer declarations.

Assembler Syntax Reference

Where:

- User-defined *varName* and *bufName* symbols identify variables and buffers.
- The *fileName.dat* parameter indicates that the elements of a buffer get their initial values from an external data file. If the initialization file is in the current project directory, only the filename need be given inside the 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.

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

- The `.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.
- Initialized variables in program memory and data memory are 16-bit variables by default.
- Uninitialized variables are implicitly initialized to '0'.
- The `.VAR` directive can list initial values in the directive statement or read them from external `.DAT` files. VisualDSP++ 2.0 assembler does not limit a number of data files per one `.VAR` statement.

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.

- The number of initial values may 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 the *length* of the implicit-size buffer.

A `.VAR` directive with more than one symbol declaration becomes a `.VAR` directive block. `.VAR` directive blocks guarantee consecutive placement. Without linker's `-ip` (individual placement option), all symbols declared within a section are guaranteed to be placed in consecutive memory locations. With `-ip` specified at assembly and link time, only symbols listed together within a single `.VAR` directive to be placed consecutively.

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

```
.SECTION/DM data1;

.VAR var2;

.VAR var3, var4;

.VAR lstate1 = { 0x1000 },
    lstate3 = { 0x3000 },
    lstate2[3] = { 0x2000, 0x2001, var2};

.VAR Mstate = { 0x1000 },
    Nstate,
    Ostate[3] = { 0x2000, 0x2001, var2 };

.VAR state1 = 0x1000;

.VAR Xstate1 = { 0x1000 };

.VAR state2[3] = 0x2000, 0x2001, var2;

.VAR Xstate2[3] = { 0x2000, 0x2001, var2 };

.VAR state3[10] = "file.dat";
```

Assembler Syntax Reference

```
.VAR  Zstate3[] = "file.dat";

.VAR  Xstate3[10] = { "file.dat" };

.VAR  Qstate = { 0x1000 },
      Rstate[10] = { "file.dat" },
      Sstate2[3],
      Tstate2[3] = {var2, var3, var4};

myData1:
    .var = 0x5000, 0x5001, 0x5002;

myData2:
    .var = { 0x6000, 0x6001, 0x6002, 0x6004 };

.var  Astate = { 0x1000 },
      Bstate[] = { "file.dat" },
      Cstate[3] = {var2, var3, var4};
```

.VAR/INIT24, Declare a Wide Data Structure

A special case of the `.VAR` directive, `.VAR/INIT24`, allows declaration and initialization of 24-bits wide data structures in program memory sections.

The `.VAR/INIT24` directive takes the following form:

```
.VAR/INIT24 varName [= initExpression];

.VAR/INIT24 varName = {initExpression};

.VAR/INIT24 bufName[[length]] = initExpression1, initExpression2,...,

.VAR/INIT24 bufName[[length]] = {initExpression1, initExpression2,...},
    varName = {initExpression};
```

Example:

```
.SECTION/PM program;
.VAR/INIT24 myPMdata = 0x157001;
    // declare a 24-bit variable in program memory
```

Assembler Syntax Reference

.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!';  
.var x[] = 'Hello world!';
```

The assembler also accepts ASCII characters within comments.

.VAR/CIRC, Declare a Circular Buffer

The assembler supports circular buffer declaration and addressing. This is accomplished with the `/CIRC` qualifier. The `.VAR` directive declares a linear buffer unless the `/CIRC` attribute is applied to define the buffer as circular.

This example declares a relocatable circular data buffer whose length is the value of the constant `taps`:

```
.SECTION/DM data_1;  
.VAR taps=15;  
.VAR/CIRC data_buffer[taps];
```

When multiple buffers are declared on one line and the `/CIRC` qualifier is used, a single circular buffer is created — the individual buffers are simple linear buffers. For example, this declaration creates one 15-word circular buffer:

```
.VAR/CIRC aa[5],bb[5],cc[5];
```

The base address of the circular buffer is `aa`; this is the symbol used to access the buffer in code. The address of `bb` is `aa+5`, and the address of `cc` is `aa+10`. The three five-word buffers can be individually accessed as linear buffers. Since the value 15 requires four bits for binary representation, the circular buffer `aa` is located at an address which is a multiple of sixteen.

The following example uses three `.VAR` directives to declare three different circular buffers:

```
.VAR/CIRC aa[5];
.VAR/CIRC bb[5];
.VAR/CIRC cc[5];
```

This example creates the structure for a sine/cosine lookup table:

```
.VAR/CIRC sin[256],cos[768];
```

A single circular buffer has a length of 1024. To access the buffer in code, you can initialize DAG index registers and buffer length registers with the following instructions:

```
I0=address(cos);
L0=1024;
I1=address(sin);
L1=1024;
```

These instructions load `I0` and `I1` with the base addresses of `cos` and `sin`. The corresponding `L` registers are loaded with the length of the circular buffer to enable wraparound addressing. A circular buffer is only implemented when an `L` register is set to a non-zero value.

Assembler Syntax Reference

The following example demonstrate how the assembler operators are used to load L (length) and I (index) registers when setting up circular buffers:

```
.SECTION/DM data1;           // data section
    .VAR real_data[n];       // n = number of input samples
    ...

.SECTION/PM program;         // code section
    I5=real_data;            // buffer's base address

    L5=length(real_data);    // buffer's length
    AR=I5;                   // load address to data register
    M4=1;                    // post-modify I5 by 1
    CNTR=length(real_data);  // set loop counter for n instructions
    DO loop1 UNTIL CE;
    AX0=DM(I5,M4);           // get next sample
    ...
loop1: ...
```

This code fragment initializes I5 (index) and L5 to the base address and length, respectively, of the circular buffer `real_data`. The buffer length value contained in L5 determines when addressing wraps around the top of the buffer.

For more information on circular buffers, refer to the *ADSP-218x DSP Hardware Reference Manual*.

.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 some DSP features. Directives let you structure your program into a 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 easm218x assembler supports the ADSP-218x DSP instruction set. For information about the 16-bit, fixed-point family DSPs instruction set, see the following publications:

- *ADSP-218x DSP Hardware Reference*
- *ADSP-218x DSP Instruction Set Reference*
- The ADSP-2181, ADSP-2183, ADSP-2184/84L/84N, ADSP-2185/85L/85M/85N, ADSP-2186/2186L/86M/86N, ADSP-2187L/87N, ADSP-2188M/88N, or ADSP-2189M/89N data sheets

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 ADSP-21xx DSPs*.

Preprocessor commands — directs the preprocessor to include files, perform macro substitutions, and control conditional assembly. [For more information, see “Preprocessor Commands and Operators” on page 4-17.](#)