Building a Gnu GCC Cross Compiler

Cmpware, Inc.

Introduction

The *Cmpware Configurable Multiprocessor Development Kit (CMP-DK)* is based around fast simulation models for multiple processors. While the Cmpware CMP-DK comes with several popular microprocessor models included, it does not supply tools for programming these processors. One popular microprocessor development tool chain is the *Gnu* tools from the *Free Software Foundation*. These tools are most often used in typical self-hosted development environments, where the processor and operating system used for development is also the processor and operating system used to execute the code being developed.

In the case of development for the Cmpware CMP-DK, it is more likely that the tools will be cross targeted. This means that the host machine used for software development uses one processor and operating system and the target system uses a different processor and operating system. This arrangement does not necessarily complicate the processor itself; it still only has to produce code for a single target. The major problem with cross targeted tools is that the possible combinations of host and target can be large and the audience for such tools relatively small. So such tools tend to be less rigorously maintained than more popular self-hosted tools.

This document briefly describes the complete process of building a cross targeted Gnu tool chain. It uses a popular combination of host and target, and the process is know to work correctly. This document does not address the problems that may occur in the build process which may require specialized knowledge of the internals of the Gnu tools. Repairing a broken build for such a system can quickly become very complicated and is well beyond the scope of this document.

Specifically, this document describes the sequence of commands used to build the *Gnu C compiler (GCC)* for a *SPARC8* target. This was done on October 25, 2005 on a Dell Precision 340 running Debian Linux version 3.1. It may be possible (and simpler) to download and install one of the pre-built 'binary' distributions for your system. On-line documentation on this process can be found at: ***http://gcc.gnu.org/install/***

Downloading the Binutils Distribution

The *binutils* for the Gnu compiler is a group of binary utility used to help manipulate the object files produced by the compiler. These utilities have to be built and installed before the compiler can be built and installed. It is beyond the scope of this document to give details on these utility programs. For more information on the *binutils*, see the main documentation web page for *Gnu Binutils* at: ***http://sourceware.org/binutils/docs-2.16/*** This is the primary reference for the *Gnu Binutils*.

The first step in building and installing the *binutils* is to download the complete distribution. This is a set of files including source code and various configuration and documentation files. The latest release of the *Gnu binutils* can be downloaded from ***http://ftp.gnu.org/gnu/binutils/*** or one of its mirror sites. From this download site, the file **`binutils-2.16.tar.bz2`** should be downloaded and saved. This is a 12 MB file and contains the complete *binutils* distribution. This file must first be uncompressed with the command:

```
$ bunzip2 binutils-2.16.tar.bz2
```

This produces a 79 MB file named **`binutils-2.16.tar`**. Next the archive should be unpacked with the command:

```
$ tar xvf binutils-2.16.tar
```

This produces a large directory tree rooted at **`binutils-2.16/`** filled with various files and subdirectories containing the distribution.

Building the Binutils

In the **`binutils-2.16/`** directory, make a new directory called **`objdir/`** and go to that directory and run the ***configure*** script. This is accomplished with the following commands:

```
$ mkdir objdir
$ cd objdir
$ ../configure --target=sparc-elf
```

This command checks various system and compiler functions and builds an appropriate **`Makefile`**. This script will print out quite a few status messages and ends with the message "**`creating Makefile`**". This indicates that the script has run successfully and produced a valid **`Makefile`** for this particular combination of host and target.

It is possible that this script may fail due to the software currently installed on the host machine. While it is beyond the scope of this document to describe the many things that could got wrong at this point, the solution to many problems is to use the most recent version of the *Gnu GCC* self-hosted compiler. In this case, since we are building the *Gnu GCC* 4.0.1 tools for a cross targeted system, it is advised that the most recent available *GCC* be used on the host. That said, a quick check reveals that the GCC used in this process was in fact version 3.3.5. While this script should work with any standard ANSI C compiler, switching to *GCC* should be the first step if problems are encountered.

At this point, a **`Makefile`** has been created and it can be run. As with the compiler, it is advised that the most recent *Gnu Make* be used to build the *binutils*. While any make program should work, problems encountered with non-Gnu implementations of make are most easily addressed by switching to the most recent version of *Gnu Make*. For reference, this example was implemented using *Gnu Make version 3.80*.

The *binutils* for the Sparc architecture can now be built with the command:

```
$ make
```

This process can take several minutes and will produce numerous status messages. When the build is complete, the newly compiled executables can be installed. This must be done using the *superuser* or *root* account, since the files will be installed in a shared directory of the file system where they can be accessed by other users. The commands to install the *binutils* are:

```
$ su
password:
# make install
# exit
exit
$
```

This install command will copy the following files into the **`/usr/local/sparc-elf/bin/`** directory :

```
$ ls -la /usr/local/sparc-elf/bin/
total 18812
drwxr-sr-x  2 root staff    4096 Oct 25 11:21 .
drwxr-sr-x  4 root staff    4096 Aug 11 11:06 ..
-rwxr-xr-x  2 root staff 2256236 Oct 25 11:21 ar
-rwxr-xr-x  2 root staff 3692422 Oct 25 11:21 as
-rwxr-xr-x  2 root staff 2976077 Oct 25 11:21 ld
-rwxr-xr-x  2 root staff 2231737 Oct 25 11:21 nm
-rwxr-xr-x  2 root staff 2844181 Oct 25 11:21 objdump
-rwxr-xr-x  2 root staff 2256231 Oct 25 11:21 ranlib
-rwxr-xr-x  2 root staff 2679790 Oct 25 11:21 strip
$
```

These executables are various utilities used by the *Sparc GCC* compiler to build and manipulate libraries and object files. In addition, manual pages and other supplementary material may have been installed by this command.

Once the *binutils* have been installed all of the source and object code used in this process may be deleted.

Downloading GCC Distribution

The process for downloading, building and installing the Gnu GCC is a very similar process to the one used to build the Gnu binutils, except of course, that a different set of distribution files are used.

The first step in building and installing the *Gnu GCC Compiler* is to download the complete distribution. This is a set of files including source code and various configuration and documentation files. The latest release of the *Gnu GCC compiler* distribution can be downloaded from **http://mirrors.rcn.net/pub/sourceware/gcc/releases/gcc-4.0.1/** or one of its mirror sites. From this download site, the file **gcc-4.0.1.tar.bz2** should be downloaded and saved. This is a 31 MB file and contains the complete *Gnu GCC* distribution. This file must first be uncompressed with the command:

```
$ bunzip2 gcc-4.0.1.tar.bz2
```

This produces a 218 MB file named **`gcc-4.0.1.tar`** Next the archive should be unpacked with the command:

```
$ tar xvf gcc-4.0.1.tar
```

This produces a large directory tree rooted at **gcc-4.0.1/** filled with various files and subdirectories containing the distribution.

It is beyond the scope of this document to give details on *Gnu GCC*. For more information on the *Gnu GCC*, see the main documentation web page for *Gnu GCC* at: ***http://gcc.gnu.org/*** This is the primary reference for the *Gnu GCC*.

Building the GCC Compiler

In the **gcc-4.0.1/** directory, make a new directory called **`objdir/`** and go to that directory and run the ***configure*** script. This is accomplished with the following commands:

```
$ mkdir objdir
$ cd objdir
$ ../configure --target=sparc-elf --with-newlib\
--without-headers --with-gnu-as --with-gnu-ld\
--enable-languages=c --disable-nls
```

Note that the **`../configure`** command and flags are all executed as a single command. The figure above shows the lines wrapped around because there is not enough room on a single line for the entire command. Also note that some of these flags may be redundant, but are specified to ensure that the correct *GCC* gets built. These flags may be more necessary when building other non-Sparc compilers. For instance, the **`--with-gnu-as`** specifies that the *Gnu Assembler* be used. Other systems may default to other system specific assemblers more typically found in self hosted environments. This ensures that the cross targeted compiler gets built correctly. Similarly, the **`--enable-languages-c`** causes only a C compiler to be produced. The default may produce other language compilers, including *FORTRAN*, *Ada* and / or *Java*.

Like the **`configure`** command for the *binutils*, this **`configure`** command checks various system and compiler functions and builds an appropriate **`Makefile`**. This script will print out quite a few status messages and ends with the message "**`creating Makefile`**". This indicates that the script has run successfully and produced a valid **`Makefile`** for this particular combination of host and target.

Again, as with the *binutils*, it is possible that this script may fail due to the software currently installed on the host machine. While it is beyond the scope of this document to describe the many things that could got wrong at this point, the solution to many problems is again to use the most recent version of the *Gnu GCC* self-hosted compiler.

At this point, a **`Makefile`** has been created and it can be run. As with the *binutils*, it is advised that the most recent *Gnu Make* be used to build the *GCC*. The *GCC* for the Sparc architecture can now be built with the command:

```
$ make
```

This process can take several minutes or even hours depending on the host machine and the configuration and will produce numerous status messages. When the build is complete, the newly compiled **`gcc`** executable can be installed. Again, as with the *binutils*, this must be done using the *superuser* or *root* account, since the files will be installed in a shared directory of the file system where they can be accessed by other users. The commands to install the *GCC* are:

```
$ su
password:
# make install
# exit
exit
$
```

Note that this install command copies a single executable file, **gcc**, into the **/usr/local/sparc-elf/bin/** directory. The previously installed **binutils** have already been installed in this directory. As with the *binutils*, manual pages and other supplementary material may have been installed by this command.

Once the *Gnu GCC* has been built and installed all of the source and object code used in this process may be deleted.

```
$ ls -la /usr/local/sparc-elf/bin/
total 18856
drwxr-sr-x  2 root staff    4096 Oct 25 11:44 .
drwxr-sr-x  4 root staff    4096 Aug 11 11:06 ..
-rwxr-xr-x  2 root staff 2256236 Oct 25 11:21 ar
-rwxr-xr-x  2 root staff 3692422 Oct 25 11:21 as
-rwxr-xr-x  1 root staff  318811 Oct 25 11:44 gcc
-rwxr-xr-x  2 root staff 2976077 Oct 25 11:21 ld
-rwxr-xr-x  2 root staff 2231737 Oct 25 11:21 nm
-rwxr-xr-x  2 root staff 2844181 Oct 25 11:21 objdump
-rwxr-xr-x  2 root staff 2256231 Oct 25 11:21 ranlib
-rwxr-xr-x  2 root staff 2679790 Oct 25 11:21 strip
$
```

These executables are various utilities used by the *Sparc GCC* compiler to build and manipulate libraries and object files.

Finally, note that the **/usr/local/bin** directory also contains copies of these binary executable utilities with "**sparc-elf-**" prefixed to their file names as shown below. In addition, other files relating to libraries are installed at **/usr/local/lib/gcc/sparc-elf/4.0.1/** and Manual pages at **/usr/local/man/man1** Other various support files may also be installed at various locations in the file system.

```
$ ls -la /usr/local/bin/sparc*
-rwxr-xr-x  1 root staff 2197207 Oct 25 11:21 /usr/local/bin/sparc-elf-addr2line
-rwxr-xr-x  2 root staff 2256236 Oct 25 11:21 /usr/local/bin/sparc-elf-ar
-rwxr-xr-x  2 root staff 3692422 Oct 25 11:21 /usr/local/bin/sparc-elf-as
-rwxr-xr-x  1 root staff 2146114 Oct 25 11:21 /usr/local/bin/sparc-elf-c++filt
-rwxr-xr-x  1 root staff  320034 Oct 25 11:44 /usr/local/bin/sparc-elf-cpp
-rwxr-xr-x  2 root staff  318811 Oct 25 11:44 /usr/local/bin/sparc-elf-gcc
-rwxr-xr-x  2 root staff  318811 Oct 25 11:44 /usr/local/bin/sparc-elf-gcc-4.0.1
-rwxr-xr-x  1 root staff   15721 Oct 25 11:44 /usr/local/bin/sparc-elf-gccbug
-rwxr-xr-x  1 root staff  131850 Oct 25 11:44 /usr/local/bin/sparc-elf-gcov
-rwxr-xr-x  2 root staff 2976077 Oct 25 11:21 /usr/local/bin/sparc-elf-ld
-rwxr-xr-x  2 root staff 2231737 Oct 25 11:21 /usr/local/bin/sparc-elf-nm
-rwxr-xr-x  1 root staff 2679791 Oct 25 11:21 /usr/local/bin/sparc-elf-objcopy
-rwxr-xr-x  2 root staff 2844181 Oct 25 11:21 /usr/local/bin/sparc-elf-objdump
-rwxr-xr-x  2 root staff 2256231 Oct 25 11:21 /usr/local/bin/sparc-elf-ranlib
-rwxr-xr-x  1 root staff  440311 Oct 25 11:21 /usr/local/bin/sparc-elf-readelf
-rwxr-xr-x  1 root staff 2100147 Oct 25 11:21 /usr/local/bin/sparc-elf-size
-rwxr-xr-x  1 root staff 2072526 Oct 25 11:21 /usr/local/bin/sparc-elf-strings
-rwxr-xr-x  2 root staff 2679790 Oct 25 11:21 /usr/local/bin/sparc-elf-strip
$
```

Testing the Compiler

The basic test is just to be sure that the executable does execute This can be done by invoking the gcc command with a flag to print out the version. This also ensures that the correct executable is being references, not some older executable somewhere else in the file system. This gives:

```
$ /usr/local/sparc-elf/bin/gcc -v
Using built-in specs.
Target: sparc-elf
Configured with: ../configure --target=sparc-elf --
with-newlib --without-headers --with-gnu-as --with-gnu-
ld --enable-languages=c --disable-nls
Thread model: single
gcc version 4.0.1
$
```

The next step is to actually compile a small piece of code and verify that it produces a proper *ELF* file. Any simple C source file will work. The test file **`Test.c`** below simply adds two integers.

```
int main(int argc, char *argv[]) {

   int  x = 1;
   int  y = 2;
   int  z;

   z = x + y;

}  /* end main() */
```

A simple attempt to compile this file results in the following error:

```
$ /usr/local/sparc-elf/bin/gcc -g Test.c
/usr/local/lib/gcc/sparc-elf/4.0.1/../../../../sparc-
elf/bin/ld: crt0.o: No such file: No such file or
directory
collect2: ld returned 1 exit status
$
```

What this means is that the linker could not find the crt0.o file, which is the file containing various default and initialization code expected by the compiler. One approach is to provide a crt0 file by modifying an existing file of creating a new one from scratch. These files are usually implemented in assembly language and require some fairly specialized knowledge of the processor. For now, the alternative is to produce a linkable (non-executable) object file and see if the code in this file looks like the expected Sparc binary.

The command below produces a Test.o object file and returns with no error or status messages, a sign of successful compilation.

```
$ /usr/local/sparc-elf/bin/gcc -c -g -o Test.o Test.c
$
```

The **`sparc-elf-objdump`** utility can be used to view the internals of this file. The command below shows a valid and expected *Sparc* disassembly.

```
$ /usr/local/sparc-elf/bin/objdump -d Test.o

Test.o:     file format elf32-sparc

Disassembly of section .text:

00000000 <main>:
   0:   9d e3 bf 88     save  %sp, -120, %sp
   4:   f0 27 a0 44     st  %i0, [ %fp + 0x44 ]
   8:   f2 27 a0 48     st  %i1, [ %fp + 0x48 ]
   c:   82 10 20 01     mov  1, %g1
  10:   c2 27 bf ec     st  %g1, [ %fp + -20 ]
  14:   82 10 20 02     mov  2, %g1
  18:   c2 27 bf f0     st  %g1, [ %fp + -16 ]
  1c:   c4 07 bf ec     ld  [ %fp + -20 ], %g2
  20:   c2 07 bf f0     ld  [ %fp + -16 ], %g1
  24:   82 00 80 01     add  %g2, %g1, %g1
  28:   c2 27 bf f4     st  %g1, [ %fp + -12 ]
  2c:   81 e8 00 00     restore
  30:   81 c3 e0 08     retl
  34:   01 00 00 00     nop
$
```

Details of the *DWARF2* debug information can be found using the **`readelf`** utility via the command:

```
$ /usr/local/bin/sparc-elf-readelf --debug-dump Test.o
```

The output from this is long and somewhat cryptic and not reproduced here. But this may be used to verify that proper *DWARF2* has been included in the *ELF* file by the **`-g`** flag.

Now that a Sparc binary has been we have produced a successfully, it is time to return to the problem of getting an executable file rather than a linkable object file. The linker expects a crt0.o file to be present by default. But this can be changed and new instructions given to the linker. A linker directive file named **`Cmpware.lnk`** is shown in *Appendix A*. This file just places the main code in the default **`.text`** section at address zero, with other program data following. The linker command below links the object file into an executable *ELF* / *DWARF* file.

```
$ /usr/local/sparc-elf/bin/ld -g -e 0x0000 -T Cmpware.lnk
-o Test.elf Test.o
$
```

As with the compiler, successful operation results in no status or error output from the linker. To check the new **`Test.elf`** file produced by the command, we can use the same **`objdump`** command that was used with the relocatable object file. The result is below.

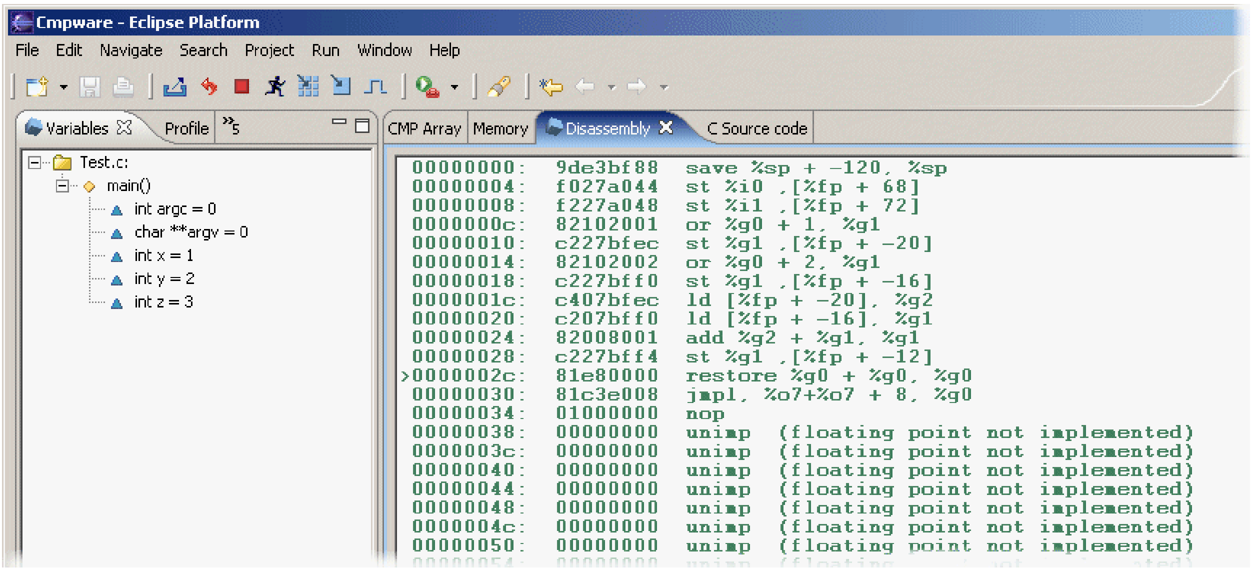
```
$ /usr/local/sparc-elf/bin/objdump -d Test.elf
Test.elf:     file format elf32-sparc

Disassembly of section .text:

00000000 <main>:
   0:   9d e3 bf 88     save  %sp, -120, %sp
   4:   f0 27 a0 44     st  %i0, [ %fp + 0x44 ]
   8:   f2 27 a0 48     st  %i1, [ %fp + 0x48 ]
   c:   82 10 20 01     mov  1, %g1
  10:   c2 27 bf ec     st  %g1, [ %fp + -20 ]
  14:   82 10 20 02     mov  2, %g1
  18:   c2 27 bf f0     st  %g1, [ %fp + -16 ]
  1c:   c4 07 bf ec     ld  [ %fp + -20 ], %g2
  20:   c2 07 bf f0     ld  [ %fp + -16 ], %g1
  24:   82 00 80 01     add  %g2, %g1, %g1
  28:   c2 27 bf f4     st  %g1, [ %fp + -12 ]
  2c:   81 e8 00 00     restore
  30:   81 c3 e0 08     retl
  34:   01 00 00 00     nop
$
```

While this appears similar to the **`Test.o`** file, the linker has resolved all relocatable references and fixed the addresses of code and variables. This file is now in '*executable*' format rather than linkable object. Further probing with the **`objdump`** utility confirms that **`Test.elf`** is indeed an executable *ELF* file with all symbols resolved and is in '*little endian*' format. See the '--help' command line flag for more information on the **`objdump`** command.

Using the Cmpware CMP-DK

Once the operation of the *GCC* compiler is verified, it can be used in conjunction with the *Cmpware CMP-DK* multiprocessor software development environment. To test the operation of the compiled test code, a *Sparc* processor or group of processors should be configured and the **`Test.elf`** file loaded into processor memory. If the source file **`Test.c`** is in the same location as when the file was compiled, or is in the same directory as the **`Test.elf`** file, the *Cmpware CMP-DK* should provide source-level tracing as well as a display of source level variables as execution proceeds.

While the operation of the Cmpware CMP-DK is beyond the scope of this document, a four processor configuration of *Sparc* processors was set up and the **`Test.elf`** file loaded into all four processors. The figure below shows the **Disassembly** and **Variables** views after several cycles of execution, and the sum of the two integers is plainly seen.


Cmpware - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Variables
Profile
CMP Array
Memory
Disassembly
C Source code

```
Test.c:
  main()
    int argc = 0
    char **argv = 0
    int x = 1
    int y = 2
    int z = 3
```

```
 00000000:  9de3bf88  save %sp + -120, %sp
 00000004:  f027a044  st %i0 ,[%fp + 68]
 00000008:  f227a048  st %i1 ,[%fp + 72]
 0000000c:  82102001  or %g0 + 1, %g1
 00000010:  c227bfec  st %g1 ,[%fp + -20]
 00000014:  82102002  or %g0 + 2, %g1
 00000018:  c227bff0  st %g1 ,[%fp + -16]
 0000001c:  c407bfec  ld [%fp + -20], %g2
 00000020:  c207bff0  ld [%fp + -16], %g1
 00000024:  82008001  add %g2 + %g1, %g1
 00000028:  c227bff4  st %g1 ,[%fp + -12]
>0000002c:  81e80000  restore %g0 + %g0, %g0
 00000030:  81c3e008  jmpl, %o7+%o7 + 8, %g0
 00000034:  01000000  nop
 00000038:  00000000  unimp  (floating point not implemented)
 0000003c:  00000000  unimp  (floating point not implemented)
 00000040:  00000000  unimp  (floating point not implemented)
 00000044:  00000000  unimp  (floating point not implemented)
 00000048:  00000000  unimp  (floating point not implemented)
 0000004c:  00000000  unimp  (floating point not implemented)
 00000050:  00000000  unimp  (floating point not implemented)
```

Conclusions

This document describes the downloading, building and use of the *Gnu GCC* compiler in a cross targeted environment. In this case, a Linux host is used to create a *GCC C* compiler for the *Sparc* processor. While this procedure may be used to build other cross targeted compilers, the end results may vary substantially depending on the particular host, *GCC* distribution and processor selected. In many cases, older of less popular processors may require significant modification of the source code to produce a working compiler. This may be a substantial development effort depending on the development environment required. But for more popular host platforms and target processors, the procedure can be accomplished in a few minutes with the commands described in this document.

Finally, it is possible to use of the *GCC* compiler within the *Eclipse* environment used by the *Cmpware CMP-DK*. The *Eclipse C Development Toolkit (CDT)* permits *Makefiles* to be created and edited and builds managed from within the IDE. This, combined with the Cmpware CMP-DK makes a very powerful programming environment for multiprocessors. The installation and use of the *Eclipse CDT* is beyond the scope of this document, but more information can be found at: ***http://www.eclipse.org/cdt/*** Care should be taken in installing the CDT in *Eclipse*. Version compatibility has been an issue. Be sure to read all applicable documents to ensure that the correct version for your system is installed.

Appendix A - The Cmpware.lnk Linker Directive File

```
/*
**  Copyright (c) 2004 Cmpware, Inc.  All rights reserved.
*/

_gp = ABSOLUTE(. + 0x7ff0);

SECTIONS {
   .text 0 : { *(.text) }
   .rodata ALIGN(8) : { *(.rodata) }
   .sdata ALIGN(8) : { *(.sdata) }
   .data ALIGN(8) : { *(.data) }
   .bss  ALIGN(8) : { *(.bss) }
   /* DWARF 1 */
   .debug          0 : { *(.debug) }
   .line           0 : { *(.line) }
   /* GNU DWARF 1 extensions */
   .debug_srcinfo  0 : { *(.debug_srcinfo) }
   .debug_sfnames  0 : { *(.debug_sfnames) }
   /* DWARF 1.1 and DWARF 2 */
   .debug_aranges  0 : { *(.debug_aranges) }
   .debug_pubnames 0 : { *(.debug_pubnames) }
   /* DWARF 2 */
   .debug_info     0 : { *(.debug_info) }
   .debug_abbrev   0 : { *(.debug_abbrev) }
   .debug_line     0 : { *(.debug_line) }
   .debug_frame    0 : { *(.debug_frame) }
   .debug_str      0 : { *(.debug_str) }
   .debug_loc      0 : { *(.debug_loc) }
   .debug_macinfo  0 : { *(.debug_macinfo) }
}
```