B UTILITIES

Overview

Your Analog Devices development software comes with several file conversion utilities, which run from a command line only. Some of these utilities provide support for legacy code, and others are intended for a group of users who prefer to use the command-line version of the tools instead of using them through the VisualDSP++ environment.

This appendix describes the ELF file dumper and Mem21k memory initializer utilities.

Dumper — ELF File Dumper

The ELF file dumper (`elfdump.exe`) extracts data from ELF executable files (`.DXE`) and provides a text output file that describes the ELF file's contents. The ELF file dumper uses the following command line:

```
C:\Program Files\Analog Devices\VisualDSP++>elfdump
```

Usage: `elfdump {option} {`*`filename`*`}`

Table B-1. ELF File Dumper Command-Line Option Switches

Switch	Description
`-fh`	Print a file header.
`-arsym`	Print the archive symbol table.

Table B-1. ELF File Dumper Command-Line Option Switches (Cont'd)

Switch	Description
`-arall`	Print every archive member.
`-ph`	Print the program header table.
`-sh`	Print the section header table. The default is `-sh` if no other options are specified.
`-n` *`section`*	Print contents of the named section(s). Section name may use `?` and `*` wild card characters. Each section's name and type determine its output format unless overridden by a modifier (see the *`filename`* description).
`-i x0[-x1]`	Print contents of the sections numbered `x0` through `x1`, where `x0` and `x1` are decimal integers, and `x1` defaults to `x0` if omitted. Formatting rules as are for `-n`.
`-all`	Print everything. Same as `-fh -ph -sh -notes -n '*'`
`-ost`	Omit string table sections.
`filename`	File whose contents are to be printed. It can be a core file, executable, shared library, or relocatable object file. If the name is in the form `A(B)`, `A` is assumed to be an archive and `B` is an ELF element in the archive. `B` can use a pattern like the one accepted by `-n`. The `-n` and `-i` options can have a modifier letter after the main option character, which forces section contents to be formatted in the following ways: `a` Dump contents in hex and ASCII format, 16 bytes per line. `x` Dump contents in hex format, 32 bytes per line. `x`*`N`* Dump contents in hex format, `N` bytes per group (default is `N=4`). `t` Dump contents in hex format, `N` bytes per line, where `N` is the section's table entry size. If `N` is not in the range `1..32`, `32` is used. `i` Print contents as list of disassembled machine instructions.

Using the Archiver and Dumper For Disassembly

The file utilities are each useful in there own way, but can become much more effective when you combine their capabilities. One interesting application of these utilities is to disassemble a library member, converting it to source code. This application is good to have around when you discover your source for a particularly useful routine has "disappeared" and is only available as a library routine.

The following procedure lists the objects in a library, extracts an object, and converts the object to a listing file. Using the following archiver command line, list the objects in the library and write the output to a text file:

```
elfar -p libc.dlb > libc.txt
```

Assuming the current directory is

```
C:\Program Files\Analog Devices\VisualDSP++\TS001\lib>
```

open the text file, scroll through it, and find the object file that you need. Then, use the following archiver command line to extract the object from the library:

```
elfar -e libc.dlb fir.doj
```

To convert the object file to an assembly listing file with labels (similar to source, but with line numbers and opcodes), use the following `elfdumper` command line:

```
elfdump -ns * fir.doj > fir.asm
```

Using disassembly, you get a listing file with symbols. Assemble source with symbols can be useful if you are familiar with the code and have some documentation on what the code does. If symbols where stripped during linking, there are no symbols in the dumped file.

🛇 Using disassembly on a third party's library may violate the license for the third party's software. Check copyright and license issues with the code’s owner before using this disassembly technique.

Dumping Overlay Archive Files

Use the `elfar` and `elfdump` commands to extract and view the contents of the overlay archive file (*`.OVL`).

For example,

```
elfar -p CLONE2.OVL
```

will show that `CLONE2.OVL` archive consists of `CLONE2.ELF` that can be viewed with `elfdump`.

To view the `CLONE2.ELF` file, enter

```
elfdump -all CLONE2.OVL(CLONE2.elf)
```

To extract `CLONE2.ELF` and dump, enter

```
elfar -e CLONE2.ovl CLONE2.elf     - to create CLONE2.elf
elfdump -all CLONE2.elf            - to extract CLONE2.elf
```

or use whatever `elfdump` options you wish.

These commands are case-sensitive.

Mem21k — Memory Initializer

The memory initializer (`mem21k.exe`) operates on the executable file produced by the linker. When run by the compiler driver from the command line (or when selected with the `Mem21k` radio button in the Load options dialog box in the VisualDSP++ environment), the linker creates an executable file that becomes the input to the initializer. If the compiler's `-nomem` switch is used to disable the initializer, the initializer does not process the executable.

The initializer is invoked as follows:

```
mem21k [-h -v] -o outputfile inputfile
```

The command options have the following meanings:

- `-h` – Display usage
- `-v` – Verbose
- `-o` – Specify output file name

The initializer program transfers all RAM memory initializations to the `seg_init` PM ROM segment. This has two effects. First, all RAM is initialized to its proper value before the call to `main()`. This is true for embedded code that was programmed into ROM, and also for programs downloaded into RAM in an ADSP-21xxx system.

In addition to memory initialization, the initializer can reduce the overall size of an executable file by combining contiguous, identical initializations into a single block. A large array of identically initialized data (for example, zeros) are compressed to a single element in the executable after it is processed by the initializer.

The C run-time header reads the `seg_init` segment generated by the initializer to determine which memory locations should be initialized to what values. This process occurs during the `___lib_setup_processor` routine that is called for the run-time header.

There are three segments that the initializer does not attempt to compress, even if they are defined in the LDF as RAM: the initialization segment (`seg_init`), the code segment (`seg_pmco`), the run-time header segment (`seg_rth`). These segments contain the initialization routines and data, so they cannot be compressed.

The initializer program is normally run automatically by the compiler. If the compiler does not produce the executable directly, it is up to the user or the make file to process the executable manually after the linker. If the initializer is not used, the compressing of RAM segments is not performed at all.