

# EE Programming Tools

© 2001 Sony Computer Entertainment Inc.

Publication date: October 2001

Sony Computer Entertainment Inc.  
1-1, Akasaka 7-chome, Minato-ku  
Tokyo 107-0052, Japan

Sony Computer Entertainment America  
919 E. Hillsdale Blvd., 2nd floor  
Foster City, CA 94404

Sony Computer Entertainment Europe  
30 Golden Square  
London W1F 9LD, U.K.


The *EE Programming Tools* manual is supplied pursuant to and subject to the terms of the Sony Computer Entertainment PlayStation® license agreements.

The *EE Programming Tools* manual is intended for distribution to and use by only Sony Computer Entertainment licensed Developers and Publishers in accordance with the PlayStation® license agreements.

Unauthorized reproduction, distribution, lending, rental or disclosure to any third party, in whole or in part, of this book is expressly prohibited by law and by the terms of the Sony Computer Entertainment PlayStation® license agreements.

Ownership of the physical property of the book is retained by and reserved by Sony Computer Entertainment. Alteration to or deletion, in whole or in part, of the book, its presentation, or its contents is prohibited.

The information in the *EE Programming Tools* manual is subject to change without notice. The content of this book is Confidential Information of Sony Computer Entertainment.

 and PlayStation are registered trademarks of Sony Computer Entertainment Inc. All other trademarks are property of their respective owners and/or their licensors.

# Table of Contents

<b>About This Manual</b>	<b>v</b>
Changes Since Last Release	v
Related Documentation	v
Typographic Conventions	v
Developer Support	v
<b>Overview</b>	<b>1</b>
<b>ee-addr2line</b>	<b>2</b>
<b>ee-ar</b>	<b>3</b>
<b>ee-as</b>	<b>5</b>
<b>ee-c++ / ee-g++</b>	<b>6</b>
<b>ee-c++filt</b>	<b>7</b>
<b>ee-dvp-as</b>	<b>8</b>
VU Micro Instruction Notation	8
VIF Code Notation	9
DMA Tag Notation	10
Embedded VIF Code Control	10
GIF Tag Notation	11
<b>ee-gcc</b>	<b>13</b>
ee-gcc options	13
Inline assembler	17
<b>ee-gdb</b>	<b>20</b>
<b>ee-ld</b>	<b>21</b>
<b>ee-nm</b>	<b>22</b>
<b>ee-objcopy</b>	<b>23</b>
<b>ee-objdump</b>	<b>25</b>
<b>ee-protolize / ee-unprotolize</b>	<b>26</b>
<b>ee-ranlib</b>	<b>28</b>
<b>ee-readelf</b>	<b>29</b>
<b>ee-size</b>	<b>30</b>
<b>ee-strings</b>	<b>31</b>
<b>ee-strip</b>	<b>32</b>
<b>make</b>	<b>33</b>
Procedure File Contents and Syntax	34



## About This Manual

This is the Runtime Library Release 2.4 version of the *EE Programming Tools* manual.

It provides general descriptions of the various EE programming tools, which are part of the Tool-Chain provided by Sony Computer Entertainment for Playstation 2 development.

## Changes Since Last Release

New

## Related Documentation

The companion "IOP Programming Tools" manual provides detailed information on IOP programming tools.

**Note:** the Developer Support Web site posts current developments regarding the Libraries and also provides notice of future documentation releases and upgrades.

## Typographic Conventions

Certain Typographic Conventions are used throughout this manual to clarify the meaning of the text:

Convention	Meaning
<code>courier</code>	Indicates literal program code.
<i>italic</i>	Indicates names of arguments and structure members (in structure/function definitions only).
<b>medium bold</b>	Indicates data types and structure/function names (in structure/function definitions only).
<a href="#">blue</a>	Indicates a hyperlink.

## Developer Support

### Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
<i>In North America:</i>	<i>In North America:</i>
Attn: Developer Tools Coordinator	E-mail: <a href="mailto:PS2_Support@playstation.sony.com">PS2_Support@playstation.sony.com</a>
Sony Computer Entertainment America	Web: <a href="http://www.devnet.scea.com/">http://www.devnet.scea.com/</a>
919 East Hillsdale Blvd.	Developer Support Hotline: (650) 655-5566
Foster City, CA 94404, U.S.A.	(Call Monday through Friday,
Tel: (650) 655-8000	8 a.m. to 5 p.m., PST/PDT)

**Sony Computer Entertainment Europe (SCEE)**

SCEE developer support is available to licensees in Europe only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
<i>In Europe:</i> Attn: Production Coordinator Sony Computer Entertainment Europe 30 Golden Square London W1F 9LD, U.K. Tel: +44 (0) 20 7859-5000	<i>In Europe:</i> E-mail: <a href="mailto:ps2_support@scee.net">ps2_support@scee.net</a> Web: <a href="https://www.ps2-pro.com/">https://www.ps2-pro.com/</a> Developer Support Hotline: +44 (0) 20 7859-5777 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT)

---

## Overview

This document gives general descriptions of the various EE programming tools, which are part of the Tool-Chain provided by Sony Computer Entertainment for Playstation 2 development.

Details on how these tools are used can be obtained using the "info" command. For example, to find out how to use ee-gcc, type the following command:

```
% info -f /usr/local/sce/ee/gcc/info/gcc.info
```

These tools are derived from GNU software and have been updated for use with the EE. Commercially available Unix programming books can also be a useful source of information.

dsedb is described in a separate document.

## ee-addr2line

ee-addr2line is a tool that accesses debugging information embedded in an executable program (elf file) and displays the source code corresponding to given addresses.

### Example

To find the source code locations corresponding to addresses 0xa022 and 0xa026 in test.elf, type the following:

```
% addr2line -e test.elf a022 a026
```

### Synopsis

```
ee-addr2line [ options... ] [ addr... ]
```

The main *options...* that can be specified are shown below.

**Table 1**

option	Function
-C	Return and output the source file symbol name for the processing system's low-level symbol name. Useful for C++ programs.
<i>e filename</i>	Specify the name of the executable file in ' <i>filename</i> '. If omitted, "a.out" is assumed.
-f	Append to output the function name corresponding to the specified address.
-s	Output only the base name as the filename.
-H	Output help for this function.
-V	Output version information.

"*addr...*" indicates the target address, in hex. A "0x" prefix is not necessary when specifying the address. If "*addr...*" is omitted, input will be taken from stdin.

### Detailed description

Outputs the filename and line number of the source program corresponding to the specified addresses, one entry per line, in the format

```
filename:linenumber
```

If the -f option is specified, the function name will be included as follows:

```
functionname
```

```
filename:linenumber
```

If the function name or filename is unknown, "??" will be output instead. If the line number is unknown, "0" will be output in its place.



## ee-ar

ee-ar is a maintenance tool used to create an archive file (library file: \*.a) from a collection of object files (\*.o). ee-ar can also be used to collect together multiple files regardless of file type.

### Example

To create an archive called "moving.a" that contains three object files "playermove.o", "enemymove.o", and "checkmap.o", execute the following command:

```
% ee-ar r s moving.a playermove.o enemymove.o checkmap.o
```

Later, to insert an updated version of enemymove.o:

```
% ee-ar r s moving.a enemymove.o
```

You can also confirm which object files are included in moving.a with the following command:

```
% ee-ar t v moving.a
```

### Synopsis

```
ee-ar operation[options...] archive [members...]
```

"operation" indicates what kind of action will be taken on the *archive* and its *members*, so an operation must always be specified. "options..." give more details about the action, and their use will differ depending on the operation. *operation* and *options* can appear in any order, and do not need to be preceded with hyphens.

Table 2

<i>operation</i>	Function	<i>options...</i>
d	Delete the specified <i>members...</i> from <i>archive</i> .	v: Output the names of the deleted members.
m	Move the specified <i>members...</i> within <i>archive</i> .	None: Move to the end of the archive. a <i>m</i> : Move to point directly after member <i>m</i> b <i>m</i> : Move to point directly before member <i>m</i> i <i>m</i> : Move to point directly before member <i>m</i>
p	Output the specified <i>members...</i> from <i>archive</i> to stdout.	v: Output the member names.
q	Add the files specified by ' <i>members...</i> ' to the end of <i>archive</i> .	v: Output the names of the added members. a/b/i are ignored.

<i>operation</i>	Function	<i>options...</i>
r	Add the files specified by ' <i>members...</i> ' to <i>archive</i> .	None: Add to the end of the archive. a <i>m</i> : Add to point directly after member <i>m</i> . b <i>m</i> : Add to point directly before member <i>m</i> . i <i>m</i> : Add to point directly before member <i>m</i> . u: Compare timestamps and add updated <i>members...</i> only v: Output names of added members and indicate whether newly added.
t	List the members in the <i>archive</i> .	v: Output file attributes for each member.
x	Extract specified <i>members...</i> from <i>archive</i> and write out to files with the same name.	o: Make the timestamps agree with those in the archive. v: Output names of extracted members.

Adding a file using the "q" operation will not change the index of files in the archive. Moreover, if a file with the same name is already present in the archive, the specified file will be added as a new member.

When the "r" operation is used to add a file, a check is first performed to see if a file with that name already appears in the archive, and if so, it is deleted first, then replaced with the specified file. If the "v" option is also specified, the newly added file will be displayed with an "a", and the replaced file with an "r".

The "t" operation is normally used without specifying any *members...*. In this case, it will output all the members of the archive. However, if *members...* are specified, information about those members will be output first.

When extracting files with the "x" operation, if no *members...* are specified, all members contained in the *archive* will be extracted and written out to files.

Additionally, the following *options...* can also be specified with this command.

**Table 3**

<i>options...</i>	Function
c	Inhibit warnings when creating new archive file if <i>archive</i> doesn't exist.
f	Reduce length of member names in accordance with system file naming rules.
l	(no operation)
s	Update <i>archive</i> index. If no index exists, creates one.
S	Do not create index.
V	Display version information.

The "s" option can be used with any *operation*, and can be specified even without indicating an *operation*.

### Description

See the descriptions of the various *operations* and *options...* above.

---

## **ee-as**

This is the assembler for the EE. It can be called when needed from ee-gcc, or can be used on its own.

---

## **ee-c++ / ee-g++**

ee-c++ and ee-g++ are the C++ compilers for the EE.

C++ programs can be compiled with ee-gcc, but to link to individual C++ libraries, it's more convenient to use ee-c++ and ee-g++.

---

## ee-c++filt

In order for the C++ compiler to perform overloading, assembly source is output after converting symbol names using certain naming rules. This utility takes the assembler symbol names and converts them back to their original names from the C++ program.

### Synopsis

```
ee-c++filt [options...] [symbols...]
```

Where "*symbols...*" specifies the assembler symbol names to be converted. If "*symbols...*" isn't specified, the assembler symbol names will be read and converted from stdin.

The main *options...* that can be specified are shown below.

**Table 4**

option	Function
-_	Delete underscores appearing before <i>symbols..names</i> . Used when the compiler prepends underscores to low-level symbol names.
-n	Do not delete underscores appearing before <i>symbols...</i> names. Used when the compiler does not prepend underscores to low-level symbol names.

## ee-dvp-as

ee-dvp-as is the assembler that supports the EE's built-in vector processor (VU).

VU programming can use inline notation, and can describe VIF packets, GIF packets, or DMA packets.

### Synopsis

```
% ee-dvp-as [options...] files...
```

The main *options...* that can be specified are shown below.

**Table 5**

option	Function
-a[dlns]	Controls assembly list output. Sub-options are as follows: d: Do not output pseudo-instructions for debug. l: Output assembly language listing. n: Do not perform page formatting based on .psize, .title, etc. s: Output symbol table listing.
-I <i>path</i>	Specify search path for include files.
-L	Include local labels in object.
-no-dma	Do not generate code for DMA commands.
-no-dma-vif	Do not generate code for DMA or VIF commands.
-o <i>filename</i>	Specify name of object file.
-W	Do not output warning messages.
-Z	Create object file even if errors occurred.
-v	Display assembler version.

## VU Micro Instruction Notation

When the .vu pseudo-instruction is placed in a program, subsequent VU micro instructions can then be included.

Example:

```
.vu
add.xyzw vf10.xyzw vf20.xyzw vf21.xyzw    NOP
```

To set the I bit with a VU instruction, that is, to transfer the lower 32 bits of the instruction to the I register, the LOI pseudo-instruction can be used. As shown below, the LOI instruction is used as a Lower instruction.

Example:

```
add.xyzw vf10.xyzw vf20.xyzw vf21.xyzw    LOI 3.1415926
```

To set the E, M, D, or T bits, the respective letter should be enclosed in square brackets and appended to an Upper command. The individual letters are not case-sensitive.

Example:

```
NOP[T] IADDIU VI01, VI00, LOOP
```

## VIF Code Notation

VIF codes, which represent instructions for the VIF, can be generated using the following opcodes.

**Table 6**

Opcode	Function
base	Set VIF1 base register.
direct/directh1	Transfer data via PATH2.
flush/flusha/flushh	Wait until VU is idle.
itop	Set VIF1 ITOPS register.
mark	Set VIF MARK register.
mpg	Load VU microprogram .
mscal/mscalf	Execute VU microprogram.
mscnt	Resume interrupted VU microprogram.
mskpath3	Control PATH3 data transfer.
offset	Set VIF1 offset register.
stcol	Set VIF Col register.
stcycl	Set VIF CYCLE register.
stmask	Set VIF MASK register.
stmod	Set VIF MODE register.
strow	Set VIF Row register.
unpack	Unpack and load data in VU memory.
vifnop	No operation.

The following options are enclosed in square brackets and can follow each opcode.

**Table 7**

Option	Function
i	Generate interrupt.
m	Mask (unpack instruction only).
r	TOPS-relative address (unpack instruction only).
u	UNPACK without encoding (unpack instruction only).

Although a VU microprogram follows an mpg instruction, and a data list follows an unpack instruction, the following pseudo-instructions can be used to terminate these instructions.

**Table 8**

Pseudo-instruction	Function
.EndMpg	Terminate VU instruction list following mpg instruction.
.EndUnpack	Terminate data list after following instruction.

For details on VIF code functions, refer to the EE User's Manual.

## DMA Tag Notation

DMA tags, which control source-chain DMA (DMA transfers from memory to peripherals), can be generated by including the following opcodes.

**Table 9**

Opcode	Function
DMAcnt[ <i>opts</i> ] <i>qwc</i>	Transfer following data, continue processing to following tag.
DMAnext[ <i>opts</i> ] <i>qwc</i> , <i>addr</i>	Transfer following data, continue processing to tag specified by <i>addr</i> .
DMAref[ <i>opts</i> ] <i>qwc</i> , <i>addr</i>	Transfer data specified by <i>addr</i> , continue processing to following tag.
DMArefe[ <i>opts</i> ] <i>qwc</i> , <i>addr</i>	Transfer data specified by <i>addr</i> and terminate processing.
DMArefs[ <i>opts</i> ] <i>qwc</i> , <i>addr</i>	Transfer data specified by <i>addr</i> (store control), continue processing to following tag.
DMAcall[ <i>opts</i> ] <i>qwc</i> , <i>addr</i>	Transfer following data, push address of following data, continue processing to tag specified by <i>addr</i> .
DMAret[ <i>opts</i> ] <i>qwc</i>	Transfer following data, continue processing to popped tag.
DMAend[ <i>opts</i> ] <i>qwc</i>	Transfer following data and terminate processing.

Destination-chain DMA (transfer from peripherals to memory) is not supported. For details on the various DMA tags, refer to the EE User's Manual.

Options, specified as [*opts*], are enclosed in square brackets and shown below.

**Table 10**

Option	Function
0	Disable priority transfer (D_PCR.PCE=0).
1	Enable priority transfer (D_PCR.PCE=1).
I	Request interrupt when transfer complete.
S	SPR address (disabled for ch-0, 1, 2).

*qwc* indicates the size of the data to transfer. When the data to transfer is included using the `.DmaData-`  
`.EndDmaData` pseudo-instructions, setting *qwc* to "\*" allows the data size to be calculated automatically if the labels are set for *addr*.

## Embedded VIF Code Control

DMA tags require 16-byte alignment, but are only 8 bytes in length themselves. Consequently, the 4-byte VIF codes can be embedded in part of the unused upper 8 bytes of transfer data. This packing process can be controlled using the `.DmaPackVif` pseudo-instruction.

By default, the two VIF codes immediately following the DMA tag will be packed into the upper part of the DMA tag. When `".DmaPackVif 0"` is used, subsequent packing will not be performed and the upper part of the DMA tags will remain unused (to be more precise, a VIFNOP is embedded). `".DmaPackVif 1"` can be used to turn packing back on.



### Sample DMA tag syntax

```

DMAref[I1] *, data1
DMAend 4
.word 0, 0, 0, 0
.EndDmaData
.DmaData data1
MPG *, *
NOP IADDIU VI01, VI00, 0x00000100
NOP LQI.xyzw VF04, (VI01++)
NOP[d] LQI.xyzw VF05, (VI01++)
NOP LQI.xyzw VF06, (VI01++)
NOP LQI.xyzw VF07, (VI01++)
NOP[e] NOP
NOP NOP
.endmpg
.EndDmaData

```

### GIF Tag Notation

GIF tags, which are prepended to data (GS primitives) for transfer to the GS, can be generated using the following three opcodes.

**Table 11**

Opcode	Function
GIFpacked	Transfer data to GS in PACKED mode.
GIFreglist	Transfer data to GS in REGLIST mode.
GIFimage	Transfer data to GS in IMAGE mode.

Each of these GIF tag opcodes is followed by a data description, and terminated with .EndGif. For more details on GIF tag functionality, GS primitives, and the overall structure of GIF packets, please see the EE User's Manual.

#### GIFpacked: transfer data in PACKED mode

```

GIFpacked [PRIM=0Xxx,] REGS={ r1, r2, .. } , [NLOOP=c,] [EOP]
... data description ..
.EndGif

```

PRIM=0Xxx is set to the value transferred to the PRIM register; xx is the lower 11 bits transferred and other bits are ignored.

REGS={ *r1*, *r2*, ... } describes a set of registers that specify the transfer destinations for the packed-mode data (128 bits) which follows the GIFpacked directive. *r1* is the first destination, *r2* the second, etc. From 1 to 16 registers can be specified.

**Table 12**

Register name	Value	Destination
PRIM	0x00	PRIM register
RGBAQ	0x01	RGBAQ register
ST	0x02	ST register
UV	0x03	UV register
XYZF2	0x04	XYZF2 register, or depending on data value, XYZF3 register

Register name	Value	Destination
XYZ2	0x05	XYZ2 register, or depending on data value, XYZ3 register
TEX0_1	0x06	TEX0_1 register
TEX0_2	0x07	TEX0_2 register
CLAMP_1	0x08	CLAMP_1 register
CLAMP_2	0x09	CLAMP_2 register
XYZF	0x0a	XYZF register
RESERVED	0x0b	(reserved)
XYZF3	0x0c	XYZF3 register
XYZ3	0x0d	XYZ3 register
A_D	0x0e	Register specified by upper 64 bits of data
NOP	0x0f	(None: data discarded, nothing transferred to GS)

NLOOP=*c* specifies the number of iterations through the data following the GIFpacked directive. In other words, the data transfers set up according to REGS={ *r1*, *r2*, .. } are repeated "*c*" times. Only the lower 15 bits of "*c*" are used. If "*c*" is omitted, it is calculated automatically from the position of the .EndGif pseudo-instruction.

EOP indicates the final GS packet. Only use EOP at the end.

#### **GIFreglist: transfer using REGLIST mode**

```
GIFreglist  REGS={ r1, r2, .. } , [NLOOP=c,] [EOP]
... data ...
.EndGif
```

REGS={ *r1*, *r2*, ... } describes a set of registers that specify the transfer destinations for the packed-mode data (64 bits) which follows the GIFreglist. The specification is similar to the GIFpacked directive. *r1* is the first destination, *r2* the second, etc. From 1 to 16 registers can be specified.

NLOOP=*c* specifies the number of iterations through the data following the GIFreglist directive. In other words, the data transfers set up according to REGS={ *r1*, *r2*, .. } are repeated "*c*" times. Only the lower 15 bits of "*c*" are used. If "*c*" is omitted, it is calculated automatically from the position of the .EndGif pseudo-instruction. Note, however, that since GS packets are 128 bits wide, if there are an odd number of data units to transfer, the upper 64 bits of the last 128-bit unit will be ignored.

EOP indicates the final GS packet. Only use EOP at the end.

#### **GIFimage: transfer using image mode**

```
GIFimage  [NLOOP=c,] [EOP]
... data ...
.EndGif
```

NLOOP=*c* is used to set the size of the data following the GIFimage directive in qwords. Only the lower 15 bits of "*c*" are used. If "*c*" is omitted, it is calculated automatically from the position of the .EndGif pseudo-command.

EOP indicates the final GS packet. Only use EOP at the end.

#### **Sample GIF tag description**

```
GIFpacked REGS={A_D}, NLOOP=1, EOP
iwzyx 0x00000000, 0x0000007f, 0x00000000, 0x00000000
.endgif
```

## ee-gcc

ee-gcc is a C/C++ compiler that is able to call the assembler and linker as needed. It determines the appropriate action to take based on the suffixes of the input files. You can also use ee-gcc just for assembling or linking.

### Example

To compile hello.c into an executable hello.elf, type the following:

```
% ee-gcc -o hello.elf -T app.cmd crt0.o hello.c
```

Here, app.cmd is a link script file that contains directions for linking, and crt0.o is the start-up object that runs up to the point where main() is called in the user program. In actual use, you would fill in the appropriate path names.

### Synopsis

```
% ee-gcc [options...] files... [linkoptions...]
```

There are many *options...* that can be used with this command and they are summarized below.

### Description

The contents of files are determined from the suffixes of the input files specified by "*files...*".

Subsequently, appropriate programs are called to execute each stage of the preprocess > compile > assemble > link flow to create an executable file ("elf" file). By specifying options, processing can also be limited to specific steps.

**Table 13**

Suffix	Description
<i>file.c</i>	Assumed to be C source file. Executes each step from preprocess on.
<i>file.i</i>	Assumed to be preprocessed C source file. Executes each step from compile on.
<i>file.ii</i>	Assumed to be preprocessed C++ source file. Executes each step from compile on.
<i>file.h</i>	Assumed to be C header file.
<i>file.cc</i>	Assumed to be C++ source files.
<i>file.cxx</i>	Executes each step from compile on.*
<i>file.cpp</i>	
<i>file.C</i>	
<i>file.s</i>	Assumed to be assembler source file. Executes each step from assemble on.
<i>file.S</i>	Assumed to be assembler source file. Executes each step from preprocess on.
other	Assumed to be object file; performs linking.

\* Any two characters following "c", expressed here as "cxx"

## ee-gcc options

The ee-gcc command supports the options shown below. Except where otherwise noted, all of these options can be used regardless of the source program language.

## General option format

The gcc command includes many options with multiple-letter names. Note that these options cannot be specified with a single letter.

When specifying the same option repeatedly, the order of appearance will affect their meaning. For example, with the -L option, directories will be searched in the order they appear.

There are many options with long names that begin with -f or -W which come in positive/negative pairings. For example the -fasm and -fno-asm options enable and disable the inline-assembler, respectively. Note that the inline assembler is enabled by default, so normally -fasm need not be specified. The description of default options, like -fasm, has been omitted in this document.

## General options

Options that control gcc's overall operation are shown below.

**Table 14**

Option	Function
-c	Create .o file. (process through assembly only -- do not link)
-S	Create .s file (do not proceed to linking).
-E	Write contents equivalent to a .i file to stdout (preprocessing only).
-I <i>dir</i>	Add <i>dir</i> to directory list for header-file search (there should be no space between "-I" and " <i>dir</i> ").
-I-	Apply system header files represented by #include < <i>file</i> > to the -I <i>dir</i> option (must appear before -I <i>dir</i> option).
-L <i>dir</i>	Add <i>dir</i> to directory list for library file search (there should be no space between "-L" and " <i>dir</i> ").
-o <i>file</i>	Use " <i>file</i> " as the output file. If omitted, the executable output file will be named a.out. For all other file types, the original file's name is used with an appropriate suffix.
-pipe	Does not generate intermediate files, transfers using a pipe.
-v	Display detailed messages regarding execution status.
-x <i>language</i>	Without attempting to determine the input file type by its suffix, forces file to be treated as if it were specified for " <i>language</i> ". Effective until next -x option is specified.  " <i>language</i> " can be any of the following: c c++ c-header cpp-output c++-cpp-output assembler l assembler-with-cpp none

## Preprocessor control options

Options that control the preprocessor are shown below. Some are used to define macros or perform conditional compilation for debugging. In these cases because the generated output cannot be properly compiled, the -E option should also be used.

Table 15

Option	Function
-C	Do not delete comment lines. (must be used with -E)
-dM	Output list of valid <i>macro</i> definitions. (must be used with -E)
-D <i>macro</i> [= <i>def</i> ]	Define macro " <i>macro</i> ". If " <i>=def</i> " is omitted, it is defined to be "1".*
-E	Preprocess only, write equivalent of a .i file to stdout.
-H	Display names of included header files.
-include <i>file</i>	Include file " <i>file</i> " at start of preprocessing.*
-imacros <i>file</i>	Include file " <i>file</i> " at start of preprocessing. Read in only macro definitions, ignore everything else.
-M	Search included header files, output makefile to stdout. (Even if -E is not specified, ends after preprocessing.)
-nostdinc	Search only the directories defined by <i>-ldir</i> . without searching predefined header-file directories.
-P	Suppress output for "#line". (must be used with -E)
-U <i>macro</i>	Delete macro definition " <i>macro</i> ".*

\* Processing related to macro definitions follows this sequence:

After macros are defined with -D / -include / -imacros, they should be deleted with -U.

\* The following macros are defined by default: `_mips_ / _ee_ / _MIPSEL_`

## C language options

The following options control details related to the C language specification.

Table 16

Option	Function
-ansi	Follow ANSI C language spec (turn off special GNU extensions).
-fcond-mismatch	Allow different 2nd-operand and 3rd-operand types for conditional operations, assume void types.
-fsigned-bitfields	Treat bitfields without a sign as signed.
-fsigned-char	Treat char as signed.
-funsigned-bitfields	Treat bitfields without a sign as unsigned.
-funsigned-char	Treat char as unsigned.
-fwritable-strings	Permit writing to string constants.
-traditional	Follow K&R style.

## C++ language options

The following options control details related to the C++ language specification.

Table 17

Option	Function
-fall-virtual	Handle member functions as virtual functions.
-fdollars-in-identifiers	Allow "\$" in identifier names.
-fenum-int-equiv	Allow implicit conversion from int to enum type.
-fexternal-templates	Only generate code for template functions where defined.

### Warning options

A warning is a message that indicates a place where non-syntactical programming errors have been detected. There are many options for controlling the behavior of warnings, such as specifying whether to output each warning based on content. Since this fine-grained level of control is rarely needed, the following is a list of options that only specify severe checks.

**Table 18**

Option	Function
-w	Do not issue warnings.
-W	Issue warnings for several specific cases where improvements can be easily made.
-Wall	Issue warnings for several other cases in addition to -W.
-Werror	Assume warnings are equivalent to errors and interrupt compilation.
-pedantic	Check warnings as specified in the ANSI C language specification.
-pedantic-errors	Perform same checks as -pedantic, but assume warnings are equivalent to errors and interrupt compilation.

### Debugging options

There are many options related to debugging, but the most commonly used one is shown below. It is used to embed debugging information in an object. Additionally, there are options that will generate snapshot dumps at every stage of the compilation process: syntactical analysis, flow analysis, register allocation, common element elimination, jump optimization, instruction generation, etc.

**Table 19**

Option	Function
-g	Embed debugging information in object.

### Optimization options

There are many options to control the optimization process, and each of the various optimization methods can be controlled independently. The following options can be used to control the optimization process in general.

**Table 20**

Option	Function
-O0	No optimization (default)
-O	Perform normal optimization.
-O1	
-O2	Perform more thorough optimization.
-O3	In addition to -O2, optimize for greater speed.
-Os	Optimize at roughly the same level as -O2, but for smaller code size.

## MIPS-specific options

The following are options that control functions specific to the MIPS family of processors.

Table 21

Option	Function
-mhard-float	Generate floating-point instructions (default),
msoft-float	Instead of floating-point instructions, generate code that calls floating-point library functions.

\* To match the characteristics of the EE core, -msingle-float (coprocessor supports single-precision floating-point calculations) and -EL (little-endian) are set by default.

## Code-generation options

Options that control the details of the code-generation process are shown below.

Table 22

Option	Function
-Gnum	Places global data/static data up to "num" bytes in the .sdata/.sbss section, but not in the .data/.bss section. While two instructions are normally needed to access data, standardizing the global pointer (gp or \$28) makes access possible with one instruction. "num" is 8 by default; if set to 0, access using the global pointer is prevented.
-fshort-double	Sets double to same size as float.

## Assembler control options

Options that control the assembler's operation are shown below.

Table 23

Option	Function
-Wa,option	Passes "option" as is as an option to the assembler.

## Link control options

Options that control the linking process are shown below.

Table 24

Option	Function
-Wl,option	Passes "option" as is as an option to the linker.

## Inline assembler

Using the "asm" directive in a C/C++ program makes it possible to insert assembly code. By including the appropriate parameters and I/O operands, C/C++ variables can be used as is in the assembly code (without having to deal with target registers and addresses).

### Synopsis

```
asm [volatile] ( asmstring : [output] : [input] [ : modify] );
```

*asmstring*: An assembly code template that can include parameters (%0, %1...%9). If *asmstring* consists of multiple assembly instructions, separate them with semicolons or newlines.

*output*: List of output operands  
*input*: List of input operands  
*modify*: List of resources destroyed when *<asmstring>* executes.

### Assembly code template syntax

"*asmstring*" is given as a text literal representing a list of assembler instructions. When using multiple assembly instructions, separate them using semicolons or newlines. Local labels can be used.

When reading from or writing to variables used in the C/C++ program, or when referring to the value of an expression, insert a parameter (%0, %1...%9) at the appropriate point in the *asmstring*. The correspondence between parameters and variables or expressions is given using "output" and "input". That is, the order in which variables and expressions appear in *output* and *input* corresponds to the parameter number (%0, %1...%9) in the *asmstring*.

### Output operand syntax

C variables in the following forms are used in "*output*" for write-destinations within *asmstring*. When using multiple C variables, separate them using commas.

"=r" (*left-side value*)  
 "=f" (*left-side value*)  
 "=&r" (*left-side value*)  
 "=&f" (*left-side value*)

Here, the "=" indicates that it is the destination for a write, "r" is a general-purpose register (GPR), "f" is a COP1 floating-point register (FPR), and "&" indicates that the register is used exclusively. If the *asmstring* encompasses multiple instructions where a given C variable could be written and then referenced, you must ensure that a dedicated register is allocated for the write target, using the &r or &f notation.

### Input operand syntax

Expressions (variables) that are referenced within *asmstring* are represented as follows in "*input*".

"r" (*expression*)  
 "f" (*expression*)  
 "0" (*expression*)  
 ...  
 "8" (*expression*)

When a number "n" is specified, it means that the input operand is the same variable as the output operand that corresponds to %n. In other words, this notation is used to mean a dual-purpose read/write operand. Simply using the same variable name may not always result in the most appropriate results (depending on compiler optimizations, etc), so you should always use a number for a read/write operand.

### Destroyed resources list

"*modify*" contains names of registers whose values can change as a result of the execution of the *asmstring*. Multiple register names should be separated by commas. In addition, when the contents of memory can be changed or accessed, the word "memory" should be included as well.

If the destroyed resources list is not properly described, the compiler may generate code with unintended behavior during optimization.



## volatile keyword

The `volatile` keyword is used when there is no output operand. An asm statement without an *output* section may not always generate code because of compiler optimization. As a result, you must place the "`volatile`" keyword at the beginning.

## Register names

Register names that can be used in assembly code are given below.

**Table 25**

Register	Name
EE Core general purpose registers (GPR)	\$0...\$31
FPU general purpose registers (FPR)	\$f0...\$f31
VU0 floating-point registers	\$vf0...\$vf31
VU0 integer registers	\$vi0...\$vi15

The "\$" can be omitted from VU0 register names. With some of the EE Core general purpose registers, the following aliases can be used instead.

**Table 26**

Register name	Alias
\$1	\$at
\$26	\$kt0
\$27	\$kt1
\$28	\$gp
\$29	\$sp
\$30	\$fp

## Local labels

Local labels can be used in assembly code. Ordinary labels can also be used, but depending on the compiler, ordinary labels may not be checked for duplicates, thereby creating the possible problem of the same label being defined in more than one place.

Local labels are represented by a number and colon at the beginning of the line.

```
0:
```

Local labels are referenced as follows, with a "b" or "f" to indicate whether the defined label appears backwards or forwards from the current position.

```
0:
... code goes here ...
    bne $0, $1, 0b

    beq $1, $2, 9f
... code goes here ...
9:
```

## Controlling the branch delay slot

With the default settings, the assembler will move an appropriate instruction or automatically generate a nop immediately after a branch instruction (the branch delay slot). To suppress this processing and explicitly insert an instruction in the branch delay slot, you must use the psuedo-instruction "`.set noreorder`" in the *asmstring*.

---

## ee-gdb

ee-gdb is the EE version of the popular gdb debugger. Those who are already comfortable with gdb can use ee-gdb instead of dsedb, with a few restrictions.

### Operating summary

First, start ee-gdb as follows.

```
% ee-gdb [-nw] name-of-program-to-debug
```

Attempting to launch ee-gdb itself through a link will result in the error "ide\_initialize\_paths failed". Either start it up with the full path name, or by using an alias. Specifying the -nw option starts ee-gdb in command-line interface mode.

Next, using the "target" command, connect to dsnetm.

```
(gdb) target deci2 hostname
```

Where "hostname" is the target DTL-T10000.

Next, using the "load" command, load the program in the target.

```
(gdb) load
```

After that, you can use all gdb commands and debug as needed.

After you are done running the program, you can run it again by re-executing the target, reset and load commands.

```
(gdb) target deci2 hostname
(gdb) reset
(gdb) load
```

### Precautions and restrictions

- The DTL-T10000's flash-ROM must be released at the same time as ee-gdb.
- stdin is not supported. We plan to support this in the future.
- VU1 program debugging is not supported. Also, some functions that appear to have worked initially cannot be used.
- A program that was compiled with optimization options can be debugged. However, because machine language instructions may execute in a different order when optimized, you may observe that the program's execution order may differ from the source code, or a given line may appear to be executed more than one time.

---

## ee-ld

ee-ld is a tool for combining objects together to create an executable file. Since ee-ld can be called automatically according to ee-gcc options, programmers usually don't need to use ee-ld directly.

## ee-nm

ee-nm is a utility that displays a list of symbolic information from an object file. It can be useful for investigating the cause of link errors.

### Example

To generate a list of symbolic information for main.elf, type the following

```
% ee-nm main.elf
(omitted)
0000000000102cf8 T sceGsResetGraph
0000000000102e18 T sceGsResetPath
0000000000103528 T sceGsSetDefAlphaEnv
0000000000103bf0 T sceGsSetDefClear
0000000000102e80 T sceGsSetDefDBuff
(omitted)
```

### Synopsis

```
ee-nm [options...] input_file...
```

The main *options...* that can be specified are shown below.

Table 27

Option	Function
-A	Output filename before each symbol name.
-o	Output filename once, at beginning (default).
-C	Output low-level symbol names (used with C++ programs).
-g	Output external symbol names only.
-n	Sort by address before outputting (default is by name).
-p	Do not sort, output in order of appearance.
-s	For symbols in the archive, also output the names of modules where the symbols are defined.
-u	Output symbols not defined in the modules (external symbols)
-l	Output each symbol name together with its line number in the source code.
-V	Display version information.
--help	Display list of options.

### Description

Symbolic information in the file specified by "*input\_file*" is output to stdout according to the specified *options...*

## ee-objcopy

ee-objcopy is a utility that copies and performs format conversion on object files. It can be used to delete symbolic information and extract overlay sections from elf files.

### Examples

To delete symbolic information from main.elf:

```
% ee-objcopy -S main.elf
```

To delete debugging information from main.elf:

```
% ee-objcopy -g main.elf
```

To extract section .overlay1/.overlay2 from main.elf (in this example, they are extracted to ov1.bin/ov2.bin):

```
% ee-objcopy -O binary -j .overlay1 main.elf ov1.bin
```

```
% ee-objcopy -O binary -j .overlay2 main.elf ov2.bin
```

To delete section .overlay1/.overlay2 from main.elf

```
% ee-objcopy -R .overlay1 -R .overlay2 main.elf
```

### Synopsis

```
ee-objcopy [options...] input_file [output_file]
```

The main *options...* that can be specified are shown below.

Table 28

Option	Function
-j <i>sectionname</i>	Extract only the section named " <i>sectionname</i> ".
-R <i>sectionname</i>	Remove the section named " <i>sectionname</i> ".
-S	Remove relocation information and symbol information.
-O <i>bfdname</i>	Specify the object format to output.
-g	Remove debugging information.
-v	Display detailed information while processing.
-V	Display version number.
--help	Display list of options.

Multiple sections can be specified with -R *sectionname*.

### Description

Reads in the object specified by "*input\_file*", converts its format based on the specified *options...*, and writes its output to "*output file*". If no options are specified, the *input\_file* is written as is to the *output\_file*. If the *output\_file* name is omitted, the *input\_file* will be converted and overwritten.

**Notes**

Deleting sections using the -R option may result in the display of the following warning message, but when extracting an overlay section, this warning can be ignored.

```
BFD: main.elf: warning: Empty loadable segment detected
```

Note that processing may not be correct if the elf file was created with a compiler other than ee-gcc.

## ee-objdump

ee-objdump is a utility for displaying information about object files. It has a number of uses, including finding addresses and sizes of sections in elf files.

### Example

To find the address of a section in main.elf,  

```
% ee-objdump -h main.elf
```

### Synopsis

```
ee-objdump options... objfiles...
```

*options...* are used to control the information that is output, so at least one must be specified. The main options that can be specified are shown below.

Table 29

Option	Function
-a	Output information on each object in the archive.
-d	Disassemble and output program code.
-f	Output header info for each file specified in " <i>objfiles...</i> ".
-h	Output header info for each section in the object.
-j <i>section</i>	Output section info for the section specified by " <i>section</i> ".
-r	Output relocation entry info.
-S	Output disassembly list with source-code lines.
-t	Output symbol table.
-x	Same as specifying all of "-a -f -h -r -t".
--version	Display version information.
--help	Display list of options.

### Description

Outputs the information specified with "*options...*" for each file specified by "*objfiles...*". In addition to object files (\*.o), archive files (\*.a) and elf files can also be specified.

## ee-protoize / ee-unprotoize

ee-protoize is a utility that adds argument type information to the function definitions and declarations in C source files and header files, then converts them to ANSI C prototypes.

ee-unprotoize performs the reverse function, deleting type information from function definitions and declarations, then converting them to the former notation.

### Synopsis

```
ee-protoize [options...] files...
ee-unprotoize [options...] files...
```

Here, "*files...*" represents C source files or header files to be converted. If these files include other source files or header files, they will also be converted.

The main *options...* that can be specified are shown below.

Table 30

Option	Function
-d <i>directory</i>	Specify the directory where files to be converted are located. The default is to use the current directory only.
-x <i>file</i>	Specify files to be excluded from conversion.
-g	Generate global prototype declarations.
(ee-protoize only)	Append function declarations with type information information to the beginning of each source file.
-l	Generate local prototype declarations.
(ee-protoize only)	Append function declarations with type information to blocks where those functions are called.
-i <i>string</i>	When generating old-style function declarations and definitions, use " <i>string</i> " as the indentation string. The default is to use five spaces: "    ".
(ee-unprotoize only)	
-c <i>gccopts</i>	Specify compilation options when special options are needed to compile the files indicated by " <i>files...</i> ". Multiple compile options should be enclosed in quotation marks to group them together for the shell. Because some compile options (-g / -O / -c / -S / -o) have no relevant output, they will be ignored.
-q	Inhibit warning messages.
-v	Display version number.

### Operation

Converts function declarations and definitions in the C source files and header files specified by "*files...*". ee-protoize generates function declarations and definitions with ANSI C-style type information for the arguments. ee-unprotoize performs the reverse operation, removing the type information for arguments from the function declarations and definitions. Note that declarations and definitions of functions that take a variable number of arguments will not be converted.

Before conversion, the contents of the pre-converted files are saved with filenames where the .c or .h suffix is replaced by .save. Be aware that if a .save file of the same name already exists, that file will not be overwritten, and the pre-converted file will not be saved.



**Notes**

Running ee-protoize/ee-unprotoize on files that were compiled with the "-aux-info" option will extract the function definitions and declarations from those files. Consequently, to use this function, you must be able to execute ee-gcc. Compiling a program with the "-aux-info" option, then running ee-protoize/ee-unprotoize may be more convenient than using the "-c gccopts" option.

---

## ee-ranlib

ee-ranlib is a utility that adds index information to an archive file to improve the speed of linking. Running ee-ranlib is the same as executing ee-ar with the "s" option.

### Synopsis

```
ee-ranlib archive
```

## ee-readelf

ee-readelf is a tool that outputs a summary of an executable file (elf file).

### Example

To generate a summary of the file "test.elf", type the following.

```
% ee-readelf -a test.elf
```

### Synopsis

```
ee-readelf options... files...
```

You must specify at least one option (in "*options...*") governing the type of output. The main options that can be specified are shown below.

**Table 31**

Option	Function
-a	Output all of the information below.
-h	Output elf file header info.
-l	Output program header info.
-S	Output header info for each section.
-e	Equivalent to -h -l -S.
-s	Output symbol table.
-r	Output if relocation info exists.
-d	Output info on dynamic sections if they exist.
-D	Use dynamic section info to output symbols.
-V	Output a version info section, if it exists.
-x <i>secno</i>	Dump the section with section number specified by <i>secno</i> .
-H	Display help.
-v	Display version information.

Input elf files should be specified in "*files...*"

---

## ee-size

ee-size is a utility that displays the size of each section in an object, as well as the object's total size.

### Example

To display the size of each section in sample.o, type the following:

```
% ee-size sample.o
```

### Synopsis

```
ee-size [options...] files...
```

Target files are specified in "*files...*" These can be object files, archive files, or executable files. The main options that can be specified are shown below.

Table 32

Option	Function
-x	Display size of each section in hex.
-o	Display size of each section in octal.
-A	Display in System V style.

## ee-strings

ee-strings is a utility for finding and displaying text strings embedded in object files or other files. ee-strings can be used to display copyright, version information, or error messages of a file, so that the contents can be roughly understood.

### Example

To display text strings embedded in the file `unknown.o`, type the following:

```
% ee-strings unknown.o
```

### Synopsis

```
ee-strings [options...] files...
```

filenames are specified by "*files...*". The main *options...* that can be specified are shown below.

Table 33

Option	Function
<i>-length</i>	Find strings of at least the specified <i>length</i> . Default is 4.
<i>-f</i>	Display filename before displaying strings.
<i>-t o</i>	Display offset from start of file before displaying strings.
<i>-t d</i>	<i>-t o</i> displays in octal, <i>-t d</i> in decimal, and <i>-t x</i> in hex.
<i>-t x</i>	
<i>-a</i>	Search entire file. If this option is not set, only sections with initial values will be searched.

## ee-strip

ee-strip is a utility for deleting variable names and other symbolic information from object files and archive files. ee-strip can also be used to convert object formats.

Since symbolic information is not needed for a program's execution, stripping symbols reduces the executable file size and also makes its internal structure harder to analyze. This should be done immediately prior to a program's release.

### Example

To strip symbols from the file `main.elf`, type the following

```
% ee-strip main.elf
```

### Synopsis

```
ee-strip [options...] input_file...
```

The main *options...* that can be specified are shown below.

**Table 34**

Option	Function
-s	Strip all symbolic information (default).
-g	Strip debugging information.
-S	
-N <i>symbolname</i>	Strip the symbol " <i>symbolname</i> ".
-R <i>sectionname</i>	Strip the section " <i>sectionname</i> ".
-O <i>bfdname</i>	Specify the object format to output.
-o <i>filename</i>	Specify a target file for output.
-p	Do not modify the file's timestamp.
-v	Display details during processing.
-V	Display version.
--help	Display list of options.

Multiple arguments can be specified for -N *symbolname*/-R *sectionname*. However, only one *input\_file* can be specified when using -o *filename*.

### Description

Strips symbolic information from the file indicated by "input\_file" that will not be needed in the final release, and overwrites the original file with a new one of the same name. The following two commands are equivalent:

```
% ee-strip main.elf
% ee-objcopy -g -S main.elf
```

## make

make is a tool that controls the process of building programs. make uses a procedure file, prepared in advance, that includes the commands needed to assemble, compile, and link all the source files and header files that will be used to build a program. Developing a program involves repeating many steps many times over until the program is complete. make is designed to simplify this process. In addition, make automatically recognizes when source files have been changed, and will rebuild only those files that are needed, shortening the rebuild process.

### Example

For starters, try executing make without specifying any arguments. This is sufficient for most sample programs that are provided with source code.

```
% make
```

Depending on the sample program, you can execute the following to build a program and see how it runs.

```
% make run
```

### Synopsis

```
% make [options...] [target]
```

The main *options...* that can be specified are shown below.

**Table 35**

Option	Function
-d	Display the command sequence during the build process, plus the reasons why make is executing those commands. Useful for debugging procedure files.
-e	Give variables from the shell environment priority over variables in the procedure file.
-f <i>filename</i>	Read " <i>filename</i> " as the procedure file. If omitted, looks for GNUmakefile / makefile / Makefile (in that order).
-i	Ignore error status returned by any commands during the build process. If omitted, make will terminate if a non-zero status is returned by any command.
-n	Check for modified files, and only display command sequence for rebuilding. Do not actually rebuild.
-p	Display preset variables and build rules,
-s	Suppress display of command sequence during build process.
-v	Display version information.

### Operating details

make reads in a procedure file, either the default file or the one specified using the -f option, and builds the *target*. The target can be explicitly specified on the command line, or if it is not specified, it can be described in the beginning of the procedure file. The details of the build process are determined by interpreting the procedure file.

## Procedure File Contents and Syntax

A procedure file is a text file having a specific format that is used to determine the procedure for building a program. Unless there is some special reason, the procedure file (named "makefile") should be saved in the same directory as the source files. Note that while the file is generally named "makefile", to avoid confusion, in this document we refer to it as a procedure file. The syntax of the procedure file is described below.

### Sample procedure file

Shown below is a procedure file for the sample program "basic3d(core)", which is part of the Tool-Chain.

```

SHELL      = /bin/sh                <== Variable definitions

TOP        = /usr/local/sce/ee
LIBDIR     = $(TOP)/lib
INCDIR     = $(TOP)/include

TARGET     = core
OBJS       = crt0.o \                <== "\" used to continue a line
             mathfunc.o \
             cube.o \
             torus1.o \
             flower.o \
             $(TARGET).o <== Refers to previously defined variable

LCFILE     = $(LIBDIR)/app.cmd
LIBS       = $(LIBDIR)/libgraph.a \
             $(LIBDIR)/libdma.a \
             $(LIBDIR)/libdev.a \
             $(LIBDIR)/libpkt.a \
             $(LIBDIR)/libpad.a \
             $(LIBDIR)/libvu0.a

PREFIX     = ee
AS         = $(PREFIX)-gcc
CC         = $(PREFIX)-gcc
LD         = $(PREFIX)-gcc
DVPASM     = $(PREFIX)-dvp-as
OBJDUMP    = $(PREFIX)-objdump
RUN        = dsedb -r run
RM         = /bin/rm -f

CFLAGS     = -O2 -Wall -Werror -Wa,-al -fno-common
CXXFLAGS   = -O2 -Wall -Werror -Wa,-al -fno-exceptions
           -fno-common
ASFLAGS    = -c -xassembler-with-cpp -Wa,-al
DVPASMFLAGS =
LDLFLAGS   = -Wl,-Map,$(TARGET).map -mno-crt0 -L$(LIBDIR) -lm
TMPFLAGS   =

.SUFFIXES: .c .s .cc .dsm          <== Special target for expressing
                                   suffix rules

all: $(TARGET).elf                <== Default target

$(TARGET).elf: $(OBJS) $(LIBS) <== Target, source files, etc.
               $(LD) -o $@ -T $(LCFILE) $(OBJS) $(LIBS) $(LDLFLAGS) <==
Command definition

```



```

crt0.o: $(LIBDIR)/crt0.s
    $(AS) $(ASFLAGS) $(TMPFLAGS) -o $@ $< > $*.lst

.s.o:                                     <== Suffix rule definition
    $(AS) $(ASFLAGS) $(TMPFLAGS) -I$(INCDIR) -o $@ $< > $*.lst

.dsm.o:
    $(DVPASM) $(DVPASMFLAGS) -I$(INCDIR) -o $@ $< > $*.lst

.c.o:
    $(CC) $(CFLAGS) $(TMPFLAGS) -I$(INCDIR) -c $< -o $*.o > $*.lst

.cc.o:
    $(CC) $(CXXFLAGS) $(TMPFLAGS) -I$(INCDIR) -c $< -o $*.o >
$*.lst

run: $(TARGET).elf
    $(RUN) $(TARGET).elf

clean:                                     <== Pseudo-target
    $(RM) *.o *.map *.lst core *.dis *.elf

```

## Line continuation

Procedure files are interpreted line-by-line. To continue a line onto the next line, a "\" should be used at the end of the line. This allows line breaks to be inserted for readability while treating the separate lines as a single line.

## Comments

The above example did not have comments, but comments can be inserted in procedure files by preceding them with a "#". make will ignore text from the '#' to the end of the line.

## Variables

The first half of a procedure file consists of definitions of variables that will be used in the procedure file. A variable definition begins on a new line, and is of the form "*variablename* = *value*". References to variable names are of the form "\$(*variablename*)". The following sample code defines variables for directories containing the header files and library files, INCDIR and LIBDIR. Since these variables refer to the variable "TOP", if ToolChain were installed in a different directory, only the variable "TOP" would need to be updated.

```

TOP      = /usr/local/sce/ee
LIBDIR   = $(TOP)/lib
INCDIR   = $(TOP)/include

```

While not shown in this example, environment variables are referenced in the same way.

## Targets and build rules

The object to be created by make is referred to as the "target". Build rules are the definitions of source files that go into the target and related materials, along with the definitions of commands that must be executed to build the target.

Target definitions begin at the start of a new line, and are of the form "*targetname*: *sourcefiles*..." (when names of files for targets that already exist are included, it is known as a dependency-relationship definition). Command definitions follow target definitions, and are indented with tabs (lines must begin with a tab, not a space). Command definitions represent the commands for building that target.

The following is a build rule

```
$(TARGET).elf: $(OBJS) $(LIBS)
    $(LD) -o $@ -T $(LDFILE) $(OBJS) $(LIBS) $(LDFLAGS)
```

... to expand variables, do the following ...

```
core.elf: crt0.o mathfunc.o (omitted) /usr/local/sce/ee/lib/
    libvu0.a
    ee-gcc -o core.elf -T /usr/local/sce/ee/lib/app.cmd crt0.o
(omitted) -lm
```

This indicates that the commands `ee-gcc -o core.elf...` need to be executed to build `core.elf` from `crt0.o/mathfunc.o/.../libvu0.a`. In this example, the build rule only uses one command, so it can be written on one line, but if multiple commands were needed to build the target, command definitions would continue to be added as needed.

There is a special variable, "\$@", used to express the complete target name. Related variables are "\$\*" which represents the target name without a suffix, and the variable "\$<" which represents the source files.

make will compare the target's timestamp with that of each source file, to determine whether there are source files newer than the target, that is, to find out if any source files have been updated. make will only perform a build if it determines that there are changes not yet reflected in the target.

Before building the target, make will check to see whether each of the source files has been defined in the build rules for that target. If so, it will first build the source files accordingly. For example, since `crt0.o`, which is specified as a source file for `core.elf`, is defined in the build rules, make will first compare the timestamps between `/usr/local/sce/ee/lib/crt0.s` and `crt0.o`, and if `crt0.s` is newer, it will rebuild `crt0.o` before rebuilding `core.elf`.

```
crt0.o: $(LIBDIR)/crt0.s
    $(AS) $(ASFLAGS) $(TMPFLAGS) -o $@ $< > $*.lst
```

Other source files are processed in the same way. This recursive process ensures that the build process has no wasted effort, even with many files involved in complex procedures.

### Suffix rules and pattern rules

Suffix rules are build rules that ensure consistent handling of files with a given suffix. The following is an example of a build rule for `.o` files from `.s` files:

```
.s.o:
    $(AS) $(ASFLAGS) $(TMPFLAGS) -I$(INCDIR) -o $@ $< > $*.lst
```

A pattern rule is a build rule using the "%" wildcard. The following pattern rule is functionally equivalent to the above suffix rule.

```
%.o: %.s
    $(AS) $(ASFLAGS) $(TMPFLAGS) -I$(INCDIR) -o %.o %.s > %.lst
```

### Pseudo-targets

Normally, the target for make is specified as a file to be created, but it is also possible to define targets that are not files, that is, where the build rules do not create a file. For example, the following build rule will delete all the intermediate files generated in the process of building a sample program.

```
clean:
    $(RM) *.o *.map *.lst core *.dis *.elf
```

Because this build rule does not specify any source files, it can be executed by simply typing "make clean" at the command line.

### Special targets

There are a number of special target names that have been reserved to provide special instructions for make.

One is the ".SUFFIXES" used in the above example. This is taken as a declaration of suffix rules for the specified suffixes. Using the ".SUFFIXES" notation without specifying any suffixes will disable the default suffix rules built into make.

Another special target is ".IGNORE" which has the same behavior as the -i option.

