A FILE FORMATS

Overview

The development tools support many file formats, in some cases several for each development tool. This appendix describes the formats of files that you prepare as input for the tools and points out features of files, which are produced by the tools.

The three types of file formats that you can learn about in this appendix are as follows:

- “Source Files” on page A-2
- “Build (Processed) Files” on page A-7
- “Debugger Files” on page A-15

Most of the development tools use industry standard file formats. Sources that describe these formats appear in “Format References” on page A-17.

Source Files

Source files are files that you prepare as input for the development tools. This section describes the following types of input file formats:

C/C++ Source Files

These are text files containing C/C++ code, compiler directives, possibly a mixture of assembler code and directives, and (typically) preprocessor commands.

Several "dialects" of C code are supported: pure (portable) ANSI C, and at least two subtypes* of ANSI C with ADI extensions. These extensions include memory type designations for certain data objects, and segment directives, used by the linker to structure and place executable files.

For information on using the C compiler and associated tools, as well as a definition of ADI extensions to ANSI C, see the *C/C++ Compiler & Library Manual for ADSP-21xxx DSPs*.

For information on specifying the C dialect and general C code handling within VisualDSP++, see the *VisualDSP++ User's Guide for ADSP-21xxx DSPs*.

---

* With and without builtin function support; a minimal differentiator. There are others.

Assembly Source Files (.ASM)

Assembly source files are text files containing assembly instructions, assembler directives, and (optionally) preprocessor commands. For information on assembly instructions, see the *Instruction Set Reference* manual for the corresponding DSP.

The DSP's instruction set is supplemented with assembler directives. Preprocessor commands control macro processing and conditional assembly or compilation. For information on the assembler and preprocessor, see the *VisualDSP++ Assembler and Preprocessor Manual for ADSP-21xxx DSPs*.

Assembly Initialization Data Files (.DAT)

These are text files containing fixed-point or floating-point data. These files can provide the initialization data for an assembler `.VAR` directive or serve in other tool operations. When a `.VAR` directive uses a `.DAT` file for initialization data, the assembler reads the data file and initializes the buffer in the output object file (`.DOJ`). Data files have one data value per line and may have any number of lines.

Floating-point values in data files consist of a decimal mantissa value with a decimal point and a decimal integer exponent value. The exponent is separated from the mantissa with an upper or lower case "e." The mantissa and exponent can be signed. The following are all valid floating-point values:

```
1.2345e12
567.8e-3
-7.1234
```

The assembler converts these decimal representations to floating-point format, either standard IEEE 32-bit single precision or 40-bit extended precision. The source code file in which a `.VAR` initialization occurs specifies the format (32-bit or 40-bit) for floating-point numbers using the

.PRECISION directives. If a floating-point number in an initialization file does not fit in the specified format, the assembler rounds the number. The rounding mode is also specified in the source code file with a .ROUND directive. For information, see the *VisualDSP++ Assembler and Preprocessor Manual for ADSP-21xxx DSPs*.

Fixed-point values (integers) in data files may be signed, and they may be decimal-, hexadecimal-, octal-, or binary-base values. The assembler uses the prefix-conventions in Table A-1 for identifying a fixed-point value's numeric base.

Table A-1. Fixed-Point Values in Data Files

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

For all numeric bases, the assembler uses 32-, 40-, or 48-bit words for data storage; 48-bit data is for PM only. The largest word in the buffer determines the size for all words in the buffer. If you have some 32-bit data in a 40-bit wide buffer, the assembler loads the 32-bit value into the most significant 32 bits of the 40-bit memory location and zero-fills the lower eight bits. Similarly, the assembler loads 32- or 40-bit data into the most significant bits in a 48-bit wide buffer.

Table A-2 shows some data storage examples.

Table A-2. Data Storage Examples

Decimal data in .DAT	Hexadecimal data in .DAT or 32-bit DM	Hexadecimal data in .DAT or 40-bit DM	Hexadecimal data in .DAT or 48-bit PM
7	0x00000007	0x0000000700	0x000000070000
255	0x000000FF	0x000000FF00	0x000000FF0000
32768	0x00008000	0x0000800000	0x000080000000
10855845	0x00A5A5A5	0x00A5A5A500	0x00A5A5A50000
305419896	0x12345678	0x1234567800	0x123456780000
N/E[1]	N/E	0x1122334455	0x112233445500
N/E	N/E	0x0012345678	0x001234567800
N/E	N/E	N/E	0x000012345678

[1] N/E = not expressible

Header Files (.H)

Header files are text files that contain macros or other preprocessor commands that the preprocessor substitutes into source files. For information on macros or other preprocessor commands, see the *VisualDSP++ Assembler and Preprocessor Manual for ADSP-21xxx DSPs*.

Linker Description Files (.LDF)

The linker's `.LDF` files are ASCII text files that contain commands for the linker in the linker's scripting language. For information on this scripting language see "Linker Description File Reference" on page 2-38.

Linker Command-Line Files (.TXT)

The linker's command-line files are ASCII text files that contain command line input for the linker. For more information on the linker's command line, see "Linker Command-Line Reference" in Chapter 2 Linker.

Build (Processed) Files

Build files are files that the development tools produce when building your VisualDSP++ project. This section describes the following types of build file formats:

Assembler Object Files (.DOJ)

The assembler's output object files are in binary, Executable and Linkable File (ELF) format. Object files contain relocatable code and debugging information for your program's segments. The linker processes your object files into an executable file. For information on the ELF and COFF formats that may be used for object files, see "Format References" on page A-17.

Archiver Archive Files (.DLB)

The archiver's output archive files are in binary, Executable-Linkable-File (ELF) format. Archive files contain one or more object files, called Archive Elements. The linker searches through archive files for any archive elements that your code uses. For information on the ELF format that is used for executable files, refer to "Format References" on page A-17.

Note that the archiver automatically converts input objects from COFF to ELF format.

Linker Executable Files (.DXE, .SM, .OVL, .DLO)

The linker's output executable files are binary, Executable-Linkable-File (ELF) format. Executable files contain your program's code and debugging information. The linker may fully or partially resolve addresses in executable files depending on the options given on the linker's command line and in your linker description file. For information on the ELF format that is used for executable files, see the Tools Interface Standards Committee texts that are cited in "Format References" on page A-17.

Note that the linker automatically converts input objects from COFF to ELF format.

Linker Memory Map Files (.MAP)

The linker's memory map files are ASCII text files that contain memory and symbol information for your executable file(s). The map contains a summary of memory that you define with `MEMORY{}` commands in your linker description file and provides a listing of the absolute addresses of all symbols.

Loader Hex Format Files (.LDR)

The loader's output Hex-format files are in ASCII, Intel Hex-32 format. Hex files from the loader support 8-bit wide PROMs. The files are used with an industry-standard PROM programmer to program memory devices for your hardware system. One file contains data for the whole series of memory chips to be programmed.

The following example shows how the Intel Hex-32 format appears in the loader's output file. Each line in the Intel Hex-32 file contains either a data record, an extended linear address record or the end of file record:

```
:020000040010E9                    extended linear address record
:0A0004003C40343434261422260850    data record
:00000401FB                        end of file record
```

The extended linear address record is used because a data record has only a 4-character (16-bit) address field, but the ADSP-210xx processors require 32 bits to address data memory and 24 bits to address program memory. The extended linear address record specifies address bits 16-31 for the data records that follow it. Data records are organized into the following fields:

```
:0A0004003C40343434261422260850      example record

:                        start character
 0A                      byte count of this record
   0004                  address of first data byte
       00                record type
3C                       first data byte
08                       last data byte
50                       checksum
```

Extended linear address records have the following fields:

```
:02000000340850
:                       start character
 02                     byte count (always 02)
   0000                 address (always 0000)
       00               record type
         3408           offset address
             50         checksum
```

The end of file record looks like this:

```
:00000001FF
:                       start character
 00                     byte count (zero for this record)
   0000                 address of first byte
       01               record type
         FF             checksum
```

Loader ASCII Format Files (.LDR)

The loader's ASCII-format file is similar to an assembler initialization data file (`.DAT`). The data order is one 16-bit hexadecimal value per line, providing lower-, middle-, then upper-16-bits of each 48-bit instruction. Use files of this format in the same manner as data files. For information on this format, see "Assembly Initialization Data Files (.DAT)" on page A-3.

Loader Include Format Files (.LDR)

The loader's include-format file is an ASCII text file that consists of 48-bit instructions one per line with each instruction presented as three 16-bit hexadecimal numbers. The data order is lower-, middle-, then

upper-16-bits of each 48-bit instruction. A few example lines from an Include format file appear as follows:

```
0x005c, 0x0002, 0x0620,
0x0045, 0x0000, 0x1103,
0x00c2, 0x0002, 0x06be
```

This file format lets you include the loader file in a C program. To include this file in a C program, use the following code:

```
const unsigned loader_file[] =
{
#include "foo.ldr"
};
const unsigned loader_file_count = sizeof loader_file / sizeof
loader_file[0];
```

`loader_file_count` reflects the actual number of elements in the array, and can be used for processing the data.

Loader Binary Format Files (.LDR)

The loader's binary-format file supports a variety of PROM and micro-controller storage options and uses less space than the other loader file formats. The binary file contains 48-bit instructions in big-endian format (most significant bit first).

Splitter Motorola S-Record Format Files (.S_#)

Motorola S Record format is similar to the Intel standard. The PROM splitter supports three file formats which differ only in the width of the address field: `S1` (16 bits), `S2` (24 bits) or `S3` (32 bits).

Each `S` Record file starts with a header record and ends with a termination record. In between are data records, one per line. Here are some examples:

```
S00600004844521B                    header record
S10D00043C4034343426142226084C      data record (S1)
S903000DEF                          termination record (S1)
```

A header record has the following fields:

```
S00600004844521B
S0                                  start character (always S0)
  06                                byte count of this record
    0000                            address of the first data byte
        484452                      identifies records that follow
              1B                    checksum
```

The S1 data record has the following format:

```
S10D00043C4034343426142226084C
S1                                  start character
  0D                                byte count of this record
    0004                            address of first data byte
        3C                          first data byte
           08                       last data byte
              4C                    checksum
```

The S2 data record has the same format, except that the start character is S2 and the address field is six characters wide. The S3 data record is the same as the S1 data record except that the start character is S3 and the address field is eight characters wide.

Termination records have an address field that is 16-bits, 24-bits or 32-bits wide, whichever matches the format of the preceding records. The S1 termination record has the following format:

```
S903000DEF
S9                                  start character
  03                                byte count of this record
    000D                            address
        EF                          checksum
```

The S2 termination record has the same format, except that the start character is S8 and the address field is six characters wide. The S3 termination record is the same as the S1 format except that the start character is S7 and the address field is eight characters wide.

Splitter Hex Format Files (.H_#)

The splitter's output Hex-format files are in ASCII, Intel Hex-32 format. Hex files from the loader support a variety of PROM devices. For an example of how the Hex format appears for an 8-bit wide PROM, see "Loader Hex Format Files (.LDR)" on page A-9. Note that the splitter prepares a set of PROM files with each PROM holding some portion of each instruction or data. This configuration differs from the loader.

Splitter Byte-Stacked Format Files (.STK)

The byte-stacked format output by the PROM splitter is intended not for PROMs, but for applications such as microcontroller transfer of data.

The byte-stacked format consists of a series of one-line headers, each followed by a block (one or more lines) of data. The last line in the file is a special header that signals the end of the file.

Lines consist of ASCII text representing hexadecimal digits. Each two characters therefore represent one byte; for example F3 represents a byte whose decimal value is 243.

The header record format is shown below:

```
2000800000000080000001E
20                          Width (in bits) of address and length
                            fields
  00                        Reserved
    80                      PROM splitter flags
                            (80=PM, 00=DM)
      00                    User-defined flags, loaded using -u
                            switch
        00000008            Start address of data block
                0000001E    Number of bytes that follow (until next
                            header record or termination record);
                            must be nonzero.
```

In this example, the start address and block length fields are 32 bits wide. The file contains program memory data (the MSB is the only flag currently used in the PROM splitter flags field). No user flags are set. The address of the first location in the block is `0x08`. The block contains 30 bytes, or 5 program memory code words.

A block of data records follows its header record, five bytes per line for data memory, six for program memory. For example:

Program Memory Segment (Code or Data):

```
3C4034343426
142226083C15
```

Data Memory Segment:

```
3C40343434
2614222608
```

The bytes are ordered left to right, most significant to least.

The termination record has the same format as the header record, except that the rightness field (number of records) is all zeros.

Debugger Files

Debugger files provide input to the debugger to define support for simulation or emulation of your program. The debugger supports all the executable file types produced by the linker (`.DXE`, `.SM`, `.OVL`, `.DLO`). To simulate I/O, the debugger supports the data file formats (`.DAT`) from the assembler, the loadable file formats from the loader (`.LDR`), and the PROM formats from the splitter (`.S_#`, `.H_#`, `.STK`)

The standard hexadecimal format for a SPORT data file is a single integer value per line. Hex numbers do not require the `0x` prefix to indicate hexadecimal. A value can have any number of digits but are read into the SPORT register, as follows:

- Hexadecimal number is converted to binary
- Number of binary bits read in matches the word size set for the SPORT register, which starts reading from the LSB. The SPORT register then fills with zeros values that are shorter than the word size or, conversely, truncates any bits beyond the word size on the MSB end

Example: a SPORT register is set for 20-bit words and the data file contains hex numbers. The simulator converts these hex numbers to binary, then fills/truncates to match the SPORT word size. In Table A-3, the number `A5A5` is filled and the number `123456` is truncated

Table A-3. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
A5A5A	1010 0101 1010 0101 1010	1010 0101 1010 0101 1010
FFFF1	1111 1111 1111 1111 0001	1111 1111 1111 1111 0001
A5A5	1010 0101 1010 0101	0000 1010 0101 1010 0101

Table A-3. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
5A5A5	0101 1010 0101 1010 0101	0101 1010 0101 1010 0101
11111	0001 0001 0001 0001 0001	0001 0001 0001 0001 0001
123456	0001 0010 0011 0100 0101 0110	0010 0011 0100 0101 0110

Format References

The following texts define industry standard file formats that are supported by VisualDSP++:

- Gircys, G. R. (1988) Understanding and Using COFF, O'Reilly & Associates, Newton, MA
- (1993) Executable and Linkable Format (ELF) V1.1 from the Portable Formats Specification V1.1, Tools Interface Standards Committee, available from `http://developer.intel.com/vtune/tis.htm`
- (1993) Debugging Information Format (DWARF) V1.1 from the Portable Formats Specification V1.1, UNIX International, Inc., available from `http://developer.intel.com/vtune/tis.htm`