

3 ASSEMBLER ENHANCEMENTS AND LEGACY SUPPORT

Overview

VisualDSP++ 2.0 introduces a new Integrated Development and Debugging Environment, assembler, C compiler, linker, debugger, RTOS Kernel, and a binary file format, ELF. As a result, VisualDSP++ 2.0 has several significant differences from the previous release that you need to be aware of.

This chapter concentrates on the assembler and preprocessor features added since DSP development software Release 6.1 (or older). Of the new features and enhancements, the following have the most impact on your existing projects:

- Some switches have been modified or removed. If you are using any of these options, you must revise your command-line scripts and batch files to rebuild your project.
- Some directives and conventions of syntax have been replaced. These directives and conventions are now referred to as legacy syntax or legacy code. Legacy syntax and the new syntax cannot be used together in the same source file. They can be mixed together within the same project as long as they are assembled in different source files. If you are using any of these statements, we recommend that you revise your source programs or begin a new project under VisualDSP++ 2.0.

Overview

- The new assembler accepts your source code developed with Release 6.1 (and older) assembler software. If you are re-assembling your legacy code program using `easm219x`, select the `-legacy` switch. In some cases you will need to modify your existing source to use the new assembler.
- The Architecture File (.ACH) is no longer supported. If you re-link using your Release 6.1 object files or object libraries, you must create a Linker Description File (.LDF) for each object or object library in order to use the new linker. If you omit an LDF from your project, the IDE uses a default LDF for your target architecture.

The following sections contain information about Release 6.1 (legacy) and VisualDSP++ 2.0 (new) switches, directives, and rules of syntax, including usage examples. Where a new switch, directive, or rule of syntax has replaced the old one, the text gives a note on the preferred usage and explains the change.

New features, legacy features, and support issues of the assembler are covered as follows:

- [“Command-Line Switches” on page 3-3](#)
- [“Legacy Directives” on page 3-6](#)
- [“Conventions of Syntax” on page 3-30](#)
- [“Debugging Capabilities” on page 3-34](#)



Note that Release 6.1 directives and conventions of syntax may not be supported in future releases of DSP development software.

Command-Line Switches

Switches are options that are selected on the operating system's command line or in the `Assemble` tab of the VisualDSP++'s `Project Options` dialog box. Using these switches, you control the assembler's and preprocessor's features, including search and source-level debugging.

This section provides information on the following topics:

- [“New Assembler Switches” on page 3-3](#)
- [“Obsolete and Modified Switches” on page 3-5](#)

New Assembler Switches

The list of the new switches and a brief description of each appear in [Table 3-1](#).

Table 3-1. New Command-Line Options

Switch	Usage Description
<code>-Ao filename</code>	Writes RESOVLE() LDF commands for absolute placement to the specified .ldf file.
<code>-flags-pp -switch</code>	Passes the specified switch(s) to the preprocessor.
<code>-g</code>	Generates line number and symbol information in ELF/DWARF-2 format.
<code>-i directory</code>	Appends named <i>directory</i> to the search path for included files (.h) and data files (.dat).
<code>-ip</code>	Marks data objects for individual placement to reduce memory fragmentation at link time. Overrides consecutive placement.

Command-Line Switches

Table 3-1. New Command-Line Options (Cont'd)

Switch	Usage Description
-JumpCallShort2Long or -jcs2l	Expands out of range short CALL/JUMP instructions to long LCALL/LJUMP.
-legacy	Calls additional preprocessing; processes source programs written in Release 6.1 assembly language.
-li <i>filename</i>	Outputs named listing, embeds included files.
-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	Disables the individual placement option (-ip).
-pp	Runs the preprocessor and stops without assembling the source into an object file.
-r	Removes preprocessor output from the listing file.
-sp	Assembles the source into an object file without running the preprocessor.
-v[erbose]	Outputs version info for the assembler program and each assembly phase.
-version	Displays version information of the assembler and preprocessor programs.
-w	Omits warning messages generated during assembly.

For more information about the assembler switches, see [“Assembler Command-Line Interface”](#) on page 2-16.

Obsolete and Modified Switches

The new assembler does not support some switches of the previous release. This section lists the switches that have been removed, modified, or function differently than in Release 6.1.

Table 3-2. Obsolete and Modified Options

Switch Name	Operation under Release 6.1	Change for VisualDSP++ 2.0
-i	Shows the contents of .INCLUDE files in the listing file.	Removed
-l	Generates a listing file.	Requires the <i>filename</i> argument: -l <i>filename</i>
-m	Expands macros in the listing file.	Removed
-s	Disables semantics checking.	Removed
-ui	Specifies one or more directories to search for included files.	Replaced with -i <i>pathname</i>

Legacy Directives

Directives are instructions that you include in source programs in order to control the assembly process. VisualDSP++ 2.0 assembler directives are listed in [Table 2-8 on page 2-45](#).

For compatibility with the Release 6.1 of the assembler software, the current release supports a set of legacy directives. [Table 3-3](#) lists these directives and corresponding replacements. A description of each directive appears in the following sections. You can use the provided replacements, preprocessor commands and assembler directives, to update your legacy source code to comply with the ADSP-219x assembler syntax.

Assembly language programs developed under Release 6.1 DSP development software, may be processed by `easm219x.exe`. To do this, use the `-legacy` command-line switch.

Table 3-3. Release 6.1 Legacy Directives

Release 6.1 Directive	Change for VisualDSP++ 2.0
<code>.CONST</code> (see page 3-8)	<code>#define</code> (see page 4-19)
<code>.DMSEG</code> (see page 3-9)	<code>.SECTION/DM</code> or <code>.SECTION/DATA</code> (page 2-59)
<code>.ENTRY</code> (see page 3-11)	<code>.GLOBAL</code> (see page 2-50)
<code>.EXTERNAL</code> (see page 3-12)	<code>.EXTERN</code> (see page 2-49)
<code>.GLOBAL</code> (see page 2-51)	Modified. You can make any symbol globally available with <code>.GLOBAL</code> .
<code>.INCLUDE</code> (see page 3-13)	<code>#include</code> (see page 4-28)
<code>.INDENT</code> (see page 3-15)	Removed
<code>.INIT</code> (see page 3-16)	<code>.VAR</code> (see page 2-62)

Assembler Enhancements and Legacy Support

Table 3-3. Release 6.1 Legacy Directives (Cont'd)

Release 6.1 Directive	Change for VisualDSP++ 2.0
.INIT24 (see page 3-16)	.VAR/INIT24 (see page 2-66)
.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 2-59)
.PAGE	Removed. The ADSP-218x processors do not support paging.
.PMSEG (see page 3-9)	.SECTION/PM or .SECTION/CODE (see page 2-59)
.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.
.VAR/ABS (see page 3-26)	RESOLVE() (see page 2-22)
.SEGMENT	.SECTION (see page 2-59)
.SETDATA	.VAR (see page 2-62)
.VAR/CIRC (see page 3-27)	Base registers B0-B7



You may need to revise your legacy source programs when assembling with the new assembler. For more information about legacy code support, see the following sections or refer to the publications listed in [“Related Documents” on page 1-6](#).

.CONST, Declare a Constant

The `.CONST` directive defines assembler constants. Once you declare a symbolic constant, you may use it in place of the actual number.

The `.CONST` directive has the following syntax:

```
.CONST symbol = replacementText;
```

Only an arithmetic or logical operation on two or more integer constants may be given as an expression; symbols are not allowed. For more information on the assembler expressions and operators, see [“Assembler Expressions” on page 2-36](#).

A single `.CONST` directive may contain one or more constant declarations, separated by commas, on a single line. A list of multiple declarations may not be continued on the following line.

To comply with VisualDSP++ assembler syntax, use C-style `#define` preprocessor command to declare a symbolic constant. To declare a hexadecimal constant, use `"0x"` numeric format for hex numbers. For more information about the `#define` command, see [“#define” on page 4-19](#).

Example:

```
.CONST TAPS=15, BASE=H#0D49, Sqrt2=H#5A82;

// This line of legacy code corresponds to the following lines:
#define TAPS 15
#define BASE 0x0D49
#define Sqrt2 0x5A82
```



Note that a single `#define` preprocessor command contains only one constant declaration.

.DMSEG and .PMSEG, Place Data and Code in Memory Sections

The `.PMSEG` and `.DMSEG` directives are similar to the `/type` qualifier of the `.SECTION` directive.

These directives have the following syntax:

```
.PMSEG pmsection_name;  
.DMSEG dmsection_name;
```

The `.PMSEG` directive causes the linker to place all of the module's code and data structures in the program memory section `pmsection_name`. The `.DMSEG` directive causes the linker to place all of the module's data structures in the data memory section `dmsection_name`. The `pmsection_name` and `dmsection_name` sections must be previously defined in the Linker Description File. The `.PMSEG` and `.DMSEG` directives must precede the `.MODULE` directive in your source file.

Here is an example that locates only the DM data of a module in a segment named `Audio_Samples`:

```
.DMSEG Audio_Samples;  
.MODULE Sample_Input;  
  
.VAR/DM/CIRC sample_buffer[15];  
.VAR/DM other_buffer[5];  
.VAR/DM another_buffer[5];  
.VAR/DM variable1;  
  
{...instructions for SAMPLE_INPUT routine}  
  
.ENDMOD;  
/* The code for the SAMPLE_INPUT routine is in program memory;  
   it assembles and links normally.*/
```

To define placement of code and data objects in program memory and data memory sections using the new assembler syntax, use the `.SECTION/PM` and `.SECTION/DM` directives and the Linker Description File.

Legacy Directives

Sections define groupings of instructions and data that are set as contiguous memory addresses in the DSP. Each `.SECTION` symbol corresponds to an input section name in the Linker Description File (`.LDF`). The `.DMSEG` example may be revised using the `.SECTION/type` directive:

```
.SECTION/DM Audio_Samples;  
.VAR sample_buffer[15];  
  
.SECTION/PM Sample_Input;  
/*...instructions for SAMPLE_INPUT routine */  
  
.SECTION/DATA Audio_Samples;  
.VAR other_buffer[5];  
.VAR another_buffer[5];  
.VAR variable1;
```

For more information on the `.SECTION` directive, see “[.SECTION, Declare a Memory Section](#)” on page 2-59.



Note that only one program `.MODULE` is allowed per source file, whereas multiple program memory sections (`.SECTION/PM`) define mapping of code and possibly data in a single assembly file.

.ENTRY, Make a Program Label Globally Available

The `.ENTRY` directive allows program labels to be referenced in other modules. By default, a label is only valid in the module it is declared. Once you have changed the label's scope to global using the `.ENTRY` directive, it is available for export. This lets you use the label for subroutine calls or inter-module jumps.

The `.ENTRY` directive uses the syntax:

```
.ENTRY program_label[, ...];
```

A single `.ENTRY` directive may declare one or more global labels, separated by commas, on a single line. A list of multiple declarations may not be continued on the following line.

Once the label is declared as global, other modules or linked files can import (reference) it with `.EXTERNAL` (Release 6.1) or `.EXTERN` (current release).

To comply with the `easm219x` syntax, use the `.GLOBAL` directive to make symbols, including program labels, globally available. For more information about the `.GLOBAL` directive, see [“.GLOBAL, Make a Symbol Globally Available” on page 2-51](#). For more information about the `.EXTERN` directive, see [“.EXTERN, Refer to a Globally Available Symbol” on page 2-49](#).

Example:

```
.ENTRY adpcm_encode, adpcm_decode;
// make labels visible outside current module using the
// Release 6.1 syntax

.GLOBAL adpcm_encode, adpcm_decode;
// make labels visible outside current file using the
// VisualDSP++ 2.0 syntax
```

.EXTERNAL, Refer to a Globally Available Symbol

The `.EXTERNAL` directive allows a code module to reference global data structures (variables, buffers, and ports) and entry labels declared in other modules. The symbol in question must be defined as a `GLOBAL` or `ENTRY` symbol in the module in which it originates and must be defined as an `.EXTERNAL` before it can be referenced in another module.

This directive has the form:

```
.EXTERNAL symbol[, ...];
```

The current version of the assembler software has replaced the `EXTERNAL` keyword with `EXTERN`. To comply with the `easm219x` syntax, before importing the symbol, ensure that it is declared `.GLOBAL` in the file to be linked with the current one.

Example:

```
.EXTERNAL fir_start;
// references the global label using the legacy syntax.

.EXTERN fir_start;
// references the global label using the new syntax.
```

For more information on the `.EXTERN` directive, see [“.EXTERN, Refer to a Globally Available Symbol” on page 2-49](#). For more information on the `.GLOBAL` directive, see [“.GLOBAL, Make a Symbol Globally Available” on page 2-51](#).

.INCLUDE, Include Other Source File

The `.INCLUDE` directive is used to include another source file in the file being assembled. The assembler opens, reads, and assembles the indicated file when it encounters the `.INCLUDE` statement. The assembled code is incorporated into the output `.DOJ` file. When the assembler reaches the end of the included file, it returns to the original source file and continues processing.

The `.INCLUDE` directive has the form:

```
.INCLUDE <filename>;
```

If the file to be included is in the current project directory, only the filename need be given inside the brackets. If the file is in a different project directory, you must give the path of this directory with the filename (or with the `ADII` environment variable). For example, if the file to be included is named `newcode.dsp` and is located in a subdirectory `C:\219x\filters\`, then the `.INCLUDE` directive must be given in this way:

```
.INCLUDE <C:\219x\filters\newcode.dsp>;
```

This allows the assembler to find the file.

Included files may in turn have `.INCLUDE` statements within them; nesting of included files is limited only by memory. Included files may not, however, contain C preprocessor directives, such as `#define`. To include a file that contains C preprocessor directives, use the `#include` command instead of `.INCLUDE`.

The `.INCLUDE` directive allows modular programming. For example, in many cases it is useful to develop a library of subroutines or macros to be shared between different programs. Rather than rewriting the routines for each program, you can incorporate the macro library into an assembled module using the `.INCLUDE` directive.

Legacy Directives

The current release of the assembler software has replaced the `.INCLUDE` directive with `#include`. To enclose the *filename* argument, use double quotation marks (“”) in place of angle brackets (<>):

```
#include "C:\219x\filters\newcode.dsp";
```

Although the source programs that use `.INCLUDE` may be re-assembled with the `-legacy` switch, we recommend that you revise legacy source programs in order to avoid incompatibility with future releases of the DSP development tools. For more information about the `#include` preprocessor command, refer to [“#include” on page 4-28](#).

.INDENT, Indent a Listing File

The `.INDENT` directive indents the text in the listing file (`.LST`) that the assembler generates when you use the `-l` switch.

The `.INDENT` directive has the following form:

```
.INDENT expression;
```

Example:

```
.INDENT 9;    // 9 spaces are left at the left margin
...
...           // instructions
...
.INDENT 5;    // 5 spaces are left at the left margin
```

The *expression* marks the left bound of a row; each text line begins at column *expression* + 1. By placing the `.INDENT` directive at the beginning of your assembly source file, you apply the formatting to the entire listing file. The current setting is valid until the assembler encounters the following `.INDENT` statement.



The `.INDENT` directive is omitted in VisualDSP++ 2.0. To format a listing output, use `.PAGEWIDTH` ([page 2-55](#)), `.PAGELENGTH` ([page 2-54](#)), and `.LEFTMARGIN` ([page 2-52](#)).

Legacy Directives

.INIT, Initialize a Variable or Buffer

The `.INIT` directive initializes variables and buffers. Initialization values may be listed in the directive statement or supplied by an external file.

The `.INIT` directive takes one of the following forms:

```
.INIT buffer_symbol: constant, constant, ...;  
.INIT buffer_symbol: ^other_buffer or %other_buffer, ...;  
.INIT buffer_symbol: <filename.dat>;
```

The `^` and `%` operators can be used to initialize the buffer or variable with the base address or length of other buffer(s). Any combination of constants, buffer address pointers, and buffer length values may be given, separated by commas.

Here are some examples:

```
.INIT seed_values: 1,2,3,5,7;  
// This initializes the buffer seed_values with the listed  
// constants.  
  
.INIT buffer_ptr: ^input_buf;  
// Here the variable buffer_ptr is initialized with the buffer  
// input_buf by referencing its start address.  
  
.INIT cos: <cosines.dat>;  
// The assembler establishes a pointer to cosines.dat and the  
// linker initializes the buffer cos with the data file  
// contents.  
  
.INIT inputs: 'ABCD';  
// This initializes the first four locations of the data buffer  
// inputs with the ASCII codes for the letters A, B, C, and D.
```

If the initialization file is in the current project directory of your operating system, only the filename need be given inside the brackets. Otherwise, you must give the path of this directory with the filename. For example, if `inits.dat` is the initialization file for a buffer named `samples`, and is

located in the DOS subdirectory `C:\219x\filter3\`, then the `.INIT` directive should be given in this way:

```
.INIT samples: <C:\219x\filter3\inits.dat>;
```

A special syntax of the `.INIT` directive, `.INIT24`, lets you store 24 bits of data in a program memory section, rather than the default 16 bits. This allows you to access the lower 8 bits of each 24-bit program memory word when initializing data buffers or variables in source code.

Example:

```
//statement computes a 16-bit address:
.INIT var: ^label + 10;

//statement computes a 24-bit address:
.INIT24 var: ^label + 10;
```

If you are upgrading your code so it adheres to the current version of the assembly language, you must use the `.VAR` directive to declare and initialize variables and buffers. See [“.VAR, Declare a Data Variable or Buffer” on page 2-62](#) for information about the `.VAR` directive and its special case, `.VAR/24`.

The following is the example of the revised code:

```
/* Legacy syntax declaration: */

.VAR/DM/SEG=seg_mydata sqrt_coeff[3], buf_input[10];
.INIT sqrt_coeff: H#5D1D, H#A9ED, H#46D6;
.INIT buf_input: <bufinits.dat>;

/* Current syntax declaration: */

.SECTION/DATA seg_mydata;
.VAR sqrt_coeff[3] = { 0x5D1D, 0xA9ED, 0x46D6 };
.VAR buf_input[10] = “bufinits.dat”;
```



Note that a single `.VAR` directive can contain one or more external data initialization file(s) (`.DAT`).

.INIT and ASCII String Initialization Support

The assembler of VisualDSP++ 2.0 supports 8-bit ASCII string initialization. This allows the full use of the 8-bit ASCII character set (256 characters), including digits, and special characters.

String initialization takes one of the following forms:

```
.INIT symbolString: 'initString', 0;
```

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

Example:

```
.VAR/RAM/DM bandwidth[21];  
.INIT bandwidth[21]: 'Rec stat : Play stat', 0;
```

The assembler also accepts 8-bit ASCII characters within comments.

.LOCAL, Create a Unique Version of the Label

The `.LOCAL` directive is given with program labels used in macros. The `.LOCAL` directive instructs the assembler to create a unique version of the label at each macro invocation. This prevents duplicate label errors occurring when a macro is called more than once in a code module.

The `.LOCAL` directive has the form:

```
.LOCAL label_symbol[, ...];
```

The assembler creates unique versions of `label_symbol` by appending a number to it; this can be seen in the `.LST` file if macros are expanded.

To comply with the current version of the assembly language, use a trailing “?” to ensure that unique label names are generated no matter how many times the same macro is invoked. The preprocessor takes the label symbol and postpends “`_num`” to it, where `num` is uniquely generated for every macro expansion, for example:

```
abcd?      ==>   abcd_1
```

Macros that need to have “?” preserved should guarantee that the question mark character does not immediately follow an identifier, for example:

“`abcd?`” is a generated label, while “`abcd ?`” is not.

Example:

```
#define loop(x, y)  mylabel?: x = 1+1; \  
x = 2+2; \  
ourlabel?: y = 3*3; \  
y = 5*5; \  
jump mylabel?; \  
jump yourlabel?;  
loop (bz, kjb)  
loop (lt, ss)  
loop (yc, jl)
```

Legacy Directives

Generates the following output:

```
mylabel_1: bz = 1+1; bz = 2+2; yourlabel_1: kjb = 3*3; kjb =  
5*5; jump mylabel_1; jump yourlabel_1;  
  
mylabel_2: lt = 1+1; lt = 2+2; yourlabel_2: ss = 3*3; ss = 5*5;  
jump mylabel_2;  
  
jump yourlabel_2;  
  
mylabel_3: yc = 1+1; yc = 2+2; yourlabel_3: jl = 3*3; jl = 5*5;  
jump mylabel_3;  
  
jump yourlabel_3;
```

.MACRO and ENDMACRO, Define a Macro

Macros are created with the assembler's `.MACRO` directive. Each statement within the macro can be an instruction, directive, or macro invocation. The `.ENDMACRO` directive marks the end of a macro definition.

A macro definition has the following syntax:

```
.MACRO macro_symbol[(%1,%2,...,%n)];  
...  
...  
.ENDMACRO;
```

In the macro's code, the arguments are marked by the placeholders `%1`, `%2`, `%3`, etc. When the macro is invoked, the placeholders are replaced by argument values passed in the call. The correct number of arguments must be passed.

A macro is invoked with its name. The invocation may not contain additional program statements, such as instructions, preprocessor directives, or other macro invocations, on the same line of source code.

When the macro is called, the arguments passed may be anything from the following list:

- constant or expression
- symbol (may be any reserved keyword except `MACRO`, `ENDMACRO`, `CONST`, or `INCLUDE`)
- expressions with special address pointer (^) and length of (%) operators

VisualDSP++ 2.0 assembler has eliminated the `.MACRO` directive, although the source programs that use `.MACRO` may be re-assembled with the `-legacy` switch. We recommend that you revise your legacy macro declarations to avoid incompatibility with future releases of the DSP development software. Another way to define macros is using the `#define` C-style

Legacy Directives

preprocessor command. For more information about this command, see [“#define” on page 4-19](#).

Revise `.MACRO` declarations as shown below:

```
// MACRO declaration using the Release 6.1 assembler syntax:

.MACRO getsfirst(%1);
start:
    M5=1; I6=1; MODIFY(I6,M4); %1=DM(I6,M5);
.ENDMACRO;

//MACRO declaration using the VisualDSP++ 2.0 assembler syntax:

#define getsfirst(a)\
start:\
    M5=1; I6=1; MODIFY(I6,M4); a=DM(I6,M5)

// Macro invocation, common:

getsfirst (MR1);
```

.MODULE and .ENDMOD, Declare a Program Module

The `.MODULE` directive defines the program module's name and marks it beginning, whereas the `.ENDMOD` directive marks the module's end. The assembler stops when it reaches the “end” directive. A source code file may contain only one program module.

This directive has the form:

```
.MODULE/qualifier/qualifier moduleSymbol;
```

Where */qualifiers* are keywords that define memory type and placement of a module. This section lists valid *qualifiers* and provides a brief description of each.

Qualifier	Description
PM or CODE	Memory type — Program Memory.
DM or DATA	Memory type — Data Memory.
RAM	Memory type — Random Access Memory.
ROM	Memory type — Read Only Memory.
ABS= <i>address</i>	Placement of a module at absolute start <i>address</i> .
SEG= <i>secName</i>	Placement of a module in the LDF-declared section <i>secName</i> .

Legacy Directives

ABS=address

The **ABS** qualifier places the module's code at a particular *address* in program memory, making it non-relocatable. This means that the linker is forced to reserve memory for the module at the specified *address*. Modules that do not have the **ABS** qualifier are relocatable.

SEG=secName

The **SEG** qualifier locates the module in a specific memory section, *secName*, which is declared in the Linker Description File. If you use both the **ABS** and **SEG** qualifiers and specify an absolute address that is not in the named section, the linker outputs an error message.

Example 1:

```
.MODULE/PM/ABS=0x0040 main_prog;
...
.ENDMOD;
/* This example declares main_prog which is to be located in
   program memory at address 40 (hexadecimal). */
```

Example 2:

```
.MODULE/SEG=fir filter_routine;
...
.ENDMOD;
/* This example declares the relocatable module
   filter_routine to be placed in a memory segment named fir.
   fir is defined in the Linker Description File. */
```

Although the **.MODULE/.ENDMOD** pair has been replaced in the current release, **easm219x** assembler processes your legacy programs when the **-legacy** switch is used.

To mark a program memory section using VisualDSP++ 2.0 assembler syntax, use the **.SECTION/PM (.SECTION/CODE)** directive. 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.

For more information about the `.SECTION` directive, including syntax and usage examples, see [“.SECTION, Declare a Memory Section” on page 2-59](#). For information about the `.LDF` file and the `elfdump` utility, see the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs*.



Note that only one `.MODULE` is allowed per source file, whereas multiple program memory sections define mapping of code and data in a single assembly file.

Legacy Directives

.VAR/ABS, Place a Variable at the Specified Address

The `/ABS` qualifier of the `.VAR` directive specifies a particular *address* in program memory or data memory at which the variable or buffer is to be placed by the linker, making the variable (buffer) non-relocatable. The linker is forced to reserve memory for the variable or buffer at the specified *address*. Variables and buffers that do not have the `/ABS` qualifier are relocatable.

To comply with the VisualDSP++ 2.0 assembler syntax, use the absolute placement qualifier on the data or code section to which the variable (buffer) in question belongs.

You can also use the `-Ao` (absolute placement) switch to direct the assembler to write resolve commands to the specified Linker Description File (`.ldf`).

For example, the following circular buffer is defined at an absolute address in the data module in the `testABS.dsp` file:

```
.MODULE test;  
.VAR/DM/RAM/ABS=0x2000/CIRC x[4];  
.ENDMOD;
```

When `testABS.dsp` assembles with:

```
easm219x -legacy -c testABS.dsp
```

the assembler automatically creates the `resolve_testABS.ldf` file with the `RESOLVE()` command in the project's LDF file:

```
RESOLVE( x, 0x2000 )  
  // Filename: testABS.dsp.ldf  
  // These are assembler generated LDF RESOLVE commands  
  // for -legacy .VAR directives with /ABS qualifiers.  
  // .VAR x in testABS.dsp, line 3, index 4
```

Note that the `RESOLVE()` commands must be used for global data.

[For more information, see “-Ao filename” on page 2-22.](#)

.VAR and Circular Buffer Setup

The assembler of Release 5x/6x 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];
```

Legacy Directives

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.

The ADSP-219x processor eliminates the existing hardware restriction of the ADSP-218x DSP architecture on a circular buffer starting address. The ADSP-219x enables you to declare any number of circular buffers by designating B0-B7 as the base registers for addressing circular buffers. These base registers are mapped to the “register” space on the core.

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
  REG(B5)=AR;
  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 about circular buffers, refer to:

- *ADSP-219x/2191 DSP Hardware Reference*
- *ADSP-219x/2192 DSP Hardware Reference*

Conventions of Syntax

The assembler supports several numeric bases and comment formats. You build assembly expressions and statements using predefined set of operators and following the assembler's syntax.

This section provides information on the following topics:

- [“Modified Assembler and Preprocessor Operators” on page 3-31](#)
- [“Modified Numeric Conventions” on page 3-32](#)
- [“New and Legacy Comment Conventions” on page 3-33](#)

Modified Assembler and Preprocessor Operators

Syntax for the special “length of”, “address of”, “page of”, and the preprocessor’s division operators have changed:

Table 3-4. Modified Operators

Release 6.1 Operator	Usage Description	Change for VisualDSP++ 2.0
\	Division	/
% <i>symbol</i>	Length of <i>symbol</i> in words.	LENGTH(<i>symbol</i>)
^ <i>symbol</i>	Address pointer to <i>symbol</i> .	<i>symbol</i>
PAGE <i>symbol</i> PAGEOF <i>symbol</i>	Upper address bits of a page associated with <i>symbol</i> .	Removed. Release 5.x and 6.x declarations refer to boot pages. 219x PAGE declarations refer to external (16-bit) memory. Since the address space is 24-bit, which cannot be address directly in a 24-bit opcode, the use of page registers is required.

Modified Numeric Conventions

Hexadecimal numeric format has been modified: the usage of the “0x” prefix is preferred for hexadecimal numbers. When assembling programs coded with any of Release 6.1 numeric conventions listed in [Table 3-5](#) using the ADSP-219x DSP assembler, select the `-legacy` switch.

Table 3-5. Hexadecimal Numeric Formats

Release 6.1 Convention	Usage Description	Change for VisualDSP++ 2.0
00 <i>number</i> H# <i>number</i> h# <i>number</i>	Hexadecimal base <i>number</i> formats.	0x <i>number</i>

New and Legacy Comment Conventions

The assembler introduces support for C++-style comment formats within assembly language source files:

The assembler introduces support for C++-style comment formats within assembly language source files:

`//comment` — `"//"` denote each single-line comment.

`/* comment */` — `"/* */"` denote each multi-line comment.

The following comment formats are legacy formats:

`!comment` — `"!"` begins each single-line comment.

`{ comment }` — `"{ }"` enclose each multiple-line comment.

When assembling programs coded with any of Release 6.1(legacy) comment conventions using the ADSP-219x DSP assembler, you must select the `-legacy` switch.

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 [page A-1](#).

Debugging Capabilities

The assembler and preprocessor include features that allow for debugging of your source code programs. These features include the ability to generate binary data in ELF/DWARF-2 file format and to output line number and the file name of a source file.

When assembling your source file, either from a command line or within VisualDSP++, you have the option of generating binary information for source level debugging. You can do it by selecting the `-g` command-line switch or checking the `Generate debug information` check box in the `Assemble` tab of the VisualDSP++'s `Project Options` dialog. The ELF assemblers generate debug information in DWARF-2 format.

For more information on the `-g` switch, see [“-g” on page 2-24](#). For more information on the VisualDSP++ IDDE, see the *VisualDSP++ 2.0 User's Guide for ADSP-21xx DSPs*.

You also have the option of outputting the line number and file name of your original source file from the assembler by using the `#line` preprocessor command. For more information on the `#line` command, see [“#line” on page 4-29](#).