AN3350
Application note

eTPU compiler tools

Introduction

This application note explains how to use eTPU build tools to build programs. The eTPU build tools translate source code into object code and then organize that object code to create a program that is ready to execute. The eTPU build tools run on the **host** system to generate software that runs on **target** systems (such as SPC563Mxx and SPC564Axx devices) which feature the eTPU peripheral. The host and target can be the same system, but usually the two are separate.

Contents

List of tables

List of figures

1 Overview

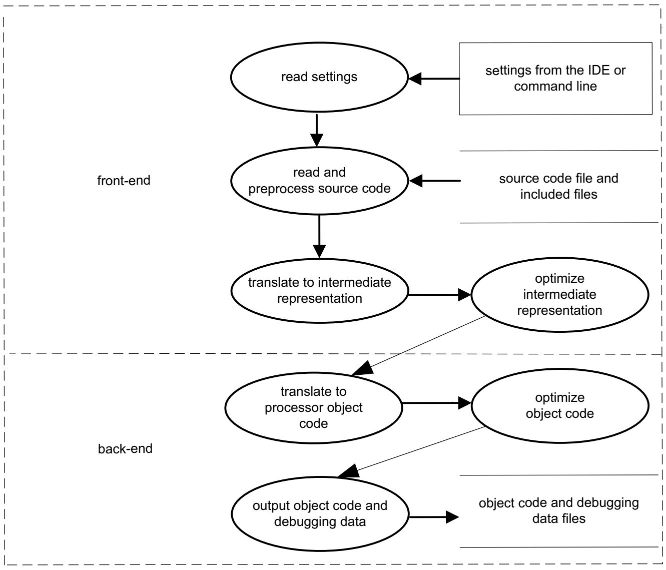
This chapter explains the processes that eTPU build tools use to create software:

- *Compiler architecture*
- *Linker architecture*

1.1 Compiler architecture

From a programmer's point of view, the compiler translates source code into object code. Internally, however, the compiler organizes its work between its front-end and back-end, each end taking several steps. *Figure 1* shows the steps the compiler takes.

Figure 1. Compiler steps



Front-end steps:

- Read settings: retrieves your settings from the host's integrated development environment (IDE) or the command line to configure how to perform subsequent steps
- Read and preprocess source code: reads your program's source code files and applies preprocessor directives
- Translate to intermediate representation: translates your program's preprocessed source code into a platform-independent intermediate representation
- Optimize intermediate representation: rearranges the intermediate representation to reduce your program's size, improve its performance, or both

Back-end steps:

- Translate to processor object code: converts the optimized intermediate representation into native object code, containing data and instructions, for the target processor
- Optimize object code: rearranges the native object code to reduce its size, improve performance, or both
- Output object code and diagnostic data: writes output files on the host system, ready for the linker and diagnostic tools such as a debugger or profiler

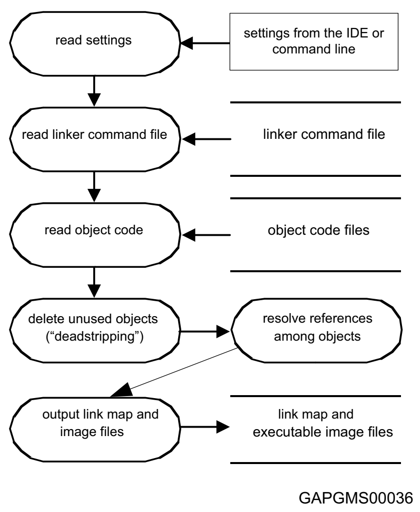
1.2 Linker architecture

A linker combines and arranges data and instructions from one or more object code files into a single file, or *image*. This image is ready to execute on the target platform. The linker uses settings from the host's integrated development environment (IDE) or command line to determine how to generate the image file.

The linker also optionally reads a linker command file. A linker command file allows you to specify precise details of how data and instructions should be arranged in the image file.

Figure 2 shows the steps the linker takes to build an executable image.

Figure 2. Linker steps



- Read settings: retrieves your settings from the IDE or the command line to determine how to perform subsequent steps
- Read linker command file: retrieves commands to determine how to arrange object code in the final image
- Read object code: retrieves data and executable objects that are the result of compilation or assembly
- Delete unused objects (“deadstripping”): deletes objects that are not referred to by the rest of the program
- Resolve references among objects: arranges objects to compose the image then computes the addresses of the objects
- Output link map and image files: writes files on the host system, ready to load onto the target system

2 Using build tools

The eTPU build tools may be invoked from the command-line. These command-line tools operate almost identically to their counterparts in an Integrated Development Environment (IDE). Command-line compilers and assemblers translate source code files into object code files. Command-line linkers then combine one or more object code files to produce an executable image file, ready to load and execute on the target platform. Each command-line tool has options that you configure when you invoke the tool.

- *Configuring command-line tools*
- *Invoking command-line tools*
- *Getting help*
- *File name extensions*

2.1 Configuring command-line tools

- *Setting environment variables*
- *Setting the PATH environment variable*

2.1.1 Setting environment variables

Use environment variables on the host system to specify to the command line tools where to find files for compiling and linking. *Table 1* describes these environment variables.

Table 1. Environment variables for command-line tools

This environment variable...	specifies this information
MWCIncludes	Directories on the host system for system header files for the compiler.
MWLibraries	Directories on the host system for system libraries for the linker.

A system header file is a header file that is enclosed with the “<“ and “>” characters in include directives. For example

```
#include <stdlib.h> /* stdlib.h system header. */
```

Typically, you define the `MWCIncludes` and `MWLibraries` environment variables to refer to the header files and libraries in the subdirectories of your software.

To specify more than one directory for the `MWCIncludes` and `MWLibraries` variables, use the conventional separator for your host operating system command-line shell.

Example 1 Setting environment variables in Microsoft® Windows® operating systems

```
rem Use ; to separate directory paths
set CWFolder=C:\Program Files\STMicroelectronics\eTPU Build Tools
v1.2
set MWCIncludes="%CWFolder%\include"
set set MWLibraries="%CWFolder%\lib"
```

2.1.2 Setting the PATH environment variable

The `PATH` variable should include the paths for your eTPU build tools, shown in *Example 2*. *Toolset* represents the name of the folder that contains the command line tools for your build target.

Example 2 Example of setting PATH

```
set CWFolder=C:\Program Files\STMicroelectronics\eTPU Build Tools
v1.2
```

```
set PATH=%PATH%;%CWFolder%\Bin
```

2.2 Invoking command-line tools

To compile, assemble, link, or perform some other programming task with the command-line tools, you type a command at a command line's prompt. This command specifies the tool you want to run, what options to use while the tool runs, and what files the tool should operate on.

The form of a command to run a command-line tool is

tool options files

where *`tool`* is the name of the command-line tool to invoke, *`options`* is a list of zero or more options that specify to the tool what operation it should perform and how it should be performed, and *`files`* is a list of files zero or more files that the tool should operate on.

Which options and files you should specify depend on what operation you want the tool to perform.

The tool then performs the operation on the files you specify. If the tool is successful it simply finishes its operation and a new prompt appears at the command line. If the tool encounters problems it reports these problems as text messages on the command-line before a new prompt appears.

Scripts that automate the process to build a piece of software contain commands to invoke command-line tools. For example, the `make` tool, a common software development tool, uses scripts to manage dependencies among source code files and invoke command-line compilers, assemblers and linkers.

2.3 Getting help

To show short descriptions of a tool's options, type this command at the command line:

```
tool -help
```

where *tool* is the name of the eTPU build tools.

To show only a few lines of help information at a time, pipe the tool's output to a pager program. For example,

```
tool -help | more
```

will use the `more` pager program to display the help information.

Enter the following command in a Command Prompt window to see a list of specifications that describe how options are formatted:

`tool -help usage`

where *tool* is the name of the eTPU build tools.

2.3.1 Parameter formats

Parameters in an option are formatted as follows:

- A parameter included in brackets "`[]`" is optional.
- Use of the ellipsis "`...`" character indicates that the previous type of parameter may be repeated as a list.

2.3.2 Option formats

Options are formatted as follows:

- For most options, the option and the parameters are separated by a space as in "`-xxx param`".
- When the option's name is "`-xxx+`", however, the parameter must directly follow the option, without the "`+`" character (as in "`-xxx45`") and with no space separator.
- An option given as "`-[no]xxx`" may be issued as "`-xxx`" or "`-noxxx`".
- The use of "`-noxxx`" reverses the meaning of the option.
- When an option is specified as "`-xxx | yy[y] | zzz`", then either "`-xxx`", "`-yy`", "`-yyy`", or "`-zzz`" matches the option.
- The symbols "`,`" and "`=`" separate options and parameters unconditionally; to include one of these symbols in a parameter or filename, escape it (e.g., as "`\,`" in `mwcc file.c\,v`).

2.3.3 Common terms

These common terms appear in many option descriptions:

- A "cased" option is considered case-sensitive. By default, no options are case-sensitive.
- "Compatibility" indicates that the option is borrowed from another vendor's tool and its behavior may only approximate its counterpart.
- A "global" option has an effect over the entire command line and is parsed before any other options. When several global options are specified, they are interpreted in order.
- A "deprecated" option will be eliminated in the future and should no longer be used. An alternative form is supplied.
- An "ignored" option is accepted by the tool but has no effect.
- A "meaningless" option is accepted by the tool but probably has no meaning for the target operating system.
- An "obsolete" option indicates a deprecated option that is no longer available.
- A "substituted" option has the same effect as another option. This points out a preferred form and prevents confusion when similar options appear in the help.
- Use of "default" in the help text indicates that the given value or variation of an option is used unless otherwise overridden.

This tool calls the linker (unless a compiler option such as `-c` prevents it) and understands linker options – use "`-help tool=other`" to see them. Options marked "passed to linker" are used by the compiler and the linker; options marked "for linker" are used only by the linker. When using the compiler and linker separately, you must pass the common options to both.

2.4 File name extensions

Files specified on the command line are identified by contents and file extension.

The command-line version of the eTPU build tools C/C++ compiler accepts non-standard file extensions as source code but also emits a warning message. By default, the compiler assumes that a file with any extensions besides `.c`, `.h`, `.pch` is C++ source code. The linker ignores all files that it can not identify as object code, libraries, or command files.

Linker command files must end in `.lcf`. They may be simply added to the link line, for example (*Example 3*).

Example 3 Example of using linker command files

```
mwldtarget file.o lib.a commandfile.lcf
```

For more information on linker command files, refer to the *Targeting* manual for your platform.

3 Command-line options for standard C conformance

3.1 -ansi

Controls the ISO/IEC 9899-1990 (“C90”) conformance options, overriding the given settings.

Syntax

`-ansi` *`keyword`*

The arguments for `keyword` are:

`off`

Turns ISO conformance off. Same as

`-stdkeywords off -enum min -strict off.`

`on | relaxed`

Turns ISO conformance on in relaxed mode. Same as

`-stdkeywords on -enum min -strict on`

`strict`

Turns ISO conformance on in strict mode. Same as

`-stdkeywords on -enum int -strict on`

3.2 -stdkeywords

Controls the use of ISO/IEC 9899-1990 (“C90”) keywords.

Syntax

`-stdkeywords on | off`

Remarks

Default setting is `off`.

3.3 -strict

Controls the use of non-standard ISO/IEC 9899-1990 (“C90”) language features.

Syntax

`-strict on | off`

Remarks

If this option is `on`, the compiler generates an error message if it encounters some eTPU build tools extensions to the C language defined by the ISO/IEC 9899-1990 (“C90”) standard:

- C++-style comments
- Unnamed arguments in function definitions
- Non-standard keywords

The default setting is `off`.

4 Command-line options for language translation

4.1 -char

Controls the default sign of the `char` data type.

Syntax

`-char` *`keyword`*

The arguments for *keyword* are:

`signed`

`char data items are signed.`

`unsigned`

`char data items are unsigned.`

Remarks

The default is `signed`.

4.2 -defaults

Controls whether the compiler uses additional environment variables to provide default settings.

Syntax

`-defaults`

`-nodefaults`

Remarks

This option is global. To tell the command-line compiler to use the same set of default settings as in an integrated development environment (IDE), use `-defaults`. For example, in the IDE, all access paths and libraries are explicit. `defaults` is the default setting.

Use `-nodefaults` to disable the use of additional environment variables.

4.3 -encoding

Specifies the default source encoding used by the compiler.

Syntax

`-enc[oding]` *`keyword`*

The options for *keyword* are:

`ascii`

American Standard Code for Information Interchange (ASCII) format. This is the default.

`autodetect | multibyte | mb`

Scan file for multibyet encoding.

`system`

Uses local system format.

`UTF[8 | -8]`

Unicode Transformation Format (UTF).

`SJIS | Shift-JIS | ShiftJIS`

Shift Japanese Industrial Standard (Shift-JIS) format.f

`EUC[JP | -JP]`

Japanese Extended UNIX Code (EUCJP) format.

`ISO[2022JP | -2022-JP]`

International Organization of Standards (ISO) Japanese format.

Remarks

The compiler automatically detects UTF-8 (Unicode Transformation Format) header or UCS-2/UCS-4 (Uniform Communications Standard) encodings regardless of setting. The default setting is `ascii`.

4.4 -flag

Specifies compiler `#pragma` as either `on` or `off`.

Syntax

`-fl[ag] [no-]pragma`

Remarks

For example, this option setting

`-flag require_prototypes`

is equivalent to

`#pragma require_prototypes on`

This option setting

`-flag no-require_prototypes`

is the same as

`#pragma require_prototypes off`

4.5 -gccext

Enables GCC (Gnu Compiler Collection) C language extensions.

Syntax

```
-gcc[ext] on | off
```

Remarks

See *Section 12.3: GCC extensions* for a list of language extensions that the compiler recognizes when this option is `on`.

The default setting is `off`.

4.6 -gcc_extensions

Equivalent to the `-gccext` option.

Syntax

```
-gcc[_extensions] on | off
```

4.7 -M

Scans source files for dependencies and emit a Makefile, without generating object code.

Syntax

```
-M
```

Remarks

This command is global and case-sensitive.

4.8 -make

Scans source files for dependencies and emit a Makefile, without generating object code.

Syntax

```
-make
```

Remarks

This command is global.

4.9 -mapcr

Swaps the values of the `\n` and `\r` escape characters.

Syntax

`-mapcr`

`-nomapcr`

Remarks

The `-mapcr` option tells the compiler to treat the `'\n'` character as ASCII 13 and the `'\r'` character as ASCII 10. The `-nomapcr` option tells the compiler to treat these characters as ASCII 10 and 13, respectively.

4.10 -MM

Scans source files for dependencies and emit a Makefile, without generating object code or listing system `#include` files.

Syntax

`-MM`

Remarks

This command is global and case-sensitive.

4.11 -MD

Scans source files for dependencies and emit a Makefile, generate object code, and write a dependency map.

Syntax

`-MD`

Remarks

This command is global and case-sensitive.

4.12 -MMD

Scans source files for dependencies and emit a Makefile, generate object code, write a dependency map, without listing system `#include` files.

Syntax

`-MMD`

Remarks

This command is global and case-sensitive.

4.13 -msext

Allows Microsoft® Visual C++ extensions.

Syntax

```
-msext on | off
```

Remarks

Turn on this option to allow Microsoft Visual C++ extensions:

- Redefinition of macros
- Allows `XXX::yyy` syntax when declaring method `yyy` of class `XXX`
- Allows extra commas
- Ignores casts to the same type
- Treats function types with equivalent parameter lists but different return types as equal
- Allows pointer-to-integer conversions, and various syntactical differences

4.14 -once

Prevents header files from being processed more than once.

Syntax

```
-once
```

Remarks

You can also add `#pragma once on` in a prefix file.

4.15 -pragma

Defines a pragma for the compiler.

Syntax

```
-pragma "name [setting]"
```

The arguments are:

`name`

Name of the pragma.

`setting`

Arguments to give to the pragma

Remarks

For example, this command-line option

```
-pragma "c99 on"
```

is equivalent to inserting this directive in source code

```
#pragma c99 on
```

4.16 -relax_pointers

Relaxes the pointer type-checking rules in C.

Syntax

```
-relax_pointers
```

Remarks

This option is equivalent to

```
#pragma mpwc_relax on
```

4.17 -requireprotos

Controls whether or not the compiler should expect function prototypes.

Syntax

```
-r[equireprotos]
```

4.18 -search

Globally searches across paths for source files, object code, and libraries specified in the command line.

Syntax

```
-search
```

4.19 -trigraphs

Controls the use of trigraph sequences specified by the ISO/IEC standards for C and C++.

Syntax

```
-trigraphs on | off
```

Remarks

Default setting is `off`.

5 Command-line options for diagnostic messages

5.1 -disassemble

Tells the command-line tool to disassemble files and send result to `stdout`.

Syntax

`-dis[assemble]`

Remarks

This option is global.

5.2 -help

Lists descriptions of the tool's command-line options.

Syntax

`-help [`*`keyword`*` [,...]]`

The options for *keyword* are:

`all`

Show all standard options

`group=`*`keyword`*

Show help for groups whose names contain *keyword* (case-sensitive).

`[no]compatible`

Use `compatible` to show options compatible with this compiler. Use `nocompatible` to show options that do not work with this compiler.

`[no]deprecated`

Shows deprecated options

`[no]ignored`

Shows ignored options

`[no]meaningless`

Shows options meaningless for this target

`[no]normal`

Shows only standard options

`[no]obsolete`

Shows obsolete options

`[no]spaces`

Inserts blank lines between options in printout.

`opt[ion]=`*`name`*

Shows help for a given option; for *name*, maximum length 63 chars

`search=`*`keyword`*

Shows help for an option whose name or help contains *keyword* (case-sensitive), maximum length 63 chars

`tool=keyword[ all | this | other | skipped | both ]`

Categorizes groups of options by tool; default.

- `all`–show all options available in this tool
- `this`–show options executed by this tool; default
- `other | skipped`–show options passed to another tool
- `both`–show options used in all tools

`usage`

Displays usage information.

5.3 -maxerrors

Specifies the maximum number of errors messages to show.

Syntax

`-maxerrors` *`max`*

`max`

Use `max` to specify the number of error messages. Common values are:

- `0` (zero) – disable maximum count, show all error messages.
- `100` – Default setting.

5.4 -maxwarnings

Specifies the maximum number of warning messages to show.

Syntax

`-maxerrors` *`max`*

`max`

Specifies the number of warning messages. Common values are:

- `0` (zero) – Disable maximum count (default).
- *`n`* – Maximum number of warnings to show.

5.5 -msgstyle

Controls the style used to show error and warning messages.

Syntax

`-msgstyle` *`keyword`*

The options for *keyword* are:

`gcc`

Uses the message style that the Gnu Compiler Collection tools use.

`ide`

Uses message style.

`mpw`

Uses Macintosh Programmer's Workshop (MPW®) message style.

`parseable`

Uses context-free machine parseable message style.

`std`

Uses standard message style. This is the default.

`enterpriseIDE`

Uses Enterprise-IDE message style.

5.6 -nofail

Continues processing after getting error messages in earlier files.

Syntax

```
-nofail
```

5.7 -progress

Shows progress and version information.

Syntax

```
-progress
```

5.8 -S

Disassembles all files and send output to a file. This command is global and case-sensitive.

Syntax

```
-S
```

5.9 -stderr

Uses the standard error stream to report error and warning messages.

Syntax

```
-stderr
```

```
-nostderr
```

Remarks

The `-stderr` option specifies to the compiler, and other tools that it invokes, that error and warning messages should be sent to the standard error stream.

The `-nostderr` option specifies that error and warning messages should be sent to the standard output stream.

5.10 -verbose

Tells the compiler to provide extra, cumulative information in messages.

Syntax

```
-v[erbose]
```

Remarks

This option also gives progress and version information.

5.11 -version

Displays version, configuration, and build data.

Syntax

```
-v[ersion]
```

5.12 -timing

Shows the amount of time that the tool used to perform an action.

Syntax

```
-timing
```

5.13 -warnings

Specifies which warning messages the command-line tool issues. This command is global.

Syntax

`-w[arning]` *`keyword`* `[,...]`

The options for `keyword` are:

`off`

Turns off all warning messages. Passed to all tools. Equivalent to

`#pragma warning off`

`on`

Turns on most warning messages. Passed to all tools. Refer *Table 2* for a list of warning messages turned on by the `-w[arning] on` command.

Equivalent to #pragma warning on

`most`

Turns on most warnings.

`all`

Turns on almost all warnings and require prototypes.

`full`

Turns on all warning messages and require prototypes. This option is likely to generate spurious warnings.

Note: `-warnings full` should be used before using any other options that affect warnings. For example, use
`-warnings full -warnings noanyptrintconv` instead of
`-warnings noanyptrintconv -warnings full.`

`[no]cmdline`

Passed to all tools.

`[no]err[or] | [no]iserr[or]`

Treats warnings as errors. Passed to all tools. Equivalent to

`#pragma warning_errors`

`[no]pragmas | [no]illpragmas`

Issues warning messages on invalid pragmas. Enabled when `most` is used. Equivalent to

`#pragma warn_illpragma`

`[no]empty[decl]`

Issues warning messages on empty declarations. Enabled when most is used. Equivalent to

`#pragma warn_emptydecl`

`[no]possible | [no]unwanted`

Issues warning messages on possible unwanted effects. Enabled when `most` is used. Equivalent to

`#pragma warn_possunwanted`

`[no]unusedarg`

Issues warning messages on unused arguments. Enabled when `most` is used. Equivalent to

`#pragma warn_unusedarg`

`[no]unusedvar`

Issues warning messages on unused variables. Enabled when `most` is used. Equivalent to

`#pragma warn_unusedvar`

`[no]unused`

Same as

`-w [no]unusedarg,[no]unusedvar`

Enabled when `most` is used.

`[no]extracomma | [no]comma`

Issues warning messages on extra commas in enumerations. The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 (“C99”) standard. Enabled when `most` is used. Equivalent to

`#pragma warn_extracomma`

`[no]extended_errorcheck`

Extended error checking. Enabled when `most` is used. Equivalent to

`#pragma extended_errorcheck`

`[no]hidevirtual | [no]hidden[virtual]`

Issues warning messages on hidden virtual functions. Enabled when `most` is used. Equivalent to

`#pragma warn_hidevirtual`

`[no]implicit[conv]`

Issues warning messages on implicit arithmetic conversions. Enabled when `all` is used. Implies

`-warn impl_float2int,impl_signedunsigned`

`[no]impl_int2float`

Issues warning messages on implicit integral to floating conversions. Enabled when `all` is used. Equivalent to

`#pragma warn_impl_i2f_conv`

`[no]impl_float2int`

Issues warning messages on implicit floating to integral conversions. Enabled when `all` is used. Equivalent to

`#pragma warn_impl_f2i_conv`

`[no]impl_signedunsigned`

Issues warning messages on implicit signed/unsigned conversions. Enabled when `all` is used.

`[no]notinlined`

Issues warning messages for functions declared with the `inline` qualifier that are not inlined. Enabled when `full` is used. Equivalent to

`#pragma warn_notinlined`

`[no]largeargs`

Issues warning messages when passing large arguments to unprototyped functions. Enabled when `most` is used. Equivalent to

`#pragma warn_largeargs`

`[no]structclass`

Issues warning messages on inconsistent use of `class` and `struct`. Enabled when `most` is used. Equivalent to

`#pragma warn_structclass`

`[no]padding`

Issue warning messages when padding is added between `struct` members. Enabled when `full` is used. Equivalent to

`#pragma warn_padding`

`[no]notused`

Issues warning messages when the result of non-void-returning functions are not used. Enabled when `full` is used. Equivalent to

`#pragma warn_resultnotused`

`[no]missingreturn`

Issues warning messages when a return without a value in non-void-returning function occurs. Enabled when `most` is used. Equivalent to

`#pragma warn_missingreturn`

`[no]unusedexpr`

Issues warning messages when encountering the use of expressions as statements without side effects. Equivalent to

`#pragma warn_no_side_effect`

`[no]ptrintconv`

Issues warning messages when lossy conversions occur from pointers to integers. Enabled when `full` is used.

`[no]anyptrintconv`

Issues warning messages on any conversion of pointers to integers. Enabled when `full` is used. Equivalent to

`#pragma warn_ptr_int_conv`

`[no]undef[macro]`

Issues warning messages on the use of undefined macros in `#if` and `#elif` conditionals. Enabled when `full` is used. Equivalent to

`#pragma warn_undefmacro`

`[no]filecaps`

Issues warning messages when `#include ""` directives use incorrect capitalization. Enabled when `most` is used. Equivalent to

`#pragma warn_filenamecaps`

`[no]sysfilecaps`

Issue warning messages when `#include <>` statements use incorrect capitalization. Enabled when `most` is used. Equivalent to

`#pragma warn_filenamecaps_system`

`[no]tokenpasting`

Issue warning messages when token is not formed by the `##` preprocessor operator. Enabled when `most` is used. Equivalent to

`#pragma warn_illtokenpasting`

`[no]relax_i2i_conv`

Relax implicit arithmetic conversion warnings on certain implicit conversions. Equivalent to

`#pragma relax_i2i_conv`

`display | dump`

Display list of active warnings.

Remarks

Table 2 lists the equivalent command option of the warning messages turned on by the `-w[arning] on` command.

Table 2. Warnings turned on by the -w[arning] on command

most	all (includes most)	full (includes all and most)
[no]pragmas | [no]illpragmas	*[no]possible | [no]unwanted*	*[no]notinlined*
[no]empty[decl]	*[no]implicit[conv]*	*[no]notused*
[no]unusedarg	*[no]impl_int2float*	*[no]ptrintconv*
[no]unusedvar	*[no]impl_float2int*	*[no]anyptrintconv*
[no]unused	*[no]impl_signedunsigned*	*[no]undef[macro]*
[no]extracomma | [no]comma		
[no]extended_errorcheck		
[no]hidevirtual | [no]hidden[virtual]		
[no]largeargs		
[no]structclass		
[no]padding		
[no]missingreturn		
[no]unusedexpr		
[no]filecaps		
[no]sysfilecaps		
[no]tokenpasting		

5.14 -wraplines

Controls the word wrapping of messages.

Syntax

```
-wraplines
```

```
-nowraplines
```

6 Command-line options for preprocessing

6.1 -convertpaths

Instructs the compiler to interpret #include file paths specified for a foreign operating system. This command is global.

Syntax

-[no]convertpaths

Remarks

The compiler can interpret file paths from several different operating systems. Each operating system uses unique characters as path separators. These separators include:

- Mac OS® – colon “:” (:sys:stat.h)
- UNIX – forward slash “/” (sys/stat.h)
- Windows® operating systems – backward slash “\” (sys\stat.h)

When convertpaths is enabled, the compiler can correctly interpret and use paths like <sys/stat.h> or <:sys:stat.h>. However, when enabled, (/) and (:) separate directories and cannot be used in filenames.

Note: This is not a problem on Windows systems since these characters are already disallowed in file names. It is safe to leave this option on.

When noconvertpaths is enabled, the compiler can only interpret paths that use the Windows form, like <\sys\stat.h>.

6.2 -cwd

Controls where a search begins for #include files.

Syntax

-cwd *keyword*

The options for *keyword* are:

explicit

No implicit directory. Search -I or -ir paths.

include

Begins searching in directory of referencing file.

proj

Begins searching in current working directory (default).

source

Begins searching in directory that contains the source file.

Remarks

The path represented by *keyword* is searched before searching access paths defined for the build target.

6.3 -D+

Same as the `-define` option.

Syntax

`-D+`*`name`*

The parameters are:

`name`

The symbol name to define. Symbol is set to 1.

6.4 -define

Defines a preprocessor symbol.

Syntax

`-d[efine]` *`name`*`[=`*`value`*`]`

The parameters are:

`name`

The symbol name to define.

`value`

The value to assign to symbol name. If no value is specified, set symbol value equal to 1.

6.5 -E

Tells the command-line tool to preprocess source files.

Syntax

`-E`

Remarks

This option is global and case sensitive.

6.6 -EP

Tells the command-line tool to preprocess source files that are stripped of `#line` directives.

Syntax

`-EP`

Remarks

This option is global and case sensitive.

6.7 -gccincludes

Controls the compilers use of GCC `#include` semantics.

Syntax

`-gccinc[ludes]`

Remarks

Use `-gccincludes` to control the compiler understanding of Gnu Compiler Collection (GCC) semantics. When enabled, the semantics include:

- Adds `-I-` paths to the systems list if `-I-` is not already specified
- Search referencing file's directory first for `#include` files (same as `-cwd include`) The compiler and IDE only search access paths, and do not take the currently `#include` file into account.

This command is global.

6.8 -I-

Changes the build target's search order of access paths to start with the system paths list.

Syntax

`-I-`

`-i-`

Remarks

The compiler can search `#include` files in several different ways. Use `-I-` to set the search order as follows:

- For include statements of the form `#include "xyz"`, the compiler first searches user paths, then the system paths
- For include statements of the form `#include <xyz>`, the compiler searches only system paths

This command is global.

6.9 -I+

Appends a non-recursive access path to the current `#include` list.

Syntax

`-I+`*`path`*

`-i` *`path`*

The parameters are:

`path`

The non-recursive access path to append.

Remarks

This command is global and case-sensitive.

6.10 -include

Defines the name of the text file or precompiled header file to add to every source file processed.

Syntax

`-include` *`file`*

`file`

Name of text file or precompiled header file to prefix to all source files.

Remarks

With the command line tool, you can add multiple prefix files all of which are included in a meta-prefix file.

6.11 -ir

Appends a recursive access path to the current `#include` list. This command is global.

Syntax

`-ir` *`path`*

`path`

The recursive access path to append.

6.12 -P

Preprocesses the source files without generating object code, and send output to file.

Syntax

`-P`

Remarks

This option is global and case-sensitive.

6.13 -precompile

Precompiles a header file from selected source files.

Syntax

`-precompile *file* | *dir* | ""`

`file`

If specified, the precompiled header name.

`dir`

If specified, the directory to store the header file.

`""`

If `""` is specified, write header file to location specified in source code. If neither argument is specified, the header file name is derived from the source file name.

Remarks

The driver determines whether to precompile a file based on its extension. The option

`-precompile *filesource*`

is equivalent to

`-c -o *filesource*`

6.14 -preprocess

Preprocesses the source files. This command is global.

Syntax

`-preprocess`

6.15 -ppopt

Specifies options affecting the preprocessed output.

Syntax

`-ppopt *keyword* [,...]`

The arguments for *keyword* are:

`[no]break`

Emits file and line breaks. This is the default.

`[no]line`

Controls whether #line directives are emitted or just comments. The default is `line`.

`[no]full[path]`

Controls whether full paths are emitted or just the base filename. The default is `fullpath`.

`[no]pragma`

Controls whether #pragma directives are kept or stripped. The default is `pragma`.

`[no]comment`

Controls whether comments are kept or stripped.

`[no]space`

Controls whether whitespace is kept or stripped. The default is `space`.

Remarks

The default settings is `break`.

6.16 -prefix

Adds contents of a text file or precompiled header as a prefix to all source files.

Syntax

```
-prefix file
```

6.17 -noprecompile

Do not precompile any source files based upon the filename extension.

Syntax

```
-noprecompile
```

6.18 -nosyspath

Performs a search of both the user and system paths, treating `#include` statements of the form `#include <xyz>` the same as the form `#include "xyz"`.

Syntax

```
-nosyspath
```

Remarks

This command is global.

6.19 -stdinc

Uses standard system include paths as specified by the environment variable `%MWCIncludes%`.

Syntax

`-stdinc`

`-nostdinc`

Remarks

Add this option after all system `-I` paths.

6.20 -U+

Same as the -undefine option.

Syntax

`-U+`*`name`*

6.21 -undefine

Undefines the specified symbol name.

Syntax

`-u[ndefine]` *`name`*

`-U+`*`name`*

`name`

The symbol name to undefine.

Remarks

This option is case-sensitive.

7 Command-line options for object code

7.1 -c

Instructs the compiler to compile but not invoke the linker to link the object code.

Syntax

```
-c
```

Remarks

This option is global.

7.2 -codegen

Instructs the compiler to compile without generating object code.

Syntax

```
-codegen
```

```
-nocodegen
```

Remarks

This option is global.

7.3 -enum

Specifies the default size for enumeration types.

Syntax

`-enum` *`keyword`*

The arguments for *keyword* are:

`int`

Uses `int` size for enumerated types.

`min`

Uses minimum size for enumerated types. This is the default.

7.4 -min_enum_size

Specifies the size, in bytes, of enumerated types.

Syntax

```
-min_enum_size 1 | 2 | 4
```

Remarks

Specifying this option also invokes the `-enum min` option by default.

7.5 -ext

Specifies which file name extension to apply to object files.

Syntax

`-ext` *`extension`*

`extension`

The extension to apply to object files. Use these rules to specify the extension:

- Limited to a maximum length of 14 characters
- Extensions specified without a leading period replace the source file's extension. For example, if *extension* is "`o`" (without quotes), then `source.cpp` becomes `source.o`.
- Extensions specified with a leading period (`.`*extension*) are appended to the object files name. For example, if `extension` is "`.o`" (without quotes), then `source.cpp` becomes `source.cpp.o`.

Remarks

This command is global. The default setting is `.o`.

7.6 -strings

Controls how string literals are stored and used.

Remarks

`-str[ings]` *`keyword`*`[, ...]`

The *keyword* arguments are:

`[no]pool`

All string constants are stored as a single data object so your program needs one data section for all of them.

`[no]reuse`

All equivalent string constants are stored as a single data object so your program can reuse them. This is the default.

`[no]readonly`

Make all string constants read-only. This is the default.

8 Command-line options for optimization

8.1 -inline

Specifies inline options. Default settings are `smart`, `noauto`.

Syntax

`-inline` *`keyword`*

The options for *keyword* are:

`off | none`

Turns off inlining.

`on | smart`

Turns on inlining for functions declared with the `inline` qualifier. This is the default.

`auto`

Attempts to inline small functions even if they are declared with `inline`.

`noauto`

Does not auto-inline. This is the default auto-inline setting.

`deferred`

Refrains from inlining until a file has been translated. This allows inlining of functions in both directions.

`level=`*`n`*

Inlines functions up to *n* levels deep. Level 0 is the same as `-inline on`. For *n*, enter 1 to 8 levels. This argument is case-sensitive.

`all`

Turns on aggressive inlining. This option is the same as `-inline on`, `-inline auto`.

8.2 -O

Sets optimization settings to `-opt level=2`.

Syntax

`-O`

Remarks

Provided for backwards compatibility.

8.3 -O+

Controls optimization settings.

Syntax

`-O+`*`keyword`* `[,...]`

The *keyword* arguments are:

`0`

Equivalent to `-opt off`.

`1`

Equivalent to `-opt level=1`.

`2`

Equivalent to `-opt level=2`.

`3`

Equivalent to `-opt level=3`.

`4`

Equivalent to `-opt level=4,intrinsics`.

`p`

Equivalent to `-opt speed`.

`s`

Equivalent to `-opt space`.

Remarks

Options can be combined into a single command. Command is case-sensitive.

8.4 -opt

Specifies code optimization options to apply to object code.

Remarks

`-opt`*`keyword`* `[,...]`

The *keyword* arguments are:

`off | none`

Suppresses all optimizations. This is the default.

`on`

Same as `-opt level=2`

`all | full`

Same as `-opt speed,level=4,intrinsics,noframe`

`l[evel]=`*`num`*

Sets a specific optimization level. The options for *num* are:

- 0 – Global register allocation only for temporary values. Equivalent to #pragma optimization_level 0.
- 1 – Adds dead code elimination, branch and arithmetic optimizations, expression simplification, and peephole optimization. Equivalent to #pragma optimization_level 1.
- 2 – Adds common subexpression elimination, copy and expression propagation, stack frame compression, stack alignment, and fast floating-point to integer conversions. Equivalent to: #pragma optimization_level 2.
- 3 – Adds dead store elimination, live range splitting, loop-invariant code motion, strength reduction, loop transformations, loop unrolling (with -opt speed only), loop vectorization, lifetime-based register allocation, and instruction scheduling. Equivalent to optimization_level 3.
- 4 – Like level 3, but with more comprehensive optimizations from levels 1 and 2. Equivalent to #pragma optimization_level 4.

For num options 0 through 4 inclusive, the default is 0.

[no]space

Optimizes object code for size. Equivalent to #pragma optimize_for_size on.

[no]speed

Optimizes object code for speed. Equivalent to #pragma optimize_for_size off.

[no]cse | [no]commonsubs

Common subexpression elimination. Equivalent to #pragma opt_common_subs.

[no]deadcode

Removes dead code. Equivalent to #pragma opt_dead_code.

[no]deadstore

Removes dead assignments. Equivalent to #pragma opt_dead_assignments.

[no]lifetimes

Computes variable lifetimes. Equivalent to #pragma opt_lifetimes.

[no]loop[invariants]

Removes loop invariants. Equivalent to #pragma opt_loop_invariants.

[no]prop[agation]

Propagation of constant and copy assignments. Equivalent to #pragma opt_propagation.

[no]strength

Strength reduction. Reducing multiplication by an array index variable to addition. Equivalent to #pragma opt_strength_reduction.

[no]dead

Same as -opt [no]deadcode and [no]deadstore. Equivalent to #pragma opt_dead_code on|off and #pragma opt_dead_assignments.

[no]peep[hole]

Peephole optimization. Equivalent to `#pragma peephole`.

`[no]schedule`

Performs instruction scheduling.

`display | dump`

Displays complete list of active optimizations.

9 Command-line options for eTPU code generation

9.1 -kif | -keep_ intermediate_ files

Keep intermediate files

9.2 -lpm

Use linking process model. In this mode, the compiler creates a separate object file for each compilation unit and the linker links them all together. In the normal mode, all files are compiled together one after the other. For this reason, in the normal mode it is not possible to use libraries or any other old object file and link it together with another object file.

9.3 -big_memory_model

Big memory model. Use indirect jumps. This is useful when using lpm and the linker issues errors, which imply that a jump is too long.

9.4 -not_engine_relative

Do not use engine relative addressing mode in etpu2.

9.5 -no_32bit_err

Do not issue an error for 32 bit arithmetic operations.

9.6 -warn_data

Warn about stack and static data usage.

9.7 -[no]sched

Schedule assembly instructions. This is the default.

10 Working with the assembler

This chapter explains the enhanced Time Processing Unit (eTPU) assembler, and shows you how to use it with assembly source code.

This chapter contains these topics:

- *Understanding the eTPU assembler*
- *Using the command-line assembler*
- *Assembly file layout*
- *Instructions and directives*
- *eTPU assembler preprocessor*

10.1 Understanding the eTPU assembler

The eTPU assembler processes assembly-language source statements written for SPC56xx family of communication microcontrollers. The assembler translates source statements into object files with a format compatible with other eTPU assembler software and hardware products.

The assembler processes assembly source files by reading the contents and preprocessing each line, as described in *Section 10.5: eTPU assembler preprocessor*.

The assembler parses each line of code (as described in *Section 10.3: Assembly file layout*) in order to verify correct syntax. It then encodes all recognized instructions and directives as object code in the specified output layout.

10.2 Using the command-line assembler

This section shows you how to invoke the assembler from the command line, for files outside of an integrated development environment (IDE).

To run the assembler command-line executable, type the full path to the executable at the Windows command prompt.

Optionally, you can add the path of the eTPU tools folder to your `PATH` environment variable. Then you can simply enter the name of the tool, `etpu_bins.exe`, to run the assembler.

10.2.1 etpu_bins

etpu_bins is a wrapper used for all binary utilities such as assembling and linking.

- Assembler is invoked with `etpu_bins --asm` *`inputFile`*
- Linker is invoked using `etpu_bins --ld` *`inputFile`*
- Disassembler is invoked using `etpu_bins --elfdump` *`inputFile`*
- Size utility is used with `etpu_bins --size` *`inputFile`*

The remaining chapter relates to the assembler tool. The linker behaviour is described in the next chapter. A single input file is allowed. The `-i` (pre-include) option can be used to specify more input files.

10.2.2 File extensions

In case the type of file makes a difference to the tool, the type is taken from the file's suffix. Extensions and their meanings are shown in *Table 3*.

Table 3. File extension meanings

Extensions	Type of file
.s, .asm, .uasm, .ucode	An assembly source file
.h	A C or assembly header file
.c	A C source file
.o, .obj, .eln	An Elf relocatable file
.elf, .eld	An Elf executable file
.a, .lib	A library file
.lcf	A linker command file
.d	A makefile dependency file
.map	An assembly map text file
.srx, .srec	An S-records file

In order to force one of these types on an arbitrary given input file, the option `-ex` can be used with one of the regular extensions known to represent the requested type. For example, `-ex .o` is used to convince `etpu_bins` that its input is an Elf relocatable object file.

When no output format (`-elf` or `-srx`) is explicitly selected, as in the example `etpu_bins --asm file.s`, the tool will check syntax validity and no output will be produced. When the `-srx` switch is added, the created file will have the extension `.srx` (not `.srec`).

10.2.3 Command-line syntax

The command-line syntax for the assembler is:

etpu_bins --asm [*options*] *inputFile*

The assembler does not require special filename extensions for input file names and ignores the actual filename extension specifed on the command line. Instead, the assembler uses the base filename to append appropriate extensions when generating the output file names.

For example, this command line causes the assembler to assemble `file.s` into an S-record file named `file.srx`:

etpu_bins --asm -srx file.s

The assembler normally reports error messages to `stderr`, but you can redirect error messages to a file by supplying the `-err` *`filepath`* command-line switch. The assembler indicates the total number of error messages in the tool exit status.

Note: To detect and report invalid instruction sequences, use the `-lint` command-line switch.

10.2.4 Command-line switches

All command line switches begin with the dash (`-`) character. If a switch requires an input value, you can enter it as a space-delimited argument immediately following the switch. Alternatively, you can attach the value to the switch name with the equals (`=`) character. For example, both of these are valid:

```
-o outfile
```

```
-o=outfile
```

When the value is optional the argument must use the '=' notation (i.e. `-switch=val` and not `-switch val`). Each switch name determines whether the following command-line word will serve as its argument, will be the next switch, or is an input file. For example, the following command line causes the assembler to assemble `file.s` into an S-record file named `new_name.srec`:

```
etpu_bins --asm -o new_name.srec -srx file.s
```

The assembler processes its arguments from left to right. For example, the following command line defines the symbol `one`, reads the input file `pre_inc.s`, defines the preprocessor macro `TWO`, then assembles `main.s`:

```
etpu_bins --asm -d one -i pre_inc.s -D TWO main.s
```

Note: In the example above, the file `pre_inc.s` cannot use the preprocessor macro `TWO`, since the macro is only defined after the assembler processes `pre_inc.s`.

Table 4 describes each of the command-line switches the assembler supports.

Table 4. Assembler command-line switches

Options	Switch	Description
General options	-h	Print a short help message and quit
	-V	Print the version of the `etpu_bins` tool and quit
	-f *filepath*	Read more command-line arguments from *filepath*
	-ex ext	Treat input as having extension ext
	-arch arch	Print a list of the supported architectures

Table 4. Assembler command-line switches (continued)

Options	Switch	Description
Output options	-o *filename*	Set the output file name to *filename*
	-srx	Create output file in S-Record format with filename extension `.srx`
	-elf	Create output file in relocatable ELF format with filename extension `.elf`
	-map -src [=*val*]	Create a text file with filename extension `.map` where each line shows the address and data generated from the original source The optional argument, *val*, is a consecutive string of one or more of these characters: – m — expand multi-line macros – a — show addresses – s — show memory address space – j — show include nest levels – n — show source line numbers – r — show relative address offsets – g — show debug sections – w — track source file changes To specify an explicit map file name use ':<file name>' as last sub-option (for example -map=w:out_map will name the map file 'out_map'.
	-sym	Create a text file with the `.sym` filename extension, listing all global symbols defined in the source code
Debugging options	-kl	Where relevant, keep track of local labels defined in source code
	-g	Effective only when -elf is used generate Dwarf2 debugging sections

Table 4. Assembler command-line switches (continued)

Options	Switch	Description
Processor options	-E	Expand all macros and other preprocessor directives and operations to stdout — the `-o` switch can be used to save results to a file
	-D *sym=val*	Set the value of symbol *sym* to *val* as if defined by a source code directive (`#define sym val`). *Note: The assembler processes macro-related switches from left to right on the command line.*
	-U *sym*	Clear the value of symbol *sym* as as if undefined by a source code directive (`#undef sym`) *Note: The assembler processes macro-related switches from left to right on the command line.*
	-d *sym=val*	Set the value of assembly symbol *sym* to *val* as if defined using the `.equ` directive (`.equ sym val`) *Note: The assembler processes macro-related switches from left to right on the command line.*
	-dg *sym=val*	Set the value of assembly symbol *sym* to *val* as if defined using the `.equ` directive (`.equ sym val`), and declare it a global symbol *Note: The assembler processes macro-related switches from left to right on the command line.*
	-I *path*	Append *path* to the user path (see *Section : User and system paths*)
	-IS *path*	Append *path* to the system path (see *Section : User and system paths*)
	-i *filepath*	Include file *filepath* in the code before processing any following argument input files
	-is *filepath*	Include the system file *filepath* in the code before processing any following argument input files
	-M *filepath*	Emit Makefile rules for all input files to the file *filepath* (Each rule makes its target dependant on all included source files.)
	-MM *filepath*	Emit Makefile rules for all input files to the file *filepath*, omitting any system files (Each rule makes its target dependant on all included source files.)
	-[no]sys	Treat '`#include <...>`' as system files
Error handling options	-err *filepath*	Redirect error messages to *filepath* rather than to standard error
	-Wall	Print extra (more strict) error messages
	-Werror	Treat warnings as errors
	-Wnone	Cause warnings not to be issued
Optimising options	-sched	Schedule Instructions

Table 4. Assembler command-line switches (continued)

Options	Switch	Description
Miscellaneous options	-lint	Dump a list of detected possible errors to stderr (no object file is created)
	-extern	Assume all undefined symbols are external
	-global	Assume all defined symbols are global
Reserved options	-ide	Reserved for IDE invocations

10.3 Assembly file layout

This section explains the assembly source file layout. eTPU assembly language includes mnemonic operation codes for machine instructions in the microcontroller's instruction set and provides mnemonic directives for specifying assembler auxiliary actions. It also explains how to define and use macro instructions with predefined statement sequences, and how to use conditional assembly code.

10.3.1 Instructions, directives and packets

The eTPU processes (fetches and executes) 32 bits words. Each word contains one or more instructions that are all executed in parallel (but see "parallelism issues" in the eTPU block guide for exceptions). The set of instructions encoded into or decoded out of a single memory word is called 'a packet'.

Note: Another terminology is using the pair Instruction/Sub-instruction for referring to Packet/Instruction respectively.

10.3.2 Syntax

Each instruction (or directive) appears in the code as a separate line. To combine several instructions into a packet, one can either write the instructions in the same line or surround them with curly braces ('{', '}'). Inside a packet, instructions are separated by the semicolon char (';') or optionally (when braced) by the newline char. Eempty lines inside braced packets are ignored.

For example, if I1 and I2 are instructions, the following are legal packets:

- I1
- { I1 }
- I1 ; I2
- { I1 ; I2 }
- { I1
I2 }
- {
I1 ;
I2
}

Note: Directives can not take part in packets.
Labels are not part of packets. they can only precede one.

A single instruction can not cross line boundaries (i.e. all of the instruction's string must reside in the same line).
Comments inside packets should follow the last instruction in their line.

10.3.3 Statement layout

Programs written in assembly language consist of a sequence of statements, each occupying one line of text.

Note: You can extend a single statement to several lines by ending each partial line of the sequence with the line-continuation symbol, a backwards slash character (\).

Note: Lines, including extended lines, can span up to a maximum of 512 characters.

Each source statement has the following syntax:

`label: instruction comment (each field is optional)`

Labels is the left-most non-blank token, and must follow immediately by a colon character (`:`). Labels are valid symbols (see *Section 10.3.6: Symbols*).

Labels, instructions, mnemonics, directives, attributes, registers, and so on, are all case sensitive.

Instructions, as well as assembly directives, use the following format:

`mnemonic[attributes] [operands]`

Note: The mnemonic field determines the number and format of the attributes and operands fields.

- *Mnemonics* are machine instructions and aliases described in the eTPU assembly manual.
- *Attributes* are optional extensions to instructions that can control the behavior of the CPU during execution of a given instruction. Attributes are concatenated to instructions by a decimal character (`.`) and the attribute name, using this syntax:

`mneumonic[.attribute1][.attribute2] (and so on)`

Attributes are divided into groups, each group controlling a specific aspect of behavior. For instance, the logical and instruction, `and` has the attribute group: `ccsv (with four members .f .f8 .f16 and a default, nameless, member)`. This group controls the sampling of conditional codes.

Note: Some attribute groups have default values that you can omit, while others require an explicit value.

For example:

```
movei.f a,1
```

This `movei` instruction has the attributes that sets appropriate condition flags in the status register. The attribute group involved has two attributes the default attribute does not have an explicit name.

Operands appear in the *instruction* portion of a command as a list separated by the comma character (`,`). Operands describe hardware entities such as addresses, registers, sizes, and so on, that are subject to manipulation by instructions.

Comments begin at the left-most occurrence of two consecutive forward slash characters (`//`) and continue until the end of the line.

As described in the eTPU Assembler Manual, the elements you can use in operands are:

- *Section 10.3.4: Registers*
- *Section 10.3.5: Integer immediate values*

Note: Exact operand format and usage depend on the related instruction.

10.3.4 Registers

Register names appears in the eTPU assembler manual and are considered reserved names, which means that these names cannot be used as identifier names for labels or symbols.

Some instructions also mention bit names. Bit values are written as a register and bit combination (i.e. register[bitNumber]), or better, when given a specific name in the *eTPU Assembler Manual*, as a lowercase symbol. To determine exact use, consult the instruction's description.

10.3.5 Integer immediate values

Use immediate integer values (in decimal, hexadecimal or binary notation) to describe absolute and relative addresses and constants that are part of numeric calculations.

In most cases, you can replace an integer constant with a constant arithmetic expression, using a combination of these operators: `+`, `-`, `<<`, `>>`, `*`, `/`, `&`, `&&`, `|`, `||`, `!`, `~`, `==`, `!=`, `<`, `<=`, `>`, `>=`, and parentheses. Priority and interpretation follows the C language standard. The assembler stores and manipulates integer constants by using 32-bit signed arithmetic.

10.3.6 Symbols

Symbols are placeholders for integer constants. You can use symbols wherever a single integer constant is required. Symbol names cannot contain spaces and can consist of alpha-numeric characters, the underscore character (_), and the decimal character (`.`). Symbol names can be up to 128 characters long.

Some symbol names are reserved for special purposes. Currently these include all names beginning or ending with the underscore character (_) or decimal character (`.`), as well as all registers and bit names.

Declare and define symbols to assign them value, scope, size and other attributes.

10.3.7 Defining a symbol

You can assign values to symbols in two ways:

- Create a label — the value is the offset of the defined label, relative to the beginning of the section in which the label definition appears
- Use the '`.equ` *`symbol, value`*' or '*`.label symbol, value`*' directives — the value is explicitly specified in the definition

10.3.8 Scope

A symbol's scope is either local or global. Symbols not otherwise declared are local, meaning they are defined and used only within the file in which they appear. To use a symbol outside the file in which it is defined, you must use the `.global` *`symbol`* directive to declare that symbol as global so it will be visible to code in other files. To allow code in other files to use the symbol, you must use the `.extern symbol` directive to declare the symbol as external.

10.3.9 Weak symbols

Weak symbols, defined using the `.weak symbol` directive, are treated as global by the assembler, however the linker handles them differently. When the linker resolves an external symbol, the linker attempts to use the (single) global definition for that symbol. If no such definition exists, the linker uses the first corresponding weak symbol definition it encounters in one of the linked files.

10.3.10 Scope rules

All `.global` declarations take precedence over `.weak` declarations. Both `.global` and `.weak` declarations take precedence over `.extern` declarations. To avoid link errors, you must declare symbols used and not defined in a file as external.

Note: The assembler must be able to fully resolve symbols before you can use them in the definition of an `.equ symbol, value` directive or symbols influencing the address location of the generated code (see Section 10.4.2: Data storage).

10.3.11 Strings

A *string* is a sequence of characters enclosed in double quotes ("). Characters preceded by the backslash character (\) have special meaning, as shown in *Table 5*.

Table 5. Special characters in strings

Special characters	Expands to
\b	backspace
\n	new line
\r	return
\t	tab
\"	double quote
\\	backslash
\x*nn*	hexadecimal equivalent — For example, `\x6B` expands to the letter `k`. *Note: Hexadecimal strings required by the `.hexa` directive do not require the `\x` prefix. Instead, the `.hexa` string is a sequence of two-digit hexadecimal values, each digit represented by one ASCII character. For example, `.hexa "4869"` stores two bytes: `72`, and `105`.*

10.4 Instructions and directives

The assembler does not translate directives directly into machine-language instructions. Instead, directives allow you to control issues such as memory layout (addresses for storing data, data contents, data alignment), symbol manipulation, structural grouping of statements, and so on.

Directives begin with the decimal character (`.`) and feature syntax similar to that of instructions. A label preceding a directive is not considered part of the directive.

10.4.1 Memory spaces and sections

A *section* is a continuous memory block. Sections are basic logical units you can use to organize code and data into groups, controlling size, content and starting point of each group or section. At loading time, the content of the user-defined logical sections and possible system-added extra sections are loaded into actual memory segments.

eTPU architecture has two types of memory space and each loadable section belongs to exactly one of them. Defining code or data from outside a section is not allowed:

- Code Space (denoted `c`) is intended for instruction storage. It features a 32-bit access width, within which the 32 bits instructions are stored (aligned to 4 bytes addresses). The address range for this space is 24 bits.
- Data Space (denoted `d`) is used to load and store data. It also has an access width of 8 bits and an address range of 24 bits.

Note: Debug and other unloadable sections do not relate to a physical memory space and have an imaginary access width of 8 bits.

Use of the term *address* should be understood as a shortcut to the full address. The latter is *`space:addr`* in which *`addr`* is the integer location for the word to be found in the *`space`* memory space. For example: `'nop at c:57'` is the instruction `NOP` located at address 57 in the code memory space. In most cases, however, there is no need for a space reference as it is uniquely inferred by the instruction. The only time space names are explicitly mentioned is when a memory space is assigned to a section using the `.org` *`address`* or the `.section` *`sec`* directive.

Sections can be either relative or absolute. Relative sections are declared by name and space (using the `.section` *`sec`* directive) and are assigned an address during the linking stage. Section names are symbols. When the memory space is omitted, the section is considered a code section. Some predefined special sections (as well as absolute sections, defined below) used by the Elf and Dwarf2 binary formats have names that begin with the dot character (`'.'`). As an exception, these sections might also be pre-assigned to a space (for example, the `.data` section will be loaded as data).

Absolute sections are declared using the `.org` *`address`* directive. They are nameless - the name is automatically built from space and address characteristics - but must have a known memory space and address. When the memory space is omitted it is assumed to be the code space. `.org` sections can not be repeated, however - other address overlaps between absolute sections will be checked and flagged by the linker tool.

Upon meeting a relative section directive, the following content is added (without alignment) at the end of the section portion generated so far.

`.org` sections, on the other hand, can not be repeated. The `.previous` directive can be used to switch back and forth between any two sections. Address overlaps between all sections will be checked and flagged by the linker tool.

10.4.2 Data storage

Each code statement stores 4 bytes into memory. Data can be stored in different sizes and ways.

In general, code and data statements are stored in the memory sequentially, following their order in the text. At any given moment, the storing address is termed *the current location* (see *Section 10.4.4: The current location*) and is relative to the beginning of the current section within which the next statement (code or data) will be stored. The current location can be changed as shown in *Table 6*.

Table 6. Data storage directives

Directive	Action
.section *sec*	Associates the following code and data with section *sec*. See *Section 10.4.1: Memory spaces and sections* for more information.
.org *address*	Store the following code and data at the resolved *address*. This statement begins a new section.
.previous	Revert to the previous section (allows you to toggle between .section and .org sections).
.endsec	End the current section and returns to the previous section on the stack.
.word *integer*	Starting at the current location, stores the numeric word *integer* (in big endian). The word's default size is the current section's natural word width (in bytes), but can be explicitly stated using the .b1, .b2, and .b4 attributes. The storage location is aligned according to word size by default. You can use the attribute .n to skip alignment.
.leb 128	Store a 32-bit integer in LEB128 format (for use in DWARF objects, for instance). Size can vary from one to four bytes. Use attributes .s and .u to store signed and unsigned integers (the default is unsigned).
.hexa *string*	Starting at the current location, stores the hexadecimal bytes *string*. For example, `.hexa "65FF"` stores two bytes: 101, and 255.
.ascii *string*	Starting at the current location, stores the ASCII bytes in *string*.
.asciz *string*	Starting at the current location, stores the ASCII bytes in *string*, and append a zero byte to the end.
.skip *integer*	Increment the current location *integer* bytes. The number of bytes incremented is *integer* times the current section's natural word width (in bytes). You can explicitly specify the number of bytes by using the .b1, .b2, and .b4 attributes. Skipped bytes are filled with zero in the data space, and with 0xFF in the code space.
.align *integer*	Advance the current location (with padding) as required to be aligned on a word address boundary set by 2^*integer* (where *integer* has a range of 1 through 16). You can use the .b1, .b2, and .b4 attributes to specify other word sizes

Note: The `.align` directive will align the current location with respect to the base of the current compilation unit only. If the object is linked with other objects, effective alignment will depend on the linker's configuration.

10.4.3 Symbol directives

During execution all symbols are stored in a single table, the symbol table. Symbols must be defined exactly once by placing them as labels, giving explicit values, using the `.equ` and `.def` directives, or declaring them external through use of the `.extern` directive.

Table 7. Symbol directives

Directive	Description
.equ *symbol*, *value*	Define *symbol* as an absolute symbol, and set its value to *value*. *value* must be a constant expression (all referenced symbols must be resolved).
.global *symbol*	Declare *symbol* as a global symbol whose value is available to other modules outside the current compilation unit. *symbol* must be defined in the same compilation unit.
.extern *symbol*	Declare *symbol* as a symbol defined outside the current compilation unit, whose value is available in the current compilation unit.
.weak *symbol*	Declare *symbol* as a global weak symbol.

10.4.4 The current location

The current location, explicitly referenced by the predefine symbol `'.'`, continuously points to the address of instruction or data storage. Current location units reflect those of the actual memory section (usually one or four bytes) wherein they will be located. For absolute sections, the location is the actual address. For relative sections, the location is calculated as though the section begins at address zero.

Example:

```
jmp.+3
```

The above example transfers control to the third instruction appearing after the current one ('`.`').

Caution: Explicit use of the current location symbol is not considered safe.

10.4.5 Change of flow

Some instructions contain direct jumps to other points in the code. The target of the jump can be specified as any legal expression. However, to create code that is less prone to error and easier to maintain, it is better to make this expression a label. When using the option `-Wall`, etpu_bins will warn if this convention is violated, as illustrated in *Example 4*.

Example 4 Using labels in code

```
   .extern There
   .equ Here,0x7686
Start:
   jmp 0x3075    ; Warning - a constant
   jmp Start     ; OK
```

```
jmp Start+5   ; Warning - an expression
jmp There     ; OK (assuming a label)
jmp Here      ; Warning (a symbol that is not a label)
jmp .         ; Warning (a symbol that is not a label)
```

10.4.6 Code checking

During code analysis, as when the `-lint` option is used, you can assume that:

- Every instruction and label defined in the code is reachable.
- Instructions without a label at their address are reached only after following the previous instruction. This, in turn, is assumed to originate from the previous address.
- For every instruction, its following instruction must also be known (this does not include indirect and scheduler related change of flow instructions).

Deviations from the last convention are flagged as ill-formed code and may cause the tool to reject input, as shown in *Example 5*.

Example 5 Deviations from convention in code

```
Start:
   nop
   move r1,r12          ; can only be executed after the nop
Loop_prefix:
   .align 4             ; Invalid - current instruction is not known
Loop:
   addi r4,1
   .word 0x187983       ; Invalid - code flow interrupted by data
   sub.f r2,r2,r3
   jmp.n zero,Loop
   nop                  ; Invalid - what next?
```

10.5 eTPU assembler preprocessor

The etpu_bins preprocessor enables macro definitions, conditional assembling, and multi-level file inclusion. These are achieved through preprocessor directives and operations. Preprocessor directives features lines with special syntax that are recognized and processed (by the preprocessor) based on their meaning. All directives are placed at the beginning of the line and start with the pound character (#). With 'operations', the directives are placed inside lines and are replaced accordingly by the preprocessor.

All objects handled by the preprocessors are sequences of characters called 'tokens'. Tokens types can be numbers, symbols, attributes, strings and operators. Consequently, the arguments for some preprocessor directives and operations might be limited to specific types (e.g. the '`#include`' directive expects its argument to be a single string) or behave according to the type of the argument (e.g. the '`%str(X)`' operation tests for a single string token).

Operators, however, may require a specific kind or number of tokens as operands. Operations experiencing bad input will either evaluate to the zero token (the `%str(X)` operation tests its argument as a single-string token) or will create an error (the `#include` directive expects its argument to be a single string).

10.5.1 Preprocessor macros

Macros, regular and multi-lined, enable the definition and use of named segments of text. A macro invocation is also named 'a call'. It is necessary to define macros in the code prior to invocation. Code expansion is done during the call (and not while preprocessing the definition).

10.5.2 Regular (single-line) macros

A *single-line macro* is defined on a single line of code using the `#define` directive:

```
#define macroName[optionalArgs] definition
```

- *`macroName`* is a case-sensitive, valid identifier.
- *`optionalArgs`* is an optional comma-separated list of unique identifiers enclosed by parentheses. When a macro has arguments, parentheses must immediately follow the name - no white space is allowed.
- *`definition`* is an arbitrary combination of tokens intended to replace the call.

For example:

```
#define START_OF_FUNC 0x1b3f
```

This example defines `START_OF_FUNC` as a macro with no arguments. The assembler replaces the identifier `START_OF_FUNC` with the number `0x1b3f` wherever the identifier appears in the code.

To substitute the value of an argument within a definition, the argument's name must be placed in the substituting code. For example:

```
#define param(offset,word_size) offset + 4 * word_size
```

This example code replaces the call `param(MY_START, 11)` with `MY_START + 4 * 11`.

Note: The token sequence `4 * 11` does not evaluate to `44` at compile time. To evaluate an expression at compile time, use `%eval: expression evaluation`.

Note: If you define macros with numerical calculations, we recommend that you enclose each argument occurrence (or even the entire definition) with parentheses. Doing so prevents unwanted side effects during evaluation.

For instance, the previous example definition would be better defined as:

```
#define param(offset,word_size) ((offset) + 4 * (word_size))
```

It is possible to nest macros by defining them inside of other macros. Macro expansion normally occurs during invocation and not during compilation (you can use the `#xdefine` directive to expand macros during compilation instead). For example, in the following two lines, the assembler expands the `BBB` macro to `100 + 200` regardless of the order of definition.

```
#define AAA 100
#define BBB AAA + 200
```

Circular definitions are allowed, but invocation of the macro will stop after one level of expansion. For example, the following example code expands the `next(10)` only once to `next(10) + 1`.

```
#define next(a) next(a) + 1

next(10)
```

Note: *All macro definitions have a signature (name and parity). Therefore, the assembler does not generate error messages for calls to macros with the same name that have a different number of arguments; instead, the assembler silently ignores them (no expansion occurs).*

Macro names are not assembly symbols. During assembly, the assembler does not assign macros numeric values. When preprocessing is complete, all macros have been expanded, and their corresponding names cease to exist.

10.5.3 Multi-line macros

A *multi-line macro* includes all code lines between a `#macro` and the closest following `#endm` directive. To invoke a multi-line macro, the macro name must be the first, or left-most, token in the line of code (the name can be prefixed by a label and white space). Multi-line macro arguments are similar to those for single line macros, except that the former are defined and used without parentheses. Multi-line macros cannot be defined within macros of the same type.

For example, the code in *Example 6* defines a macro that, given a register (*R*), creates the value `3 * R + 2` and stores that value in register `a`. A possible invocation of this macro is `do_it p`.

Example 6 Multi-line macro example

```
#macro do_it R
    move a,R
    add a,a,R
    add a,a,R
    addi a,2
#endm
```

Note: *The assembler does not perform type or semantic checking on the arguments. Consequently the macro can be invoked, for example, with the argument `a + 4`, which would result in the expansion of invalid code. Passing `R1` as an argument results in the calculation, with unexpected results, of `4 * a + 2`.*

Local labels inside a macro

When labels are defined or used inside a macro it follows that all macro invocations will use that name. This may be problematic (and, in fact, invalid) as labels are not singly defined. To declare and use labels local to macros, the label name should be prepended with `\@`. For example, the following code causes the assembler to interpret the label `next` in the `epilog` macro differently each time it is invoked:

```
#macro epilog
   jmp.n zero,next\@
```

```
   next\@:
#endm
```

The mechanism implementing this behavior replaces these labels with new ones that use an integer counter beginning with zero and incremented on each call.

Default values for macros

Both types of macros accept default values for their parameters. Default values can be set to the last parameters by using the equal character (=). The value assigned is the following sequence of tokens ending at the first comma met. For the last parameter, default value ends at the closing right parenthesis (in single-line macros) or at the end of the line (in multi-line macros), as shown in *Example 7*.

Example 7 Appropriate macro ending

```
#macro ADD R, A = 2, B = 3
   movei R, A + B
#endm


ADD a, 34
ADD a,78,98
ADD a
```

The macro in *Example 7* expands to the code in *Example 8*.

Example 8 Macro expansion

```
movei a, 34 + 3
movei a, 78 + 98
movei a, 2 + 3
```

Note: The actual number of arguments passed in a call can be obtained using `%0`.

10.6 Macro-related directives

See *Table 8* for macro-related directives.

Table 8. Macro-related directives

Directive	Explanation
#assign	Allows you to quickly delete and redefine the value of a single-line macro with no parameters.
#define	Defines a single-line macro as explained in *Section 10.5.2: Regular (single-line) macros*. Single-line macros can be declared inside multi-line macros.
#endm	Ends a macro definition.

Table 8. Macro-related directives (continued)

Directive	Explanation
#macro	Begins the definition of a multi-line macro. As macro names can override special identifiers, you can create macros that replace ordinary instructions. Single- and multi-lined macros share the same name space; so only one macro type can exist for an identifier.
#rmdef	Deletes all user macro definitions. Effect the same as using the `#undef` directive on all existing user macros.
#undef	Use this directive to reverse the effect of a `#define`, `#xdefine`, or `#macro` directive, cancelling any definitions made for an identifier. For example, `#undef XYZ` causes the assembler not to expand further references to the term `XYZ`.
#xdefine	Similar to the `#define` directive, except that the assembler does not postpone definition expansion until the macro is invoked. Instead, the assembler expands the definition at compile time.

The `#assign` directive

```
#assign <name> <numeric expression>
```

first deletes the macro and then redefines it as if defined using

```
#xdefine <name> %eval(<numeric expression>)
```

An example of this is shown in *Example 9* .

Example 9 Using the #assign directive

```
#define XXX 1
#assign XXX 2
#assign XXX XXX + 1
#assign XXX YYY ; invalid
```

The first three lines above will assign macro XXX the values 1, 2, and 3 respectively. The last line is invalid since the defining expression is not a numeric constant.

10.6.1 Conditional assembly

The assembler can assemble code portions in order to fulfill conditions set by the user conditions defined with clausal directives similar to the `if` statement of the C program, as shown in *Example 10*.

Example 10 Assembler directives

```
#if {condition1}
   // this code is processed only if condition1 is true
#elif {condition2}
   // this code is processed if condition1 is false and
   // condition2 is true
```

```
#else
   // this code is processed if both conditions are false
#endif
```

Note: The `#else` and `#elif` clauses are optional. You can use several `#elif` clauses in succession. You can also nest `#if` conditions.

Another illustration of conditional assembly is *Example 11*.

Example 11 Conditional assembly

```
#if %defined(INTERRUPTS_LEVEL)
   nop
   #if INTERRUPTS_LEVEL < 3
      or.f a,d,a
   #endif
#else
   ori.f a,d,TRNR
#endif
```

#if

The `#if` directive, defined as `#if` *`expr`*, results in the processing of code only if the numeric expression *`expr`* is evaluated to an integer other than zero. All expression elements must be known at the preprocessing point when expressions are met, or error messages occur.

The condition of the `#if` clause is preprocessed before evaluation. Therefore it is possible to use preprocessor macros and operators as parts of the expression.

#ifdef

The `#ifdef` directive, defined as `#ifdef` *`macro`*, results in the processing of code only if the *`macro`* has been defined by the `#define` directive.

#ifndef

The `#ifndef` directive, defined as `#ifndef` *`macro`*, results in the processing of code only if the *`macro`* has not been defined by the `#define` directive.

#elif

The `#elif` directive, similar to `#if`, must follow an `#if` or `#elif` directive. This offers another expression to test in case the previous conditions failed.

#else

The `#else` directive results in the processing of following code lines, in the instance that all conditions defined by the `else's` corresponding `#if` and `#elif` have failed.

#endif

Every `#if` directive must end with a matching `#endif` directive. A file shouldn't end unless its active condition has been concluded.

#abort

The `#abort` directive aborts preprocessor operations (also responsible for reading the input). Assembling will continue only on the lines produced thus far.

#quit

The `#quit` directive terminates overall assembling and, with an exit status of 1, returns control to the user.

#error and #warn

The `#error` directive prints an error message to standard error as specified in its string operand.

Likewise, `#warn` directives produce a warning message.

Note: Any non-string arguments to these directives are subject to macro expansion, thereby allowing messages containing previously calculated values or texts.

#rem

Like comments, `#rem` directive lines are simply discarded by the preprocessor.

10.6.2 Including files

The `#include` *`file`* directive results in the inclusion of the named source file in the code. File names need to be quoted as described below. On all operating systems the slash character (`/`) is used to separate directories and file names.

User and system files

The assembler makes a distinction between user and system files:

- User file names are double quoted (`"user_file"`).
- System files are regular text files that are part of the tool distribution. The files are identified by name and, you are not concerned about their exact location. System files are enclosed by angled brackets instead of double quotes (`<system_file>`).

#include

The directive `#include` *`directory`* is used to begin processing a new file. After completing that file (and all files recursively related) the next line of the current file will be read.

User and system paths

Search for user files in the directory lists within a user path list. When code is executed, this list contains the current directory and the directory from which the tool was invoked. This is then extended to contain new items from each `'#include'` directive met, and the new file's directory is added to the user path.

Searches are retroactive — when files with the same name reside in the path, then the file chosen for inclusion is taken from the path's most recently added directory, unless the file's absolute name is provided. You can explicitly add more directories to the path from within the code by using `#path` *`directory`* or a command line option. All path additions made by a file during its processing are deleted upon file completion. System files are similarly searched in the 'system path'. The path begins from the directory containing the etpu_bins tool.

You are responsible for avoiding repeated inclusion of files. This can be achieved by defining a distinct macro in the included file and testing for its definition.

Example: the following code can be used to prevent multiple inclusion of the file '`foo.def`':

```
#ifndef(FOO_DEF)
   #define FOO_DEF
   ; actual content of file 'foo.def'
#endif // FOO_DEF //
```

#path

The directive `'#path {directory}'` adds directories to the include search path. Enclose the directory name in double quotes. To add a directory to the system path, enclose the directory name in angled brackets.

Note: As this directive seriously limits portability of source code, its use is not generally recommended. It is usually preferable to update the path using the IDE or from the command line.

10.6.3 Preprocessor operations

Most operations begin with the modulo character (`'%'`). This section lists all preprocessor directives and operators. Nested operator behavior, however, is not defined.

%defined

The `'%defined'` operator checks the definition of single-line macros: %defined (MACRO) will evaluate to true (the integer one) if a single-line macro called 'MACRO' exists, otherwise it will evaluate to zero. To test whether or not a macro has been defined, the result of the above noted operation can be negated (e.g., `'#if !%defined(MACRO)'`).

when testing definition of a single macro inside an #if condition, the shorter notation (#ifdef / #ifndef) is preferred

%mac

The `'%mac'` operator checks for the existence of multi-line macros.

%id, %int, %attr, %reg and %str

The condition operators, `%id, %int, %attr, %reg and %str`, test their single token argument (normally an argument passed to a macro) in order to verify belonging to a certain type. The operators perform as follows:

- `%id` tests for identifiers
- `%int` tests for integers
- `%attr` tests for attributes
- `%reg` tests for registers
- `%str` tests for strings.

%streq

The `'%streq'` operator compares the content of two strings.

%len

The `'%len'` operator evaluates the length (number of characters) of a string argument.

Example:

```
%len("abcd") // evaluates to 4
```

%eval

The `'%eval'` operator reads and evaluates an integer expression. The expression is replaced by the result token.

#

The `'#'` operator is used inside single line macros to convert macro arguments into strings. During expansion every macro argument preceded by the character `'#'` is replaced (including the operator token) with the string constant token. The latter is formed from the literal text of the argument.

##

During macro expansion the `'##'` operator is used inside single line macros to paste tokens. Upon expansion the token pairs found on either side of each `'##'` operator (and the operator itself), are replaced by a single token. The single token is a concatenation of a replaced token pair. This operation will be performed only if one of the input tokens is a macro argument and the result of the combination is a valid token.

%#()

This operator will freeze its result, i.e., during subsequent preprocessor activity the result will not be further expanded automatically. Before doing so it might also do the following:

- For one argument that is a string, the string will be broken into individual tokens.
- For more than one argument, the comma separated list of arguments will be concatenated "as is" into a single identifier token.

%##()

Works like %#, but expands its arguments first.

%*()

Turns all tokens of its single argument non-frozen.

%**()

Same as %*, but expands its argument first

These operators can be used to create sophisticated macros, as demonstrated in the Dwarf2 header that comes with the standard distribution. However, as they tend to make the code harder to follow, use should be considered with care.

10.6.4 Predefined macros

The assembler defines a set of macros available for use.

__VERSION_NUM__

This macro expands to six digits hexadecimal integer `0xJJNNCC` where JJ, NN and CC are, respectively, the major, minor, and micro version numbers.

__VERSION__

This macro expands to a string containing the current version number.

__FILE__

This macro expands to a string and features the original file name (as it appears in the command line or the including `'#include'` directive) within which it appears.

__ABS_FILE__

This macro expands to a string and features the absolute path to the file name (as it appears in the command line or the including `'#include'` directive) within which it appears.

__LINE__

This macro expands to the current number of source lines.

__DATE__

This macro expands to a string containing the current date.

__TIME__

This macro expands to a string containing the current time.

11 Working with the ELF linker

This chapter documents the Executable and Linkable Format (ELF) Linker. The linker is a command line tool used to join relocatable ELF object files into an executable ELF file. The linker accepts the names of object and library files as arguments and produces its result by arranging their content according to directives and templates that reside in a Linker Command File. Most aspects of the linking process are described in the LCF file, and some can be controlled by using command line arguments (for contradicting options - the command line ones take precedence over the LCF).

The ELF Linker has several extended functions that allow you to manipulate code in different ways. You can define variables during linking, control the link order to the granularity of a single section, and change the alignment.

You access these functions through commands in the linker command file (LCF). The linker command file has its own syntax complete with keywords, directives, and expressions, that you use to manipulate the linker. The command file syntax and structure is similar to that of a programming language, and is described in these sections:

- *Invocation and command line switches* describes the command line switches
- *Structure of linker command files* describes command file organization
- *Linker command file syntax* shows how to direct the linker for specific tasks
- *Alphabetical keyword listing* an alphabetical listing of LCF functions and commands
- *Code and data sections* shows how to determine to which memory space a section loads

11.1 Invocation and command line switches

This section shows the command line switches that linker supports.

Following is the syntax for linker usage:

```
etpu_bins --ld <linker arguments>
```

Table 9 describes each of the command line switches that linker supports.

Table 9. Linker command line switches

Options	Switch	Description
General options	-h -help	Display usage message
	-V -version	Display version number
	-err <file>	Log errors to file
	-o -out <file>	Specify output file name
	-f <file>	Read more arguments from a file
	-arch <string>	Choose architecture

Table 9. Linker command line switches (continued)

Options	Switch	Description
Linking options	-lcf -script <file>	Use a linker command file (a file with suffix .lcf is also considered as the LCF file)
	-xlcf	Do not warn when using internal lcf (by default, in case no LCF file is given, the linker issues a warning and uses a trivial LCF template)
	-e -m -main <sym>	Set main entry point (this is the address execution will start from)
	-L <dir>	Add <dir> to library search path
	-l <file>	Link library lib<file>.<ext> or <file>
	-open_libs	Consider all libs unconditionally
	-d	Perform sections dead stripping
	-[no]links_abs	Link all absolute ('.org') sections
	-zerobbs	Expand and zero-initialize .bss data section
	-T <mem=addr>	Set segment <mem> start address to <addr>
Debug options	-g	Keep debug information
	-log <string>	Log link closure to file
	-map	Generate link map file
	-[no]check	Check objects compatibility (recommended)
	-t -trace	Print name of input files upon processes
	-y -trace_sym <string>	Show all occurrences of symbol in objects
	-[no]check_segments	Check segments address overlapping
	-mseg	Use segments names for output sections

11.2 Structure of linker command files

Linker command files consist of the following three main segments that should appear in this order in each file:

- Mandatory—*Memory segment*—maps memory segments
- Mandatory—*Sections segment*—defines segment contents
- Optional—*Closure blocks*—forces functions into closure

11.2.1 Memory segment

The memory segment divides available memory into segments. *Section 11.4.6: MEMORY* explains this segment type. *Example 12* shows an example `MEMORY` segment.

Example 12 Example MEMORY segment

```
MEMORY {
   segment_1 (RWX): ORIGIN = 0x800000, LENGTH = 0x190
```

```
   segment_2 (RX): ORIGIN = 0x801000, LENGTH = 0x19000
}
```

The `(RWX)` portion specifies these ELF flags:

- `R` — read
- `W` — write
- `X` — executable code

`ORIGIN` represents the memory segment's *start address*. The address may also denote the relevant memory space (c:0x800000), if the memory space is omitted, then the memory space is determined according to the ELF flags.

`LENGTH` represents the memory segment's *size*.

Note: If you cannot predict how much space a segment requires, you can use the function `AFTER` *and* `LENGTH = 0` *(unlimited length) to have the linker automatically fill in the unknown values.*

11.2.2 Sections segment

The sections segment defines the contents of memory segments and defines global symbols used in the output file. *Section 11.4.8: SECTIONS* explains this segment type. *Example 13* shows an example `SECTIONS` segment.

Example 13 Example SECTIONS segment

```
SECTIONS {
   .section_name :     //the section name is for your reference
   {                   //the section name must begin with a '.'
      filename.o (.text) //put the .text section from filename.o
      filename2.o (.text) //then the .text section from filename2.o
      . = ALIGN (0x10);   //align next section on 16-byte boundary.
   } > segment_1          //this means "map these contents to
segment_1"
   .next_section_name:
   {
      (more content descriptions)
   } > segment_x         // end of .next_section_name
definition}                        // end of the sections block
```

11.2.3 Closure blocks

The linker can automatically remove unused code and data (see *Section 11.3.4: Dead strip prevention*). Sometimes, however, certain symbols must be kept in output files, even if code does not directly reference those symbols. For example, interrupt handlers are usually linked at special addresses without any explicit jumps to transfer control to these addresses.

Closure blocks allow you to prevent the linker from dead stripping specified symbols. The closure is transitive — all symbols referenced by the closed symbol are also forced into closure.

There are two types of closure blocks:

- *Symbol-level closure blocks*
- *Section-level closure blocks*

Symbol-level closure blocks

Use `FORCE_ACTIVE` when you want to include a symbol in the link that would not be otherwise included. For example:

```
FORCE_ACTIVE {break_handler, interrupt_handler, my_function}
```

Section-level closure blocks

Use `KEEP_SECTION` when you want to keep a section (usually a user-defined section) in the link. For example:

```
KEEP_SECTION {.interrupt1, .interrupt2}
```

A variant is `REF_INCLUDE`. It keeps a section in the link, but only if the file from which the section comes is referenced. This is very useful for including version numbers. For example:

```
REF_INCLUDE {.version}
```

11.3 Linker command file syntax

This section describes some practical ways in which you can use the commands of the linker command file to perform common tasks.

The topics in this section are:

- *Alignment*
- *Arithmetic operations*
- *Comments*
- *Dead strip prevention*
- *Expressions, variables and integral types*
- *File selection*
- *Writing data to memory*

11.3.1 Alignment

Use the `ALIGN` command to align data on a specific byte-boundary. For example, the following fragment uses `ALIGN` to increment the location counter to the next 16-byte boundary. *Example 14* shows an example of using the `ALIGN` command.

Example 14 Example ALIGN command

```
file.o (.text)
 ALIGN (0x10);
file.o (.data) // this is aligned on a 16-byte boundary
```

For more information, read Section 11.4.2: ALIGN.

11.3.2 Arithmetic operations

You can use standard C arithmetic and logical operations when you define and use symbols in linker command files. Table 10 shows the order of precedence for each operator. All operators are left-associative. To learn more about C operators, refer to the *C Compiler*.

Table 10. Arithmetic operators

Precedence	Operators
1 (highest)	- ˜ !
2	* / %
3	+ -
4	>> <<
5	== != > < <= >=
6	&
7	|
8	&&
9 (lowest)	||

11.3.3 Comments

You can add comments to your file by using the C++ style double slash characters (`//`), C-style slash and asterisks (`/*`, `*/`). The linker ignores comments. For example, the comments shown in Example 15 are valid comments.

Example 15 Example comments

```
/* This is a
             multiline comment */
* (.text) // This is a partial-line comment
```

You can also place comments in a special section,.comment.

```
.comment_section :
{
   *(.comment)
} > .comment
```

11.3.4 Dead strip prevention

Linkers remove unused code and data from the output file in a process known as *dead stripping*. To prevent the linker from stripping unreferenced code and data, use the `FORCE_ACTIVE`, `KEEP_SECTION`, and `REF_INCLUDE` directives. Details about these directives can be found in Section 11.4.3: FORCE_ACTIVE, Section 11.4.4: KEEP_SECTION, Section 11.4.7: REF_INCLUDE and Section 11.4.5: FORCE_FILE.

11.3.5 Expressions, variables and integral types

This section describes various expressions, and variable and integral types.

Variables and symbols

Symbol names in a linker command file consists of letters, digits, and underscore characters. *Example 16* shows examples of valid symbol names. Traditionally, symbols defined inside a linker command file start with an underscore character

Example 16 Valid symbol names

```
_dec_num = 99999999;

_hex_num_ = 0x9011276;
```

Expressions and assignments

You can create global symbols and assign addresses to these global symbols using the standard assignment operator, as shown:

```
_symbolicname = some_expression;
```

You must place a semicolon at the end of each assignment statement.

You must place assignments only at the start of an expression. For example, the linker would reject the following expression:

```
_sym1 + _sym2 = _sym3;  // ILLEGAL!
```

When the linker evaluates an expression and assigns it to a variable, the linker gives it either an absolute or a relocatable type. An *absolute* expression type is one in which the symbol contains the value that it will have in the output file. A *relocatable* expression is one in which the value is expressed as a fixed offset from the base of a section.

Integer types

The syntax for linker command file expressions is very similar to the syntax of the C programming language. The linker stores and manipulates integer constants by using 32-bit signed arithmetic.

Octal integers (commonly known as *base eight integers*) start with a leading zero, followed by numerals in the range zero through seven. For example, here are some valid octal patterns you could put in your linker command file:

- _octal_number = 01234567;
- _octal_number2 = 03245;

Decimal integers start with non-zero numeral, followed by numerals in the range of zero through nine. Here are some examples of valid decimal integers you could put in your linker command file:

- _dec_num = 99999999;
- _decimal_number = 123245;
- _decyone = 9011276;

Hexadecimal (base 16) integers start with 0x or 0X (a zero followed by an X), followed by numerals in the range of zero through nine, and/or characters A through F. Here are some examples of valid hexadecimal integers you could put in your linker command file:

- _somenumber = 0x999999FF;
- _fudgefactorspace = 0X123245EE;
- _hexonyou = 0xFFEE;

To create a negative decimal integer, use the minus sign (-) in front of the number, as in:

- _decimal_number = -123456;

11.3.6 File selection

When defining the contents of a `SECTION` block, you must specify the source files that are contributing their sections. The standard way of doing this is to simply list the files, as shown in *Example 17*.

Example 17 Source files listing

```
SECTIONS {
  .example_section :
    {
       main.o  (.text)
       file2.o (.text)
       file3.o (.text)
    }
 }
```

In a large project, the list can become very long. For this reason, the `*` keyword can be used to represent the filenames of every file in your project. Note that since we have already added the `.text` sections from the files `main.o`, `file2.o`, and `file3.o`, the '`*`' keyword will not add the `.text` sections from those files again.

```
* (.text)
```

11.3.7 Writing data to memory

You can write data directly to memory using the `WRITE`x commands in the linker command file.

- `WRITEB` writes a byte
- `WRITEH` writes a two-byte half word
- `WRITEW` writes a four-byte word.
- `WRITES` writes a string. The data is inserted at the section’s current address.

The example in *Example 18* shows examples of the WRITEx commands.

Example 18 Embedding data directly into the output.

```
.example_data_section :
{
   WRITEB (0x48);  /*  'H'  */
   WRITEB (0x69);  /*  'i'  */
   WRITEB (0x21);  /*  '!'  */
   WRITES ("Hi!")
} > example_data_section
```

11.4 Alphabetical keyword listing

Table 11 lists all the functions, keywords, directives, and commands that linker command files can include.

Table 11. Linker command file keywords

. (location counter)	*SECTIONS*
ALIGN	*WRITEB*
FORCE_ACTIVE	*WRITEH*
KEEP_SECTION	*WRITES*
MEMORY	*WRITEW*

11.4.1 . (location counter)

The period character (`.`) always maintains the current position of the output location. Since the period always refers to a location in a `SECTIONS` block, it cannot be used outside a section definition.

`.` can appear anywhere a symbol is allowed. Assigning a value to `.` that is greater than its current value causes the location counter to move, but the location counter can never be decremented.

This keyword can be used to create empty space in an output section. In the example that follows, the location counter is moved to a position that is 0x10000 bytes past the symbol `__start`.

Example 19 Moving the location counter

```
.data :
{
     *.(data)
     *.(bss)
     *.(COMMON)
     __start = .;
     . = __start + 0x10000;
```

```
    __end = .;
} > DATA
```

11.4.2 ALIGN

Advance the location counter so that it will be aligned on a boundary specified by the value of `alignValue`.

Prototype

`ALIGN(`*`alignValue`*`)`

Parameter

`alignValue`

The number of address lower bits that should be cleared, for example - ALIGN(3) will align to the nearest aligned 8 byte address.

11.4.3 FORCE_ACTIVE

Allows you to specify symbols that you do not want the linker to dead strip. When using C++, you must specify symbols using their mangled names.

Prototype

`FORCE_ACTIVE{ symbol[, symbol] }`

11.4.4 KEEP_SECTION

Allows you to specify sections that you do not want the linker to dead strip.

Prototype

`KEEP_SECTION{ sectionType[, sectionType] }`

11.4.5 FORCE_FILE

Allows you to specify files that you do not want the linker to dead strip.

Prototype

`FORCE_FILE{file[, file]}`

11.4.6 MEMORY

Allows you to describe the location and size of memory segment blocks on the target system. You use this directive to tell the linker the memory areas to avoid, and the memory areas into which it should link your code and data.

Note: The linker command file must contain only one `MEMORY` directive. Within the confines of the `MEMORY` directive, however, you can define as many memory segments as you wish.

Prototype

`MEMORY {`*`memory_spec`*`}`

where `memory_spec` is one or more lines in this format:

`segmentName (flags) : ORIGIN = address, LENGTH = length [> fileName]`

Parameters

`segmentName`

The name of the segment. This name must be a consecutive string of alphanumeric and/or underscore (_) characters.

`flags`

Access flags for the output file (`Phdr.p_flags`). Flags can be any combination of:

- `R -` read
- `W -` write
- `X -` executable code

`address`

One of the following:

- A memory address (optionally including the memory space name), in hexadecimal format, such as `0x800400`.
- An `AFTER` command — If you do not want to compute the addresses using offsets, you can use the `AFTER(name [,name])` command to tell the linker to place the memory segment after the specified segment.

Note: If you specify multiple memory segments as parameters for `AFTER`, the linker uses the segment with the highest memory address. This is useful when you do not know which overlay takes up the most memory space.

`length`

One of the following:

- A value greater than zero indicating the size (in bytes) of the segment. If you try to put more code and data into a memory segment than your specified length allows, the linker generates an error at link time.
- Zero (linker automatically calculates the segment size)

Note: The linker does not perform overflow checking when you specify zero. If you do not leave enough memory free to hold the entire segment, you will get unexpected results. For this reason, we recommend that whenever you specify zero, you also use the `AFTER` keyword to specify the start address.

`> fileName`

An optional argument to have the linker write the segment to a binary file on disk instead of an ELF program header. The linker places this file into the same folder as the ELF output file. This option has two variants:

- `> fileName` — writes the segment to a new file
- `>> fileName` — appends the segment to an existing file

In *Example 20*, the linker places `overlay1` and `overlay2` immediately after the `code` segment. The linker places `data` immediately after the overlay segments.

Example 20 MEMORY example

```
MEMORY {
    code   (RWX) : ORIGIN = 0x800400, LENGTH = 0
    data   (RW)  : ORIGIN = 0, LENGTH = 0
    data1  (RW)  : ORIGIN = AFTER (data), LENGTH = 0
}
```

11.4.7 REF_INCLUDE

Allows you to specify sections that you do not want the linker to dead strip, but only if they satisfy this condition: the file that contains the section must be referenced. This is useful if you want to include version information from your source file components.

Prototype

`REF_INCLUDE{ sectionType [, sectionType]}`

11.4.8 SECTIONS

Defines a new section.

Prototype

`SECTIONS { section_spec }`

where *section_spec* is in the format:

`sectionName : [AT (loadAddress)] {contents} > segmentName`

Parameters

`sectionName`

The name of the section. This name must start with a period character (`.`), followed by a consecutive string of alphanumeric and/or underscore characters (_). For example, `.mysection`.

`loadAddress`

An optional parameter that specifies the address of the section. The linker sets this to the relocation address if you do not specify it.

`contents`

One or more statements that:

- Assign a value to a symbol. See *Section 11.4: Alphabetical keyword listing*, *Section 11.3.2: Arithmetic operations*, and *Section 11.4.1: . (location counter)*.
- Describe the placement of an output section, including which input sections are placed into it. See *Section 11.3.6: File selection* and *Section 11.3.1: Alignment*.

`segmentName`

The name of the memory segment into which you want to put the contents of this section. This option has two variants:

- `> segmentName` — places the section contents at the beginning of the memory segment *segmentName*
- `>> segmentName` — appends the section contents to the memory segment *segmentName*

Example 21 shows an example section definition.

Example 21 Example section definition

```
SECTIONS
{
   .text :
   {
      _textSegmentStart = .;
      foobar.o (.text)
      . = ALIGN (0x10);
      barfoo.o (.text)
      _textSegmentEnd = .;
   }
   .data : { *(.data) }
   .bss  :
   {
      *(.bss)
      *(COMMON)
   }
}
```

11.4.9 WRITEB

Inserts a byte of data at the current address of a section.

Prototype

`WRITEB (expression);`

Parameters

`expression`

A value in the range `0x00` through `0xFF`.

11.4.10 WRITEH

Inserts a half word of data at the current address of a section.

Prototype

```
WRITEH (expression);
```

Parameters

`expression`

A value in the range `0x0000` through `0xFFFF`.

11.4.11 WRITES

Inserts a string at the current address of a section.

Prototype

```
WRITES ("string")
```

Parameters

`string`

A quoted string.

Example 22 shows an example.

Example 22 Example WRITES command

```
.comment_section :
{
   WRITES("This is a .comment section")
} > .comment
```

11.4.12 WRITEW

Inserts a word of data at the current address of a section.

Prototype

```
WRITEW (expression);
```

Parameters

`expression`

A value in the range `0x00000000` through `0xFFFFFFFF`.

11.5 Code and data sections

Because the eTPU has two memory spaces, it is not enough to specify an address for a section. You must also specify a memory space. Specifying a memory space is done in the LCF using the segments access permission flags.

A segment that has the X (executable) flag will be loaded to the *instruction* memory.

A segment that does not have the X (executable) flag will be loaded to the *data* memory.

Note: You can also use the eTPU assembly `.org address` and `.section sec` directives, to specify the memory space for a given section.

Note: You should not assign sections from different memory spaces to the same segment. If you do, the result is undefined.

12 C compiler

This chapter explains the implementation of the C programming language:

- *Extensions to standard C*
- *C99 extensions*
- *GCC extensions*

12.1 Extensions to standard C

The C compiler adds extra features to the C programming language. These extensions make it easier to port source code from other compilers and offer some programming conveniences. Note that some of these extensions do not conform to the ISO/IEC 9899-1990 C standard (“C90”).

- *Controlling standard C conformance*
- *C++-style comments*
- *Unnamed arguments*
- *Extensions to the preprocessor*
- *Non-standard keywords*

12.1.1 Controlling standard C conformance

The compiler offers settings that verify how closely your source code conforms to the ISO/IEC 9899-1990 C standard (“C90”). Enable these settings to check for possible errors or improve source code portability.

Some source code is too difficult or time-consuming to change so that it conforms to the ISO/IEC standard. In this case, disable some or all of these settings.

Table 12 shows how to control the compiler’s features for ISO conformance.

Table 12. Controlling conformance to the ISO/IEC 9899-1990 C language

To control this option from here...	use this setting
Source code	#pragma ANSI_strict #pragma only_std_keywords
Command line	-ansi

12.1.2 C++-style comments

When ANSI strictness is off, the C compiler allows C++-style comments. *Example 23* shows an example.

Example 23 C++ comments

```
a = b;    // This is a C++-style comment.
c = d;     /* This is a regular C-style comment. */
```

12.1.3 Unnamed arguments

When ANSI strictness is off, the C compiler allows unnamed arguments in function definitions. *Example 24* shows an example.

Example 24 Unnamed arguments

```
void f(int) {}  /* OK if ANSI Strict is disabled. */
void f(int i) {} /* Always OK. */
```

12.1.4 Extensions to the preprocessor

When ANSI strictness is off, the C compiler allows a `#` to prefix an item that is not a macro argument. It also allows an identifier after an `#endif` directive. *Example 25* and *Example 26* show examples.

Example 25 Using # in macro definitions

```
#define add1(x) #x #1

    /* OK, if ANSI_strict is disabled,
       but probably not what you wanted:
       add1(abc) creates "abc"#1
     */

#define add2(x) #x "2"

    /* Always OK: add2(abc) creates "abc2". */
```

Example 26 Identifiers after #endif

```
#ifdef __CWCC__
  /* . . . */
#endif __CWCC__ /* OK if ANSI_strict is disabled. */

#ifdef __CWCC__
  /* . . . */
#endif /*__CWCC__*/ /* Always OK. */
```

12.1.5 Non-standard keywords

When the ANSI keywords setting is off, the C compiler recognizes non-standard keywords that extend the language.

12.2 C99 extensions

The eTPU build tools C compiler accepts the enhancements to the C language specified by the ISO/IEC 9899-1999 standard, commonly referred to as “C99.”

- *Controlling C99 extensions*
- *Trailing commas in enumerations*
- *Compound literal values*
- *Designated initializers*
- *Predefined Symbol __func__*
- *Implicit return from main()*
- *Non-constant static data initialization*
- *Variable argument macros*
- *Extra C99 keywords*
- *C++-style comments*
- *C++-style digraphs*
- *Empty arrays in structures*
- *Hexadecimal floating-point constants*
- *Variable-length arrays*
- *Unsuffixed decimal literal values*
- *C99 complex data types*

12.2.1 Controlling C99 extensions

Table 13 shows how to control C99 extensions.

Table 13. Controlling C99 extensions to the C language

To control this option from here...	use this setting
Source code	#pragma c99
Command line	-c99

12.2.2 Trailing commas in enumerations

When the C99 extensions setting is on, the compiler allows a comma after the final item in a list of enumerations. *Example 27* shows an example.

Example 27 Trailing comma in enumeration example

```
enum
{
    violet,
    blue
    green,
    yellow,
    orange,
    red, /* OK: accepted if C99 extensions setting is on. */
};
```

12.2.3 Compound literal values

When the C99 extensions setting is on, the compiler allows literal values of structures and arrays. *Example 28* shows an example.

Example 28 Example of a compound literal

```
#pragma c99 on

struct my_struct {
  int i;
  char c[2];
} my_var;


my_var = ((struct my_struct) {x + y, 'a', 0});
```

12.2.4 Designated initializers

When the C99 extensions setting is on, the compiler allows an extended syntax for specifying which structure or array members to initialize. *Example 29* shows an example.

Example 29 Example of designated initializers

```
#pragma c99 on


struct X {
    int a,b,c;
} x = { .c = 3, .a = 1, 2 };

union U {
    char a;
    long b;
} u = { .b = 1234567 };

int arr1[6] = { 1,2, [4] = 3,4 };

int arr2[6] = { 1, [1 ... 4] = 3,4 }; /* GCC only, not part of C99.
*/
```

12.2.5 Predefined Symbol __func__

When the C99 extensions setting is on, the compiler offers the `__func__` predefined variable. *Example 30* shows an example.

Example 30 Predefined symbol __func__

```
void abc(void)

{
    puts(__func__); /* Output: "abc" */

}
```

12.2.6 Implicit return from main()

When the C99 extensions setting is on, the compiler inserts this statement at the end of a program's main() function if the function does not return a value:

```
return 0;
```

12.2.7 Non-constant static data initialization

When the C99 extensions setting is on, the compiler allows static variables to be initialized with non-constant expressions.

12.2.8 Variable argument macros

When the C99 extensions setting is on, the compiler allows macros to have a variable number of arguments. *Example 31* shows an example.

Example 31 Variable argument macros example

```
#define MYLOG(...) fprintf(myfile, __VA_ARGS__)

#define MYVERSION 1
#define MYNAME "SockSorter"

int main(void)
{
    MYLOG("%d %s\n", MYVERSION, MYNAME);
    /* Expands to: fprintf(myfile, "%d %s\n", 1, "SockSorter"); */

    return 0;
}
```

12.2.9 Extra C99 keywords

When the C99 extensions setting is on, the compiler recognizes extra keywords and the language features they represent. *Table 14* lists these keywords.

Table 14. Extra C99 keywords

This keyword or combination of keywords...	represents this language feature
_Bool	boolean data type
long long	integer data type
restrict	type qualifier
inline	function qualifier
_Complex	complex number data type
_Imaginary	imaginary number data type

12.2.10 C++-style comments

When the C99 extensions setting is on, the compiler allows C++-style comments as well as regular C comments. A C++-style comment begins with

```
//
```

and continues until the end of a source code line.

A C-style comment begins with

```
/*
```

ends with

```
*/
```

and may span more than one line.

12.2.11 C++-style digraphs

When the C99 extensions setting is on, the compiler recognizes C++-style two-character combinations that represent single-character punctuation. *Table 15* lists these digraphs.

Table 15. C++-style digraphs

This digraph	is equivalent to this character
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

12.2.12 Empty arrays in structures

When the C99 extensions setting is on, the compiler allows an empty array to be the last member in a structure definition. *Example 32* shows an example.

Example 32 Example of an empty array as the last struct member

```
struct {
  int r;
  char arr[];
} s;
```

12.2.13 Hexadecimal floating-point constants

Precise representations of constants specified in hexadecimal notation to ensure an accurate constant is generated across compilers and on different hosts. The compiler generates a warning message when the mantissa is more precise than the host floating point format. The compiler generates an error message if the exponent is too wide for the host float format.

Examples:

```
0x2f.3a2p3
```

```
0xEp1f
```

```
0x1.8p0L
```

The standard library supports printing values of type `float` in this format using the "`%a`" and "`%A`" specifiers.

12.2.14 Variable-length arrays

Variable length arrays are supported within local or function prototype scope, as required by the ISO/IEC 9899-1999 ("C99") standard. *Example 33* shows an example.

Example 33 Example of C99 variable length array usage

```
#pragma c99 on


void f(int n) {
   int arr[n];
   /* ... */
}
```

While the example shown in *Example 34* generates an error message.

Example 34 Bad example of C99 variable length array usage

```
#pragma c99 on

int n;
int arr[n];
// ERROR: variable length array
// types can only be used in local or

// function prototype scope.
```

A variable length array cannot be used in a function template's prototype scope or in a local template `typedef`, as shown in *Example 35*.

Example 35 Bad example of C99 usage in function prototype

```
#pragma c99 on

template<typename T> int f(int n, int A[n][n]);
{
};
// ERROR: variable length arrays
// cannot be used in function template prototypes

// or local template variables
```

12.2.15 Unsuffixed decimal literal values

Example 36 shows an example of specifying decimal literal values without a suffix to specify the literal's type.

Example 36 Examples of C99 unsuffixed constants

```
#pragma c99 on  // Note: ULONG_MAX == 4294967295
```

```
sizeof(4294967295)  == sizeof(long long)
sizeof(4294967295u) == sizeof(unsigned long)

#pragma c99 off

sizeof(4294967295)  == sizeof(unsigned long)

sizeof(4294967295u) == sizeof(unsigned long)
```

12.2.16 C99 complex data types

The compiler supports the C99 complex and imaginary data types when the `C99 extensions` option is enabled. *Example 37* shows an example.

Example 37 C99 complex data type

```
#include <complex.h>

complex double cd = 1 + 2*I;
```

Note: *This feature is currently not available for all targets.*
Use `#if __has_feature(C99_COMPLEX)` to check if this feature is available for your target.

12.3 GCC extensions

The compiler accepts many of the extensions to the C language that the GCC (Gnu Compiler Collection) tools allow. Source code that uses these extensions does not conform to the ISO/IEC 9899-1990 C (“C90”) standard.

- *Controlling GCC extensions*
- *Initializing automatic arrays and structures*
- *The sizeof() operator*
- *Statements in expressions*
- *Redefining macros*
- *The typeof() operator*
- *Void and function pointer arithmetic*
- *The __builtin_constant_p() operator*
- *Forward declarations of static arrays*
- *Omitted operands in conditional expressions*
- *The __builtin_expect() operator*
- *Void return statements*
- *Minimum and maximum operators*
- *Local labels*

12.3.1 Controlling GCC extensions

Table 16 shows how to turn GCC extensions on or off.

Table 16. Controlling GCC extensions to the C language

To control this option from here...	use this setting
Source code	#pragma gcc_extensions
Command line	-gcc_extensions

12.3.2 Initializing automatic arrays and structures

When the GCC extensions setting is on, array and structure variables that are local to a function and have the automatic storage class may be initialized with values that do not need to be constant. *Example 38* shows an example.

Example 38 Initializing arrays and structures with non-constant values

```
void f(int i)
{
    int j = i * 10; /* Always OK. */
    /* These initializations are only accepted when GCC extensions
     * are on. */
    struct { int x, y; } s = { i + 1, i + 2 };
    int a[2] = { i, i + 2 };
}
```

12.3.3 The sizeof() operator

When the GCC extensions setting is on, the `sizeof()` operator computes the size of function and void types. In both cases, the `sizeof()` operator evaluates to 1. The ISO/IEC 9899-1990 C Standard ("C90") does not specify the size of the `void` type and functions. *Example 39* shows an example.

Example 39 Using the sizeof() operator with void and function types

```
int f(int a)
{
    return a * 10;
}

void g(void)
{
    size_t voidsize = sizeof(void); /* voidsize contains 1 */
    size_t funcsize = sizeof(f); /* funcsize contains 1 */
}
```

12.3.4 Statements in expressions

When the GCC extensions setting is on, expressions in function bodies may contain statements and definitions. To use a statement or declaration in an expression, enclose it within braces. The last item in the brace-enclosed expression gives the expression its value. *Example 40* shows an example.

Example 40 Using statements and definitions in expressions

```
#define POW2(n) ({ int i,r; for(r=1,i=n; i>0; --i) r *= 2; r;})


int main()
{
    return POW2(4);
}
```

12.3.5 Redefining macros

When the GCC extensions setting is on, macros may be redefined with the `#define` directive without first undefining them with the `#undef` directive.*Example 41* shows an example.

Example 41 Redefining a macro without undefining first

```
#define SOCK_MAXCOLOR 100

#undef SOCK_MAXCOLOR
#define SOCK_MAXCOLOR 200 /* OK: this macro is previously undefined.
*/


#define SOCK_MAXCOLOR 300
```

12.3.6 The typeof() operator

When the GCC extensions setting is on, the compiler recognizes the `typeof()` operator. This compile-time operator returns the type of an expression. You may use the value returned by this operator in any statement or expression where the compiler expects you to specify a type. The compiler evaluates this operator at compile time. The `__typeof()__` operator is the same as this operator.*Example 42* shows an example.

Example 42 Using the typeof() operator

```
int *ip;

/* Variables iptr and jptr have the same type. */
typeof(ip) iptr;
int *jptr;

/* Variables i and j have the same type. */
typeof(*ip) i;

int j;
```

12.3.7 Void and function pointer arithmetic

The ISO/IEC 9899-1990 C Standard does not accept arithmetic expressions that use pointers to `void` or functions. With GCC extensions on, the compiler accepts arithmetic manipulation of pointers to `void` and functions.

12.3.8 The __builtin_constant_p() operator

When the GCC extensions setting is on, the compiler recognizes the `__builtin_constant_p()` operator. This compile-time operator takes a single argument and returns 1 if the argument is a constant expression or 0 if it is not.

12.3.9 Forward declarations of static arrays

When the GCC extensions setting is on, the compiler will not issue an error when you declare a static array without specifying the number of elements in the array if you later declare the array completely.*Example 43* shows an example.

Example 43 Forward declaration of an empty array

```
static int a[]; /* Allowed only when GCC extensions are on. */
/* ... */
static int a[10]; /* Complete declaration. */
```

12.3.10 Omitted operands in conditional expressions

When the GCC extensions setting is on, you may skip the second expression in a conditional expression. The default value for this expression is the first expression.*Example 44* shows an example.

Example 44 Using the shorter form of the conditional expression

```
void f(int i, int j)
{
    int a = i ? i : j;
    int b = i ?: j; /* Equivalent to int b = i ? i : j; */
    /* Variables a and b are both assigned the same value. */
}
```

12.3.11 The __builtin_expect() operator

When the GCC extensions setting is on, the compiler recognizes the `__builtin_expect()` operator. Use this compile-time operator in an `if` or `while` statement to specify to the compiler how to generate instructions for branch prediction.

This compile-time operator takes two arguments:

- The first argument must be an integral expression
- The second argument must be a literal value

The second argument is the most likely result of the first argument.*Example 45* shows an example.

Example 45 Example for __builtin_expect() operator

```
void search(int *array, int size, int key)
{
    int i;
```

```
        for (i = 0; i < size; ++i)
        {
              /* We expect to find the key rarely. */
              if (__builtin_expect(array[i] == key, 0))
             {
                 rescue(i);
             }
        }
    }
```

12.3.12 Void return statements

When the GCC extensions setting is on, the compiler allows you to place expressions of type `void` in a `return` statement.*Example 46* shows an example.

Example 46 Returning void

```
void f(int a)
{
    /* ... */
    return; /* Always OK. */
}

void g(int b)
{
    /* ... */
    return f(b); /* Allowed when GCC extensions are on. */
}
```

12.3.13 Minimum and maximum operators

When the GCC extensions setting is on, the compiler recognizes built-in minimum (`<?`) and maximum (`>?`) operators.

Example 47 Example of minimum and maximum operators

```
int a = 1 <? 2; // 1 is assigned to a.
int b = 1 >? 2; // 2 is assigned to b.
```

12.3.14 Local labels

When the GCC extensions setting is on, the compiler allows labels limited to a block's scope. A label declared with the `__label__` keyword is visible only within the scope of its enclosing block.*Example 48* shows an example.

Example 48 Example of using local labels

```
void f(int i)
{
  if (i >= 0)
  {
```

```
    __label__ again; /* First again. */
    if (--i > 0)
      goto again; /* Jumps to first again. */
  }
  else
  {
    __label__ again; /* Second again. */
    if (++i < 0)
      goto again; /* Jumps to second again. */
  }
}
```

13 Intermediate optimizations

After it translates a program's source code into its intermediate representation, the compiler optionally applies optimizations that reduce the program's size, improve its execution speed, or both. The topics in this chapter explains these optimizations and how to apply them:

- *Intermediate optimizations*
- *Inlining*

13.1 Intermediate optimizations

After it translates a function into its intermediate representation, the compiler may optionally apply some optimizations. The result of these optimizations on the intermediate representation will either reduce the size of the executable code, improve the executable code's execution speed, or both.

- *Dead code elimination*
- *Expression simplification*
- *Common subexpression elimination*
- *Copy propagation*
- *Dead store elimination*
- *Live range splitting*
- *Loop-invariant code motion*
- *Strength reduction*
- *Loop unrolling*

13.1.1 Dead code elimination

The dead code elimination optimization removes expressions that are not accessible or are not referred to. This optimization reduces size and increases execution speed.

Table 17 explains how to control the optimization for dead code elimination.

Table 17. Controlling dead code elimination

Turn control this option from here...	use this setting
Source code	#pragma opt_dead_code on | off | reset
Command line	-opt [no]deadcode

In *Example 49*, the call to `func1()` will never execute because the `if` statement that it is associated with will never be true. Consequently, the compiler can safely eliminate the call to `func1()`, as shown in *Example 50*.

Example 49 Before dead code elimination

```
void func_from(void)
{
    if (0)
    {
```

```
        func1();
    }
    func2();
}
```

Example 50 After dead code elimination

```
void func_to(void)
{
    func2();
}
```

13.1.2 Expression simplification

The expression simplification optimization attempts to replace arithmetic expressions with simpler expressions. Additionally, the compiler also looks for operations in expressions that can be avoided completely without affecting the final outcome of the expression. This optimization reduces size and increases speed.

Table 18 explains how to control the optimization for expression simplification.

Table 18. Controlling expression simplification

Turn control this option from here...	use this setting
Source code	There is no pragma to control this optimization.
Command line	`-opt level=1`, `-opt level=2`, `-opt level=3`, `-opt level=4`

For example, *Example 51* contains a few assignments to some arithmetic expressions:

- Addition to zero
- Multiplication by a power of 2
- Subtraction of a value from itself
- Arithmetic expression with two or more literal values

Example 51 Before expression simplification

```
void func_from(int* result1, int* result2, int* result3, int*
result4, int x)
{
    *result1 = x + 0;
    *result2 = x * 2;
    *result3 = x - x;
    *result4 = 1 + x + 4;
}
```

Example 52 shows source code that is equivalent to expression simplification. The compiler has modified these assignments to:

- Remove the addition to zero
- Replace the multiplication of a power of 2 with bit-shift operation
- Replace a subtraction of x from itself with 0
- Consolidate the additions of 1 and 4 into 5

Example 52 After expression simplification

```
void func_to(int* result1, int* result2, int* result3, int* result4,
int x){
    *result1 = x;
    *result2 = x << 1;
    *result3 = 0;
    *result4 = 5 + x;
}
```

13.1.3 Common subexpression elimination

Common subexpression elimination replaces multiple instances of the same expression with a single instance. This optimization reduces size and increases execution speed.

Table 19 explains how to control the optimization for common subexpression elimination.

Table 19. Controlling common subexpression elimination

Turn control this option from here...	use this setting
Source code	#pragma opt_common_subs on | off | reset
Command line	-opt [no]cse

For example, in *Example 53*, the subexpression `x * y` occurs twice.

Example 53 Before common subexepression elimination

```
void func_from(int* vec, int size, int x, int y, int value)
{
    if (x * y < size)
    {
        vec[x * y - 1] = value;
    }
}
```

Example 54 shows equivalent source code after the compiler applies common subexpression elimination. The compiler generates instructions to compute `x * y` and store it in a hidden, temporary variable. The compiler then replaces each instance of the subexpression with this variable.

Example 54 After common subexpression elimination

```
void func_to(int* vec, int size, int x, int y, int value)
{
    int temp = x * y;
    if (temp < size)
    {
```

```
            vec[temp - 1] = value;
        }
    }
```

13.1.4 Copy propagation

Copy propagation replaces variables with their original values if the variables do not change. This optimization reduces runtime stack size and improves execution speed.

Table 20 explains how to control the optimization for copy propagation.

Table 20. Controlling copy propagation

Turn control this option from here...	use this setting
Source code	#pragma opt_propagation on | off | reset
Command line	-opt [no]prop[agation]

For example, in *Example 55* , the variable `j` is assigned the value of `x`. But `j`'s value is never changed, so the compiler replaces later instances of `j` with `x`, as shown in *Example 56*.

By propagating `x`, the compiler is able to reduce the number of registers it uses to hold variable values, allowing more variables to be stored in registers instead of slower memory. Also, this optimization reduces the amount of stack memory used during function calls.

Example 55 Before copy propagation

```
void func_from(int* a, int x)
{
    int i;
    int j;
    j = x;
    for (i = 0; i < j; i++)
    {
        a[i] = j;
    }
}
```

Example 56 After copy propagation

```
void func_to(int* a, int x)
{
    int i;
    int j;
    j = x;
    for (i = 0; i < x; i++)
    {
        a[i] = x;
    }
}
```

13.1.5 Dead store elimination

Dead store elimination removes unused assignment statements. This optimization reduces size and improves speed.

Table 21 explains how to control the optimization for dead store elimination.

Table 21. Controlling dead store elimination

Turn control this option from here...	use this setting
source code	#pragma opt_dead_assignments on | off | reset
command line	-opt [no]deadstore

For example, in *Example 57* the variable `x` is first assigned the value of `y * y`. However, this result is not used before `x` is assigned the result returned by a call to `getresult()`.

In *Example 58* the compiler can safely remove the first assignment to `x` since the result of this assignment is never used.

Example 57 Before dead store elimination

```
void func_from(int x, int y)
{
    x = y * y;
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

Example 58 After dead store elimination

```
void func_to(int x, int y)
{
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

13.1.6 Live range splitting

Live range splitting attempts to reduce the number of variables used in a function. This optimization reduces a function's runtime stack size, requiring fewer instructions to invoke the function. This optimization potentially improves execution speed.

Table 22 explains how to control the optimization for live range splitting.

Table 22. Controlling live range splitting

Turn control this option from here...	use this setting
Source code	There is no pragma to control this optimization.
Command line	-opt level=3, -opt level=4

For example, in *Example 59* three variables, a, b, and c, are defined. Although each variable is eventually used, each of their uses is exclusive to the others. In other words, a is not referred to in the same expressions as b or c, b is not referred to with a or c, and c is not used with a or b.

In *Example 60*, the compiler has replaced a, b, and c, with a single variable. This optimization reduces the number of registers that the object code uses to store variables, allowing more variables to be stored in registers instead of slower memory. This optimization also reduces a function's stack memory.

Example 59 Before live range splitting

```
void func_from(int x, int y)
{
    int a;
    int b;
    int c;

    a = x * y;
    otherfunc(a);

    b = x + y;
    otherfunc(b);

    c = x - y;
    otherfunc(c);
}
```

Example 60 After live range splitting

```
void func_to(int x, int y)
{
    int a_b_or_c;

    a_b_or_c = x * y;
    otherfunc(temp);

    a_b_or_c = x + y;
    otherfunc(temp);

    a_b_or_c = x - y;
    otherfunc(temp);
}
```

13.1.7 Loop-invariant code motion

Loop-invariant code motion moves expressions out of a loop if the expressions are not affected by the loop or the loop does not affect the expression. This optimization improves execution speed.

Table 23 explains how to control the optimization for loop-invariant code motion.

Table 23. Controlling loop-invariant code motion

Turn control this option from here...	use this setting
Source code	#pragma opt_loop_invariants on | off | reset
Command line	-opt [no]loop[invariants]

For example, in *Example 61*, the assignment to the variable `circ` does not refer to the counter variable of the `for` loop, `i`. But the assignment to `circ` will be executed at each loop iteration.

Example 62 shows source code that is equivalent to how the compiler would rearrange instructions after applying this optimization. The compiler has moved the assignment to `circ` outside the `for` loop so that it is only executed once instead of each time the `for` loop iterates.

Example 61 Before loop-invariant code motion

```
void func_from(float* vec, int max, float val)
{
    float circ;
    int i;
    for (i = 0; i < max; ++i)
    {
        circ = val * 2 * PI;
        vec[i] = circ;
    }
}
```

Example 62 After loop-invariant code motion

```
void func_to(float* vec, int max, float val)
{
    float circ;
    int i;
    circ = val * 2 * PI;
    for (i = 0; i < max; ++i)
    {
        vec[i] = circ;
    }
}
```

13.1.8 Strength reduction

Strength reduction attempts to replace slower multiplication operations with faster addition operations. This optimization improves execution speed but increases code size.

Table 24 explains how to control the optimization for strength reduction.

Table 24. Controlling strength reduction

Turn control this option from here...	use this setting
Source code	#pragma opt_strength_reduction on | off | reset
Command line	-opt [no]strength

For example, in *Example 63*, the assignment to elements of the `vec` array use a multiplication operation that refers to the `for` loop's counter variable, `i`.

In *Example 64*, the compiler has replaced the multiplication operation with a hidden variable that is increased by an equivalent addition operation. Processors execute addition operations faster than multiplication operations.

Example 63 Before strength reduction

```
void func_from(int* vec, int max, int fac)
{
    int i;
    for (i = 0; i < max; ++i)
    {
        vec[i] = fac * i;
    }
}
```

Example 64 After strength reduction

```
void func_to(int* vec, int max, int fac)
{
    int i;
    int strength_red;
    hidden_strength_red = 0;
    for (i = 0; i < max; ++i)
    {
        vec[i] = hidden_strength_red;
        hidden_strength_red = hidden_strength_red + i;
    }
}
```

13.1.9 Loop unrolling

Loop unrolling inserts extra copies of a loop's body in a loop to reduce processor time executing a loop's overhead instructions for each iteration of the loop body. In other words, this optimization attempts to reduce the ratio of time that the processor executes a loop's completion test and branching instructions compared to the time the processor executes the loop's body. This optimization improves execution speed but increases code size.

Table 25 explains how to control the optimization for loop unrolling.

Table 25. Controlling loop unrolling

Turn control this option from here...	use this setting
Source code	#pragma opt_unroll_loops on | off | reset
Command line	-opt level=3, -opt level=4

For example, in *Example 65*, the `for` loop's body is a single call to a function, `otherfunc()`. For each time the loop's completion test executes

```
for (i = 0; i < MAX; ++i)
```

the function executes the loop body only once.

In *Example 66*, the compiler has inserted another copy of the loop body and rearranged the loop to ensure that variable `i` is incremented properly. With this arrangement, the loop's completion test executes once for every 2 times that the loop body executes.

Example 65 Before loop unrolling

```
const int MAX = 100;
void func_from(int* vec)
{
    int i;
    for (i = 0; i < MAX; ++i)
    {
        otherfunc(vec[i]);
    }
}
```

Example 66 After loop unrolling

```
const int MAX = 100;
void func_to(int* vec)
{
    int i;
    for (i = 0; i < MAX;)
    {
        otherfunc(vec[i]);
        ++i;
        otherfunc(vec[i]);
        ++i;
    }
}
```

13.2 Inlining

Inlining replaces instructions that call a function and return from it with the actual instructions of the function being called. Inlining functions makes your program faster because it executes the function code immediately without the overhead of a function call and return. However, inlining can also make your program larger because the compiler may insert the function's instructions many times throughout your program.

The rest of this section explains how to specify which functions to inline and how the compiler performs the inlining:

- *Choosing which functions to inline*
- *Inlining techniques*

13.2.1 Choosing which functions to inline

The compiler offers several methods to specify which functions are eligible for inlining.

To specify that a function is eligible to be inlined, precede its definition with the `inline`, `__inline__`, or `__inline` keyword. To allow these keywords in C source code turn off the `only_std_keywords` pragma in your source code.

To verify that an eligible function has been inlined or not, use the Non-Inlined Functions option in the IDE's C/C++ Warnings panel or the `warn_notinlined` pragma. *Example 67* shows an example.

Example 67 Specifying to the compiler that a function may be inlined

```
#pragma only_std_keywords off
inline int attempt_to_inline(void)
{
    return 10;
}
```

To specify that a function must never be inlined, follow its definition's specifier with `__attribute__((never_inline))`.*Example 68* shows an example.

Example 68 Specifying to the compiler that a function must never be inlined

```
int never_inline(void) __attribute__((never_inline))
{
   return 20;
}
```

To specify that no functions in a file may be inlined, including those that are defined with the `inline`, `__inline__`, or `__inline` keywords, use the `dont_inline` pragma.*Example 69* shows an example.

Example 69 Specifying that no functions may be inlined

```
#pragma dont_inline on

/* Will not be inlined. */
inline int attempt_to_inline(void)
{
    return 10;
}

/* Will not be inlined. */
int never_inline(void) __attribute__((never_inline))
{
   return 20;
}
```

```
#pragma dont_inline off
/* Will be inlined, if possible. */
inline int also_attempt_to_inline(void)
{
    return 10;
}
```

Some kinds of functions are never inlined:

- Functions with variable argument lists
- Functions defined with `__attribute__((never_inline))`
- Functions compiled with `#pragma optimize_for_size on` or the Optimize For Size setting in the IDE's Global Optimizations panel
- Functions which have their addresses stored in variables

The compiler will not inline these functions, even if they are defined with the `inline`, `__inline__`, or `__inline` keywords.

13.2.2 Inlining techniques

The depth of inlining explains how many levels of function calls the compiler will inline. The Inline Depth setting in the IDE's C/C++ Language settings panel and the `inline_depth` pragma control inlining depth.

Normally, the compiler only inlines an eligible function if it has already translated the function's definition. In other words, if an eligible function has not yet been compiled, the compiler has no object code to insert. To overcome this limitation, the compiler can perform interprocedural analysis (IPA) either in file or program mode. This lets the compiler evaluate all the functions in a file or even the entire program before inlining the code. The IPA setting in the IDE's C/C++ Language settings panel and the `ipa` pragma control this capability.

The compiler normally inlines functions from the first function in a chain of function calls to the last function called. Alternately, the compiler may inline functions from the last function called to the first function in a chain of function calls. The Bottom-up Inlining option in the IDE's C/C++ Language settings panel and the `inline_bottom_up` and `inline_bottom_up_once` pragmas control this reverse method of inlining.

Some functions that have not been defined with the `inline`, `__inline__`, or `__inline` keywords may still be good candidates to be inlined. Automatic inlining allows the compiler to inline these functions in addition to the functions that you explicitly specify as eligible for inlining. The Auto-Inline option in the IDE's C/C++ Language panel and the `auto_inline` pragma control this capability.

When inlining, the compiler calculates the complexity of a function by counting the number of statements, operands, and operations in a function to determine whether or not to inline an eligible function. The compiler does not inline functions that exceed a maximum complexity. The compiler uses three settings to control the extent of inlined functions:

- Maximum auto-inlining complexity: the threshold for which a function may be auto-inlined
- Maximum complexity: the threshold for which any eligible function may be inlined
- Maximum total complexity: the threshold for all inlining in a function

The `inline_max_auto_size`, `inline_max_size`, and `inline_max_total_size` pragmas control these thresholds, respectively.

14 Declaration specifications

Declaration specifications describe special properties to associate with a function or variable at compile time. You insert these specifications in the object's declaration.

- *Syntax for declaration specifications*
- *__declspec(never_inline)*

14.1 Syntax for declaration specifications

The syntax for a declaration specification is

```
__declspec(spec [ options ]) function-declaration;
```

where *spec* is the declaration specification, *options* represents possible arguments for the declaration specification, and *function-declaration* represents the declaration of the function. Unless otherwise specified in the declaration specification's description, a function's definition does not require a declaration specification.

14.2 __declspec(never_inline)

Specifies that a function must not be inlined.

Syntax

```
__declspec (never_inline) function_prototype;
```

Remarks

Declaring a function's prototype with this declaration specification tells the compiler not to inline the function, even if the function is later defined with the `inline`, `__inline__`, or `__inline` keywords.

14.3 Syntax for attribute specifications

The syntax for an attribute specification is

```
__attribute__((list-of-attributes))
```

where *list-of-attributes* is a comma-separated list of zero or more attributes to associate with the object. Place an attribute specification at the end of the delcaration and definition of a function, function parameter, or variable. *Example 70* shows an example.

Example 70 Example of an attribute specification

```
int f(int x __attribute__((unused))) __attribute__((never_inline));

int f(int x __attribute__((unused))) __attribute__((never_inline))
{
    return 20;
}
```

14.4 Attribute specifications

14.4.1 __attribute__((deprecated))

Specifies that the compiler must issue a warning when a program refers to an object.

Syntax

```
variable-declaration __attribute__((deprecated));

variable-definition __attribute__((deprecated));

function-declaration __attribute__((deprecated));

function-definition __attribute__((deprecated));
```

Remarks

This attribute instructs the compiler to issue a warning when a program refers to a function or variable. Use this attribute to discourage programmers from using functions and variables that are obsolete or will soon be obsolete.

Example 71 Example of deprecated attribute

```
int velocipede(int speed) __attribute__((deprecated));
int bicycle(int speed);

int f(int speed)
{
  return velocipede(speed); /* Warning. */
}

int g(int speed)
{
  return bicycle(speed * 2); /* OK */
}
```

14.4.2 __attribute__((force_export))

Prevents a function or static variable from being dead-stripped.

Syntax

```
function-declaration __attribute__((force_export));

function-definition __attribute__((force_export));

variable-declaration __attribute__((force_export));

variable-definition __attribute__((force_export));
```

Remarks

This attribute specifies that the linker must not dead-strip a function or static variable even if the linker determines that the rest of the program does not refer to the object.

14.4.3 __attribute__((malloc))

Specifies that the pointers returned by a function will not point to objects that are already referred to by other variables.

Syntax

```
function-declaration __attribute__((malloc));
function-definition __attribute__((malloc));
```

Remarks

This attribute specification gives the compiler extra knowledge about pointer aliasing so that it can apply stronger optimizations to the object code it generates.

14.4.4 __attribute__((noalias))

Prevents access of data object through an indirect pointer access.

Syntax

```
function-parameter __attribute__((noalias));
variable-declaration __attribute__((noalias));
variable-definition __attribute__((noalias));
```

Remarks

This attribute specifies to the compiler that a data object is only accessed directly, helping the optimizer to generate a better code. The sample code in *Example 72* will not return a correct result if `ip` is pointed to `a`.

Example 72 Example of the noalias attribute

```
extern int a __attribute__((noalias));
int f(int *ip)
{
  a = 1;
  *ip = 0;
  return a;   // optimized to return 1;
}
```

14.4.5 __attribute__((returns_twice))

Specifies that a function may return more than one time because of multithreaded or non-linear execution.

Syntax

```
function-declaration __attribute__((returns_twice));
function-definition __attribute__((returns_twice));
```

Remarks

This attribute specifies to the compiler that the program's flow of execution might enter and leave a function without explicit function calls and returns. For example, the standard library's `setjmp()` function allows a program to change its execution flow arbitrarily.

With this information, the compiler limits optimizations that require explicit program flow.

14.4.6 __attribute__((unused))

Specifies that the programmer is aware that a variable or function parameter is not referred to.

Syntax

```
function-parameter __attribute__((unused));

variable-declaration __attribute__((unused));

variable-definition __attribute__((unused));
```

Remarks

This attribute specifies that the compiler should not issue a warning for an object if the object is not referred to. This attribute specification has no effect if the compiler's unused warning setting is off.

Example 73 Example of the unused attribute

```
void f(int a __attribute__((unused))) /* No warning for a. */
{
  int b __attribute__((unused)); /* No warning for b. */
  int c; /* Possible warning for c. */

  return 20;
}
```

14.4.7 __attribute__((used))

Prevents a function or static variable from being dead-stripped.

Syntax

```
function-declaration __attribute__((used));

function-definition __attribute__((used));

variable-declaration __attribute__((used));

variable-definition __attribute__((used));
```

Remarks

This attribute specifies that the linker must not dead-strip a function or static variable even if the linker determines that the rest of the program does not refer to the object.

15 Predefined macros

The compiler preprocessor has predefined macros (some refer to these as predefined symbols). The compiler simulates variable definitions that describe the compile-time environment and properties of the target processor.

This chapter lists the predefined macros that all eTPU build tools compilers make available.

- *__COUNTER__*
- *__cplusplus*
- *__CWCC__*
- *__embedded_cplusplus*
- *__FILE__*
- *__func__*
- *__FUNCTION__*
- *__ide_target()*
- *__LINE__*
- *__MWERKS__*
- *__PRETTY_FUNCTION__*
- *__profile__*
- *__STDC__*
- *__TIME__*

15.1 __COUNTER__

Preprocessor macro that expands to an integer.

Syntax

```
__COUNTER__
```

Remarks

The compiler defines this macro as an integer that has an initial value of 0 incrementing by 1 every time the macro is used in the translation unit.

The value of this macro is stored in a precompiled header and is restored when the precompiled header is used by a translation unit.

15.2 __cplusplus

Preprocessor macro defined if compiling C++ source code.

Syntax

```
__cplusplus
```

Remarks

The compiler defines this macro when compiling C++ source code. This macro is undefined otherwise.

15.3 __CWCC__

Preprocessor macro defined as the version of the compiler frontend.

Syntax

`__CWCC__`

Remarks

Compilers issued after 2006 define this macro with the compiler's frontend version. For example, if the compiler frontend version is 4.2.0, the value of `__CWCC__` is `0x4200`.

Compilers issued prior to 2006 used the pre-defined macro `__MWERKS__`. The `__MWERKS__` predefined macro is still functional as an alias for `__CWCC__`.

The ISO standards do not specify this symbol.

15.4 __DATE__

Preprocessor macro defined as the date of compilation.

Syntax

`__DATE__`

Remarks

The compiler defines this macro as a character string representation of the date of compilation. The format of this string is

"*Mmm dd yyyy*"

where *Mmm* is the a three-letter abbreviation of the month, *dd* is the day of the month, and *yyyy* is the year.

15.5 __embedded_cplusplus

Defined as 1 when compiling embedded C++ source code, undefined otherwise.

Syntax

`__embedded_cplusplus`

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the Embedded C++ proposed standard. The compiler does not define this macro otherwise.

15.6 __FILE__

Preprocessor macro of the name of the source code file being compiled.

Syntax

```
__FILE__
```

Remarks

The compiler defines this macro as a character string literal value of the name of the file being compiled, or the name specified in the last instance of a `#line` directive.

15.7 __func__

Predefined variable of the name of the function being compiled.

Prototype

```
static const char __func__[] = "function-name";
```

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__func__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

15.8 __FUNCTION__

Predefined variable of the name of the function being compiled.

Prototype

```
static const char __FUNCTION__[] = "function-name";
```

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__FUNCTION__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body.

15.9 __ide_target()

Preprocessor operator for querying the IDE about the active build target.

Syntax

```
__ide_target("target_name")
```

`target-name`

The name of a build target in the active project in an integrated development environment (IDE).

Remarks

Expands to `1` if *`target_name`* is the same as the active build target in the IDE's active project. Expands to `0` otherwise. The ISO standards do not specify this symbol.

15.10 __LINE__

Preprocessor macro of the number of the line of the source code file being compiled.

Syntax

```
__LINE__
```

Remarks

The compiler defines this macro as a integer value of the number of the line of the source code file that the compiler is translating. The `#line` directive also affects the value that this macro expands to.

15.11 __MWERKS__

Deprecated. Preprocessor macro defined as the version of the compiler.

Syntax

```
__MWERKS__
```

Remarks

Replaced by the built-in preprocessor macro `__CWCC__`.

Compilers issued after 1995 define this macro with the compiler's version. For example, if the compiler version is 4.0, the value of `__MWERKS__` is `0x4000`.

This macro is defined as `1` if the compiler was issued before the 1995.

The ISO standards do not specify this symbol.

15.12 __PRETTY_FUNCTION__

Predefined variable containing a character string of the "unmangled" name of the C++ function being compiled.

Syntax

```
__PRETTY_FUNCTION__
```

Prototype

```
static const char __PRETTY_FUNCTION__[] = "function-name";
```

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__PRETTY_FUNCTION__`. This name, *function-name*, is the same identifier that appears in source code, not the “mangled” identifier that the compiler and linker use. The C++ compiler “mangles” a function name by appending extra characters to the function’s identifier to denote the function’s return type and the types of its parameters.

The ISO/IEC 14882-1998 C++ standard does not specify this symbol.

15.13 __profile__

Preprocessor macro that specifies whether or not the compiler is generating object code for a profiler.

Syntax

```
__profile__
```

Remarks

Defined as 1 when generating object code that works with a profiler. Undefined otherwise. The ISO standards does not specify this symbol.

15.14 __STDC__

Defined as 1 when compiling ISO/IEC Standard C source code, undefined otherwise.

Syntax

```
__STDC__
```

Remarks

The compiler defines this macro as 1 when the compiler’s settings are configured to restrict the compiler to translate source code that conforms to the ISO/IEC 9899-1990 and ISO/IEC 9899-1999 standards. The compiler does not define this macro otherwise.

15.15 __TIME__

Preprocessor macro defined as a character string representation of the time of compilation.

Syntax

```
__TIME__
```

Remarks

The compiler defines this macro as a character string representation of the time of compilation. The format of this string is

```
"hh:mm:ss"
```

where *hh* is a 2-digit hour of the day, *mm* is a 2-digit minute of the hour, and *ss* is a 2-digit second of the minute.

16 Using pragmas

The #pragma preprocessor directive specifies option settings to the compiler to control the compiler and linker's code generation.

- *Checking pragma settings*
- *Saving and restoring pragma settings*
- *Determining which settings are saved and restored*
- *Invalid pragmas*

16.1 Checking pragma settings

The preprocessor function `__option()` returns the state of pragma settings at compile-time. The syntax is

`__option(`*`setting-name`*`)`

where *setting-name* is the name of a pragma that accepts the `on`, `off`, and `reset` arguments.

If *setting-name* is `on`, `__option(`*`setting-name`*`)` returns 1. If *setting-name* is `off`, `__option(`*`setting-name`*`)` returns 0. If *setting-name* is not the name of a pragma, `__option(`*`setting-name`*`)` returns false. If *setting-name* is the name of a pragma that does not accept the `on`, `off`, and `reset` arguments, the compiler issues a warning message.

Example 74 shows an example.

Example 74 Using the __option() preprocessor function

```
#if __option(ANSI_strict)
#include "portable.h" /* Use the portable declarations. */
#else
#include "custom.h" /* Use the specialized declarations. */
#endif
```

16.2 Saving and restoring pragma settings

There are some occasions when you would like to apply pragma settings to a piece of source code independently from the settings in the rest of the source file. For example, a function might require unique optimization settings that should not be used in the rest of the function's source file.

Remembering which pragmas to save and restore is tedious and error-prone. Fortunately, the compiler has mechanisms that save and restore pragma settings at compile time. Pragma settings may be saved and restored at two levels:

- All pragma settings
- Some individual pragma settings

Settings may be saved at one point in a compilation unit (a source code file and the files that it includes), changed, then restored later in the same compilation unit. Pragma settings cannot be saved in one source code file then restored in another unless both source code files are included in the same compilation unit.

Pragmas `push` and `pop` save and restore, respectively, most pragma settings in a compilation unit. Pragmas `push` and `pop` may be nested to unlimited depth.*Example 75* shows an example.

Example 75 Using push and pop to save and restore pragma settings

```
/* Settings for this file. */
#pragma opt_unroll_loops on
#pragma optimize_for_size off
void fast_func_A(void)
{
/* ... */
}

/* Settings for slow_func(). */
#pragma push /* Save file settings. */
#pragma optimization_size 0
void slow_func(void)
{
/* ... */
}
#pragma pop /* Restore file settings. */

void fast_func_B(void)
{
/* ... */
}
```

Pragmas that accept the `reset` argument perform the same actions as pragmas `push` and `pop`, but apply to a single pragma. A pragma's `on` and `off` arguments save the pragma's current setting before changing it to the new setting. A pragma's `reset` argument restores the pragma's setting. The `on`, `off`, and `reset` arguments may be nested to an unlimited depth.*Example 76* shows an example.

Example 76 Using the reset option to save and restore a pragma setting

```
/* Setting for this file. */
#pragma opt_unroll_loops on

void fast_func_A(void)
{
/* ... */
}

/* Setting for smallslowfunc(). */
#pragma opt_unroll_loops off
void small_func(void)
{
/* ... */
}
/* Restore previous setting. */
```

```
#pragma opt_unroll_loops reset

void fast_func_B(void)
{
/* ... */
}
```

16.3 Determining which settings are saved and restored

Not all pragma settings are saved and restored by pragmas `push` and `pop`. Pragmas that do not change compiler settings are not affected by `push` and `pop`. For example, pragma `message` cannot be saved and restored.

Example 77 shows an example that checks if the `ANSI_strict` pragma setting is saved and restored by pragmas `push` and `pop`.

Example 77 Testing if pragmas push and pop save and restore a setting

```
/* Preprocess this source code. */
#pragma ANSI_strict on
#pragma push
#pragma ANSI_strict off
#pragma pop
#if __option(ANSI_strict)
#error "Saved and restored by push and pop."
#else
#error "Not affected by push and pop."
#endif
```

16.4 Invalid pragmas

If you enable the compiler's setting for reporting invalid pragmas, the compiler issues a warning when it encounters a pragma it does not recognize. For example, the pragma statements in *Example 78* generate warnings with the invalid pragmas setting enabled.

Example 78 Invalid pragmas

```
#pragma silly_data off        // WARNING: silly_data is not a pragma.
#pragma ANSI_strict select   // WARNING: select is not defined
#pragma ANSI_strict on       // OK
```

Table 26 shows how to control the recognition of invalid pragmas.

Table 26. Controlling invalid pragmas

To control this option from here...	use this setting
Source code	#pragma warn_illpragma
Command line	-warnings illpragmas

16.5 Pragma scope

The scope of a pragma setting is limited to a compilation unit (a source code file and the files that it includes).

At the beginning of compilation unit, the compiler uses its default settings. The compiler then uses the settings in command-line options.

The compiler uses the setting in a pragma beginning at the pragma's location in the compilation unit. The compilers continues using this setting:

- Until another instance of the same pragma appears later in the source code
- Until an instance of pragma `pop` appears later in the source code
- Until the compiler finishes translating the compilation unit

17 Pragmas for standard C conformance

17.1 ANSI_strict

Controls the use of non-standard language features.

Syntax

```
#pragma ANSI_strict on | off | reset
```

Remarks

If you enable the pragma `ANSI_strict`, the compiler generates an error message if it encounters some eTPU build tools extensions to the C language defined by the ISO/IEC 9899-1990 (“C90”) standard:

- C++-style comments
- Unnamed arguments in function definitions
- Non-standard keywords

By default, this pragma is `off`.

17.2 c99

Controls the use of a subset of ISO/IEC 9899-1999 (“C99”) language features.

Syntax

```
#pragma c99 on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts many of the language features described by the ISO/IEC 9899-1999 standard:

- More rigid type checking.
- Trailing commas in enumerations.
- GCC/C99-style compound literal values.
- Designated initializers.
- `__func__` predefined symbol.
- Implicit `return 0;` in `main()`.
- Non-`const` static data initializations.
- Variable argument macros (`__VA_ARGS__`).
- `bool` and `_Bool` support.
- `long long` support (separate switch).
- `restrict` support.
- `//` comments.
- `inline` support.
- Digraphs.
- `_Complex` and `_Imaginary` (treated as keywords but not supported).
- Empty arrays as last struct members.
- Designated initializers
- Hexadecimal floating-point constants.
- Variable length arrays are supported within local or function prototype scope (as required by the C99 standard).
- Unsuffixed decimal constant rules.
- `++bool--` expressions.
- `(T) (int-list)` are handled/parsed as cast-expressions and as literals.
- `__STDC_HOSTED__` is `1`.

By default, this pragma is disabled.

17.3 c9x

Equivalent to `#pragma c99`.

17.4 ignore_oldstyle

Controls the recognition of function declarations that follow the syntax conventions used before ISO/IEC standard C (in other words, “K&R” style).

Syntax

```
#pragma ignore_oldstyle on | off | reset
```

Remarks

If you enable this pragma, the compiler ignores old-style function declarations and lets you prototype a function any way you want. In old-style declarations, you specify the types of arguments on separate lines instead of the function's argument list. For example, the code in *Example 79* defines a prototype for a function with an old-style definition.

Example 79 Mixing old-style and prototype function declarations

```
int f(char x, short y, float z);

#pragma ignore_oldstyle on

f(x, y, z)
char x;
short y;
float z;
{
  return (int)x+y+z;
}

#pragma ignore_oldstyle reset
```

This pragma does not correspond to any panel setting. By default, this setting is disabled.

17.5 only_std_keywords

Controls the use of ISO/IEC keywords.

Syntax

```
#pragma only_std_keywords on | off | reset
```

Remarks

The compiler recognizes additional reserved keywords. If you are writing source code that must follow the ISO/IEC C standards strictly, enable the pragma `only_std_keywords`.

By default, this pragma is disabled.

17.6 require_prototypes

Controls whether or not the compiler should expect function prototypes.

Syntax

```
#pragma require_prototypes on | off | reset
```

Remarks

This pragma only affects non-static functions.

If you enable this pragma, the compiler generates an error message if you use a function that does not have a preceding prototype. Use this pragma to prevent error messages caused by referring to a function before you define it. For example, without a function prototype, you might pass data of the wrong type. As a result, your code might not work as you expect even though it compiles without error.

In *Example 80*, function `main()` calls `PrintNum()` with an integer argument even though `PrintNum()` takes an argument of type `float`.

Example 80 Unnoticed type-mismatch

```
#include <stdio.h>

void main(void)
{
  PrintNum(1);  /* PrintNum() tries to interpret the
                   integer as a float. Prints 0.000000. */
}

void PrintNum(float x)
{
    printf("%f\n", x);
}
```

When you run this program, you could get this result:

```
0.000000
```

Although the compiler does not complain about the type mismatch, the function does not give the result you intended. Since `PrintNum()` does not have a prototype, the compiler does not know to generate instructions to convert the integer to a floating-point number before calling `PrintNum()`. Consequently, the function interprets the bits it received as a floating-point number and prints nonsense.

A prototype for `PrintNum()`, as in *Example 81*, gives the compiler sufficient information about the function to generate instructions to properly convert its argument to a floating-point number. The function prints what you expected.

Example 81 Using a prototype to avoid type-mismatch

```
#include <stdio.h>

void PrintNum(float x); /* Function prototype. */

void main(void)
{
    PrintNum(1);        /*  Compiler converts int to float.
}                           Prints 1.000000. */

void PrintNum(float x)
{
    printf("%f\n", x);
}
```

In other situations where automatic conversion is not possible, the compiler generates an error message if an argument does not match the data type required by a function prototype. Such a mismatched data type error is easier to locate at compile time than at runtime.

18 Pragmas for language translation

18.1 asmpoundcomment

Controls whether the "#" symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmpoundcomment on | off | reset
```

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

```
#pragma asmpoundcomment off
```

is used.

Using this pragma may interfere with the function-level inline assembly language.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

18.2 asmsemicolcomment

Controls whether the "`;`" symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmsemicolcomment on | off | reset
```

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

```
#pragma asmsemicolcomment off
```

is used.

Using this pragma may interfere with the assembly language of a specific target.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

18.3 const_strings

Controls the `const`-ness of character string literals.

Syntax

```
#pragma const_strings [ on | off | reset ]
```

Remarks

If you enable this pragma, the type of string literals is an array `const char[n]`, or `const wchar_t[n]` for wide strings, where *n* is the length of the string literal plus 1 for a terminating `NUL` character. Otherwise, the type `char[n]` or `wchar_t[n]` is used.

By default, this pragma is `on` when compiling C++ source code and `off` when compiling C source code.

18.4 dollar_identifiers

Controls use of dollar signs ($) in identifiers.

Syntax

```
#pragma dollar_identifiers on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts dollar signs (`$`) in identifiers. Otherwise, the compiler issues an error if it encounters anything but underscores, alphabetic, numeric character, and universal characters (`\uxxxx`, `\Uxxxxxxxx`) in an identifier.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

18.5 gcc_extensions

Controls the acceptance of GNU C language extensions.

Syntax

```
#pragma gcc_extensions on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts GNU C extensions in C source code. This includes the following non-ANSI C extensions:

- Initialization of automatic `struct` or `array` variables with non-`const` values.
- Illegal pointer conversions
- sizeof(void) == 1
- sizeof(function-type) == 1
- Limited support for GCC statements and declarations within expressions.
- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`
- Function pointer arithmetic supported
- `void*` arithmetic supported
- Void expressions in return statements of void
- __builtin_constant_p `(`*`expr`*`)` supported
- Forward declarations of arrays of incomplete type
- Forward declarations of empty static arrays
- Pre-C99 designated initializer syntax (deprecated)
- shortened conditional expression (*`c`* `?:` *`y`*)
- long __builtin_expect `(long exp, long c)` now accepted

By default, this pragma is disabled.

18.6 mark

Adds an item to the function pop-up menu in the IDE editor.

Syntax

`#pragma mark` *itemName*

Remarks

This pragma adds *itemName* to the source file's Function pop-up menu. Note that if the pragma is inside a function definition, the item does not appear in the Function pop-up menu.

If *itemName* begins with "`--`", a menu separator appears in the IDE's Function pop-up menu:

```
#pragma mark --
```

This pragma does not correspond to any panel setting.

18.7 mpwc_newline

Controls the use of newline character convention.

Syntax

```
#pragma mpwc_newline on | off | reset
```

Remarks

If you enable this pragma, the compiler translates `'\n'` as a Carriage Return (0x0D) and `'\r'` as a Line Feed (0x0A). Otherwise, the compiler uses the ISO standard conventions for these characters.

If you enable this pragma, use ISO standard libraries that were compiled when this pragma was enabled.

If you enable this pragma and use the standard ISO standard libraries, your program will not read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings your program's output to the beginning of the current line instead of inserting a newline.

This pragma does not correspond to any IDE panel setting. By default, this pragma is disabled.

18.8 mpwc_relax

Controls the compatibility of the `char*` and `unsigned char*` types.

Syntax

```
#pragma mpwc_relax on | off | reset
```

Remarks

If you enable this pragma, the compiler treats `char*` and `unsigned char*` as the same type. Use this setting to compile source code written before the ISO C standards. Old source code frequently uses these types interchangeably.

This setting has no effect on C++ source code.

Note: Turning this option on may prevent the compiler from detecting some programming errors. We recommend not turning on this option.

Example 82 shows how to use this pragma to relax function pointer checking.

Example 82 Relaxing function pointer checking

```
#pragma mpwc_relax on
extern void f(char *);

/* Normally an error, but allowed. */
extern void(*fp1)(void *) = &f;

/* Normally an error, but allowed. */
extern void(*fp2)(unsigned char *) = &f;
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

18.9 multibyteaware

Controls how the Source encoding option in the IDE is treated

Syntax

```
#pragma multibyteaware on | off | reset
```

Remarks

This pragma is deprecated. See `#pragma text_encoding` for more details.

By default, this pragma is `off`.

18.10 multibyteaware_preserve_literals

Controls the treatment of multibyte character sequences in narrow character string literals.

Syntax

```
#pragma multibyteaware_preserve_literals on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

18.11 text_encoding

Identifies the character encoding of source files.

Syntax

```
#pragma text_encoding ( "name" | unknown | reset [, global] )
```

Parameters

`name`

The IANA or MIME encoding name or an OS-specific string that identifies the text encoding. The compiler recognizes these names and maps them to its internal decoders:

```
system US-ASCII ASCII ANSI_X3.4-1968
ANSI_X3.4-1968 ANSI_X3.4 UTF-8 UTF8 ISO-2022-JP
CSISO2022JP ISO2022JP CSSHIFTJIS SHIFT-JIS
SHIFT_JIS SJIS EUC-JP EUCJP UCS-2 UCS-2BE
UCS-2LE UCS2 UCS2BE UCS2LE UTF-16 UTF-16BE
UTF-16LE UTF16 UTF16BE UTF16LE UCS-4 UCS-4BE
UCS-4LE UCS4 UCS4BE UCS4LE 10646-1:1993
ISO-10646-1 ISO-10646 unicode
```

`global`

Tells the compiler that the current and all subsequent files use the same text encoding. By default, text encoding is effective only to the end of the file.

Remarks

By default, `#pragma text_encoding` is only effective through the end of file. To affect the default text encoding assumed for the current and all subsequent files, supply the “global” modifier.

By default, this setting is ASCII.

18.12 trigraphs

Controls the use trigraph sequences specified in the ISO standards.

Syntax

```
#pragma trigraphs on | off | reset
```

Remarks

If you are writing code that must strictly adhere to the ANSI standard, enable this pragma.

Table 27. Trigraph table

<table>
<tr><th>Trigraph</th><th>Character</th></tr>
<tr><td>??=</td><td>#</td></tr>
<tr><td>??/</td><td>\</td></tr>
<tr><td>??'</td><td>^</td></tr>
<tr><td>??(</td><td>[</td></tr>
<tr><td>??)</td><td>]</td></tr>
<tr><td>??!</td><td>|</td></tr>
<tr><td>??<</td><td>{</td></tr>
<tr><td>??></td><td>}</td></tr>
<tr><td>??-</td><td>~</td></tr>
</table>

Note: Use of this pragma may cause a portability problem for some targets.

Be careful when initializing strings or multi-character constants that contain question marks.

Example 83 Example of pragma trigraphs

```
char c = '????'; /* ERROR: Trigraph sequence expands to '??^ */
char d = '\?\?\?\?'; /* OK */
```

By default, this pragma is disabled.

18.13 unsigned_char

Controls whether or not declarations of type `char` are treated as `unsigned char`.

Syntax

```
#pragma unsigned_char on | off | reset
```

Remarks

If you enable this pragma, the compiler treats a `char` declaration as if it were an `unsigned char` declaration.

Note: If you enable this pragma, your code might not be compatible with libraries that were compiled when the pragma was disabled. In particular, your code might not work with the ISO standard libraries included with eTPU build tools.

By default, this setting is disabled.

19 Pragmas for diagnostic messages

19.1 extended_errorcheck

Controls the issuing of warning messages for possible unintended logical errors.

Syntax

#pragma extended_errorcheck on | off | reset

Remarks

If you enable this pragma, the compiler generates a warning message (not an error) if it encounters some common programming errors:

- An integer or floating-point value assigned to an `enum` type. *Example 84* shows an example.

Example 84 Assigning to an enumerated type

```
enum Day { Sunday, Monday, Tuesday, Wednesday,
           Thursday, Friday, Saturday } d;

d = 5; /* WARNING */
d = Monday; /* OK */
d = (Day)3; /* OK */
```

- An empty `return` statement in a function that is not declared `void`. For example, *Example 85* results in a warning message.

Example 85 A non-void function with an empty return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return; /* WARNING: empty return statement */
}
```

Example 86 shows how to prevent this warning message.

Example 86 A non-void function with a proper return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return err; /* OK */
}
```

By default, this setting is off.

19.2 maxerrorcount

Limits the number of error messages emitted while compiling a single file.

Syntax

```
#pragma maxerrorcount( num | off )
```

Parameters

`num`

Specifies the maximum number of error messages issued per source file.

`off`

Does not limit the number of error messages issued per source file.

Remarks

The total number of error messages emitted may include one final message:

`Too many errors emitted`

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

19.3 message

Tells the compiler to issue a text message to the user.

Syntax

```
#pragma message( msg )
```

Parameter

`msg`

Actual message to issue. Does not have to be a string literal.

Remarks

On the command line, the message is sent to the standard error stream.

This pragma does not correspond to any panel setting.

19.4 showmessagenumber

Controls the appearance of warning or error numbers in displayed messages.

Syntax

```
#pragma showmessagenumber on | off | reset
```

Remarks

When enabled, this pragma causes messages to appear with their numbers visible. You can then use the `warning` pragma with a warning number to suppress the appearance of specific warning messages.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

19.5 show_error_filestack

Controls the appearance of the current `#include` file stack within error messages occurring inside deeply-included files.

Syntax

#pragma show_error_filestack on | off | reset

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

19.6 suppress_warnings

Controls the issuing of warning messages.

Syntax

#pragma suppress_warnings on | off | reset

Remarks

If you enable this pragma, the compiler does not generate warning messages, including those that are enabled.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

19.7 sym

Controls the generation of debugger symbol information for subsequent functions.

Syntax

```
#pragma sym on | off | reset
```

Remarks

The compiler pays attention to this pragma only if you enable the debug marker for a file in the IDE project window. If you disable this pragma, the compiler does not put debugging information into the source file debugger symbol file (SYM or DWARF) for the functions that follow.

The compiler always generates a debugger symbol file for a source file that has a debug diamond next to it in the IDE project window. This pragma changes only which functions have information in that symbol file.

This pragma does not correspond to any panel setting. By default, this pragma is enabled.

19.8 unused

Controls the suppression of warning messages for variables and parameters that are not referenced in a function.

Syntax

#pragma unused (var_name [, var_name]...)

var_name

The name of a variable.

Remarks

This pragma suppresses the compile time warning messages for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body. The listed variables must be within the scope of the function.

In C++, you cannot use this pragma with functions defined within a class definition or with template functions.

Example 87 Example of pragma unused() in C

```
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int a)
{
  int b;
#pragma unused(a,b)
/* Compiler does not warn that a and b are unused. */

}
```

Example 88 Example of Pragma unused() in C++

```
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int /* No warning */)
{
  int b;
#pragma unused(b)
/* Compiler does not warn that b is unused. */

}
```

19.9 warning

Controls which warning numbers are displayed during compiling.

Syntax

`#pragma warning on | off | reset (num [, ...])`

This alternate syntax is allowed but ignored (message numbers do not match):

#pragma warning(*warning_type* : *warning_num_list* [, *warning_type*: *warning_num_list*, ...])

Parameters

`num`

The number of the warning message to show or suppress.

warning_type

Specifies one of the following settings:

- Default
- Disable
- Enable

warning_num_list

The warning_num_list is a list of warning numbers separated by spaces.

Remarks

Use the pragma `showmessagenumber` to display warning messages with their warning numbers.

This pragma only applies to front-end warnings. Using the pragma for the Power Architecture back-end warnings returns invalid message number warning.

The compiler allows, but ignores, the alternative syntax for compatibility with Microsoft® compilers.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

19.10 warning_errors

Controls whether or not warnings are treated as errors.

Syntax

#pragma warning_errors on | off | reset

Remarks

If you enable this pragma, the compiler treats all warning messages as though they were errors and does not translate your file until you resolve them.

19.11 warn_any_ptr_int_conv

Controls if the compiler generates a warning message when an integral type is explicitly converted to a pointer type or vice versa.

Syntax

#pragma warn_any_ptr_int_conv on | off | reset

Remarks

This pragma is useful to identify potential 64-bit pointer portability issues. An example is shown in *Example 89*.

Example 89 Example of warn_any_ptr_int_conv

```
#pragma warn_ptr_int_conv on

short i, *ip

void func() {
  i = (short)ip;
  /* WARNING: short type is not large enough to hold pointer. */
}

#pragma warn_any_ptr_int_conv on

void bar() {
  i  = (int)ip; /* WARNING: pointer to integral conversion. */
  ip = (short *)i; /* WARNING: integral to pointer conversion. */
}
```

Remarks

By default, this pragma is `off`.

19.12 warn_emptydecl

Controls the recognition of declarations without variables.

Syntax

#pragma warn_emptydecl on | off | reset

Remarks

If you enable this pragma, the compiler displays a warning message when it encounters a declaration with no variables.

Example 90 Examples of empty declarations in C and C++

```
#pragma warn_emptydecl on
int ; /* WARNING: empty variable declaration. */
int i; /* OK */

long j;; /* WARNING */
long j; /* OK */
```

Example 91 Example of empty declaration in C++

```
#pragma warn_emptydecl on
extern "C" {
}; /* WARNING */
```

By default, this pragma is disabled.

19.13 warn_extracomma

Controls the recognition of superfluous commas in enumerations.

Syntax

#pragma warn_extracomma on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a trailing comma in enumerations. For example,*Example 92* is acceptable source code but generates a warning message when you enable this setting.

Example 92 Warning about extra commas

```
#pragma warn_extracomma on
enum { mouse, cat, dog, };
/* WARNING: compiler expects an identifier after final comma. */
```

The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 (“C99”) standard.

By default, this pragma is disabled.

19.14 warn_filenamecaps

Controls the recognition of conflicts involving case-sensitive filenames within user includes.

Syntax

#pragma warn_filenamecaps on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when an `#include` directive capitalizes a filename within a user include differently from the way the filename appears on a disk. It also detects use of “8.3” DOS filenames in Windows® operating systems when a long filename is available. Use this pragma to avoid porting problems to operating systems with case-sensitive file names.

By default, this pragma only checks the spelling of user includes such as the following:

`#include "file"`

For more information on checking system includes, see *warn_filenamecaps_system*.

By default, this pragma is `off`.

19.15 warn_filenamecaps_system

Controls the recognition of conflicts involving case-sensitive filenames within system includes.

Syntax

#pragma warn_filenamecaps_system on | off | reset

Remarks

If you enable this pragma along with `warn_filenamecaps`, the compiler issues a warning message when an `#include` directive capitalizes a filename within a system include differently from the way the filename appears on a disk. It also detects use of “8.3” DOS filenames in Windows® systems when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive file names.

To check the spelling of system includes such as the following:

```
#include <file>
```

Use this pragma along with the *warn_filenamecaps* pragma.

By default, this pragma is off.

Note: *Some SDKs (Software Developer Kits) use “colorful” capitalization, so this pragma may issue a lot of unwanted messages.*

19.16 warn_hiddenlocals

Controls the recognition of a local variable that hides another local variable.

Syntax

#pragma warn_hiddenlocals on | off | reset

Remarks

When `on`, the compiler issues a warning message when it encounters a local variable that hides another local variable. An example appears in *Example 93*.

Example 93 Example of hidden local variables warning

```
#pragma warn_hiddenlocals on

void func(int a)
{
  {
    int a; /* WARNING: this 'a' obscures argument 'a'.
  }
}
```

By default, this setting is `off`.

19.17 warn_illpragma

Controls the recognition of invalid pragma directives.

Syntax

#pragma warn_illpragma on | off | reset

Remarks

If you enable this pragma, the compiler displays a warning message when it encounters a pragma it does not recognize.

By default, this setting is off.

19.18 warn_illtokenpasting

Controls whether or not to issue a warning message for improper preprocessor token pasting.

Syntax

#pragma warn_illtokenpasting on | off | reset

Remarks

An example of this is shown below:

```
#define PTR(x) x##* / PTR(y)
```

Token pasting is used to create a single token. In this example, `y` and `x` cannot be combined. Often the warning message indicates the macros uses "`##`" unnecessarily.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

19.19 warn_illunionmembers

Controls whether or not to issue a warning message for invalid union members, such as unions with reference or non-trivial class members.

Syntax

#pragma warn_illunionmembers on | off | reset

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

19.20 warn_impl_f2i_conv

Controls the issuing of warning messages for implicit `float`-to-`int` conversions.

Syntax

#pragma warn_impl_f2i_conv on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting floating-point values to integral values.*Example 94* provides an example.

Example 94 Example of implicit `float`-to-`int` conversion

```
#pragma warn_impl_f2i_conv on

float f;
signed int si;

int main()
{
   f  = si; /* WARNING */

#pragma warn_impl_f2i_conv off
   si = f; /* OK */
}
```

By default, this pragma is on.

19.21 warn_impl_i2f_conv

Controls the issuing of warning messages for implicit `int`-to-`float` conversions.

Syntax

#pragma warn_impl_i2f_conv on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting integral values to floating-point values. *Example 95* shows an example.

Example 95 Example of implicit `int`-to-`float` conversion

```
#pragma warn_impl_i2f_conv on

float f;
signed int si;

int main()
{
   si = f; /* WARNING */

#pragma warn_impl_i2f_conv off
   f  = si; /* OK */

}
```

By default, this pragma is `off`.

19.22 warn_impl_s2u_conv

Controls the issuing of warning messages for implicit conversions between the `signed int` and `unsigned int` data types.

Syntax

#pragma warn_impl_s2u_conv on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting either from `signed int` to `unsigned int` or vice versa.*Example 96* provides an example.

Example 96 Example of implicit conversions between `signed int` and `unsigned int`

```
#pragma warn_impl_s2u_conv on

signed int si;
unsigned int ui;

int main()
{
   ui = si; /* WARNING */
   si = ui; /* WARNING */

#pragma warn_impl_s2u_conv off
   ui = si; /* OK */
   si = ui; /* OK */
}
```

By default, this pragma is enabled.

19.23 warn_implicitconv

Controls the issuing of warning messages for all implicit arithmetic conversions.

Syntax

`#pragma warn_implicitconv on | off | reset`

Remarks

If you enable this pragma, the compiler issues a warning message for all implicit arithmetic conversions when the destination type might not represent the source value.*Example 97* provides an example.

Example 97 Example of implicit conversion

```
#pragma warn_implicitconv on

float f;
signed int si;
unsigned int ui;
```

```
int main()
{
   f  = si; /* WARNING */
   si = f; /* WARNING */
   ui = si; /* WARNING */
   si = ui; /* WARNING */
}
```

Note: This option "opens the gate" for the checking of implicit conversions. The sub-pragmas `warn_impl_f2i_conv`*,* `warn_impl_i2f_conv`*, and* `warn_impl_s2u_conv` *control the classes of conversions checked.*

By default, this pragma is `off`.

19.24 warn_largeargs

Controls the issuing of warning messages for passing non-"int" numeric values to unprototyped functions.

Syntax

#pragma warn_largeargs on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message if you attempt to pass a non-integer numeric value, such as a `float` or `long long`, to an unprototyped function when the `require_prototypes` pragma is disabled.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

19.25 warn_missingreturn

Issues a warning message when a function that returns a value is missing a `return` statement.

Syntax

#pragma warn_missingreturn on | off | reset

Remarks

An example is shown in *Example 98*.

Example 98 Example of warn_missingreturn pragma

```
#pragma warn_missingreturn on

int func()
{
  /* WARNING: no return statement. */
}
```

19.26 warn_no_side_effect

Controls the issuing of warning messages for redundant statements.

Syntax

#pragma warn_no_side_effect on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that produces no side effect. To suppress this warning message, cast the statement with `(void)`.*Example 99* provides an example.

Example 99 Example of Pragma warn_no_side_effect

```
#pragma warn_no_side_effect on
void func(int a,int b)
{
  a+b; /* WARNING: expression has no side effect */
  (void)(a+b); /* OK: void cast suppresses warning. */
}
```

By default, this pragma is `off`.

19.27 warn_padding

Controls the issuing of warning messages for data structure padding.

Syntax

#pragma warn_padding on | off | reset

Remarks

If you enable this pragma, the compiler warns about any bytes that were implicitly added after an ANSI C `struct` member to improve memory alignment. Refer to the appropriate *Targeting* manual for more information on how the compiler pads data structures for a particular processor or operating system.

By default, this setting is off.

19.28 warn_pch_portability

Controls whether or not to issue a warning message when `#pragma once on` is used in a precompiled header.

Syntax

#pragma warn_pch_portability on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when you use `#pragma once on` in a precompiled header. This helps you avoid situations in which transferring a precompiled header from machine to machine causes the precompiled header to produce different results. For more information, see pragma `once`.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

19.29 warn_possunwant

Controls the recognition of possible unintentional logical errors.

Syntax

#pragma warn_possunwant on | off | reset

Remarks

If you enable this pragma, the compiler checks for common, unintended logical errors:

- An assignment in either a logical expression or the conditional portion of an `if`, `while`, or `for` expression. This warning message is useful if you use `=` when you mean to use `==`.*Example 100* shows an example.

Example 100 Confusing = and == in Comparisons

```
if (a=b) f(); /* WARNING: a=b is an assignment. */

if ((a=b)!=0) f(); /* OK: (a=b)!=0 is a comparison. */

if (a==b) f(); /* OK: (a==b) is a comparison. */
```

- An equal comparison in a statement that contains a single expression. This check is useful if you use `==` when you meant to use `=`. *Example 101* shows an example.

Example 101 Confusing = and == Operators in Assignments

```
a == 0;         // WARNING: This is a comparison.
a = 0;          // OK: This is an assignment, no warning
```

- A semicolon (`;`) directly after a `while`, `if`, or `for` statement.

For example, *Example 102* generates a warning message.

Example 102 Empty statement

```
i = sockcount();
while (--i);  /* WARNING: empty loop. */
    matchsock(i);
```

If you intended to create an infinite loop, put white space or a comment between the `while` statement and the semicolon. The statements in *Example 103* suppress the above error or warning messages.

Example 103 Intentional empty statements

```
while (i++) ; /* OK: White space separation. */
while (i++) /* OK: Comment separation */ ;
```

By default, this pragma is off.

19.30 warn_ptr_int_conv

Controls the recognition the conversion of pointer values to incorrectly-sized integral values.

Syntax

#pragma warn_ptr_int_conv on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message if an expression attempts to convert a pointer value to an integral type that is not large enough to hold the pointer value.

Example 104 Example for #pragma warn_ptr_int_conv

```
#pragma warn_ptr_int_conv on

char *my_ptr;
char too_small = (char)my_ptr;  /* WARNING: char is too small. */
```

See also *Section 19.11: warn_any_ptr_int_conv*.

By default, this pragma is off.

19.31 warn_resultnotused

Controls the issuing of warning messages when function results are ignored.

Syntax

#pragma warn_resultnotused on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`.*Example 105* provides an example.

Example 105 Example of function calls with unused results

```
#pragma warn_resultnotused on

extern int bar();
void func()
{
  bar(); /* WARNING: result of function call is not used. */
  void(bar()); /* OK: void cast suppresses warning. */
}
```

This pragma does not correspond to any panel setting. By default, this pragma is off.

19.32 warn_undefmacro

Controls the detection of undefined macros in `#if` and `#elif` directives.

Syntax

#pragma warn_undefmacro on | off | reset

Remarks

Example 106 provides an example.

Example 106 Example of undefined macro

```
#if BADMACRO == 4 /* WARNING: undefined macro. */
```

Use this pragma to detect the use of undefined macros (especially expressions) where the default value 0 is used. To suppress this warning message, check if defined first.

Note: A warning message is only issued when a macro is evaluated. A short-circuited "`&&`" or "`||`" test or unevaluated "`?:`" will not produce a warning message.

By default, this pragma is `off`.

19.33 warn_uninitializedvar

Controls the compiler to perform some dataflow analysis and emits warning messages whenever local variables are initialized before being used.

Syntax

#pragma warn_uninitializedvar on | off | reset

Remarks

By default, this pragma is `on`.

19.34 warn_unusedarg

Controls the recognition of unreferenced arguments.

Syntax

#pragma warn_unusedarg on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters an argument you declare but do not use.

This check helps you find arguments that you either misspelled or did not use in your program.*Example 107* shows an example.

Example 107 Warning about unused function arguments

```
void func(int temp, int error);
{
  error = do_something(); /* WARNING: temp is unused. */
}
```

To prevent this warning, you can declare an argument in a few ways:

- Use the pragma `unused`, as in *Example 108*.

Example 108 Using pragma unused() to prevent unused argument messages

```
void func(int temp, int error)
{
  #pragma unused (temp)
  /* Compiler does not warn that temp is not used. */

  error=do_something();
}
```

- Do not give the unused argument a name.*Example 109* shows an example.

The compiler allows this feature in C++ source code. To allow this feature in C source code, disable ANSI strict checking.

Example 109 Unused, unnamed arguments

```
void func(int /* temp */, int error)
{
  /* Compiler does not warn that "temp" is not used. */

  error=do_something();
}
```

By default, this pragma is *`off`*.

19.35 warn_unusedvar

Controls the recognition of unreferenced variables.

Syntax

#pragma warn_unusedvar on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a variable you declare but do not use.

This check helps you find variables that you either misspelled or did not use in your program.*Example 110* shows an example.

Example 110 Unused local variables example

```
int error;
void func(void)
{
  int temp, errer; /* NOTE: errer is misspelled. */
  error = do_something(); /* WARNING: temp and errer are unused. */
}
```

If you want to use this warning but need to declare a variable that you do not use, include the pragma `unused`, as in *Example 111*.

Example 111 Suppressing unused variable warnings

```
void func(void)
{
  int i, temp, error;

  #pragma unused (i, temp) /* Do not warn that i and temp */
  error = do_something();    /* are not used */
}
```

By default, this pragma is off.

20 Pragmas for preprocessing

20.1 check_header_flags

Controls whether or not to ensure that a precompiled header's data matches a project's target settings.

Syntax

```
#pragma check_header_flags on | off | reset
```

Remarks

This pragma affects precompiled headers only.

If you enable this pragma, the compiler verifies that the precompiled header's preferences for `double` size, `int` size, and floating point math correspond to the build target's settings. If they do not match, the compiler generates an error message.

If your precompiled header file depends on these settings, enable this pragma. Otherwise, disable it.

By default, this pragma is `off`.

20.2 faster_pch_gen

Controls the performance of precompiled header generation.

Syntax

```
#pragma faster_pch_gen on | off | reset
```

Remarks

If you enable this pragma, generating a precompiled header can be much faster, depending on the header structure. However, the precompiled file can also be slightly larger.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

20.3 flat_include

Controls whether or not to ignore relative path names in `#include` directives.

Syntax

```
#pragma flat_include on | off | reset
```

Remarks

For example, when `on`, the compiler converts this directive

```
#include <sys/stat.h>
```

to

```
#include <stat.h>
```

Use this pragma when porting source code from a different operating system.

By default, this pragma is `off`.

20.4 fullpath_file

Controls if `__FILE__` macro expands to a full path or the base file name.

Syntax

```
#pragma fullpath_file on | off | reset
```

Remarks

When this pragma `on`, the `__FILE__` macro returns a full path to the file being compiled, otherwise it returns the base file name.

20.5 fullpath_prepdump

Shows the full path of included files in preprocessor output.

Syntax

```
#pragma fullpath_prepdump on | off | reset
```

Remarks

If you enable this pragma, the compiler shows the full paths of files specified by the `#include` directive as comments in the preprocessor output. Otherwise, only the file name portion of the path appears.

By default, this pragma is `off`.

20.6 keepcomments

Controls whether comments are emitted in the preprocessor output.

Syntax

```
#pragma keepcomments on | off | reset
```

Remarks

By default, this pragma is `off`.

20.7 line_prepdump

Shows `#line` directives in preprocessor output.

Syntax

```
#pragma line_prepdump on | off | reset
```

Remarks

If you enable this pragma, `#line` directives appear in preprocessing output. The compiler also adjusts line spacing by inserting empty lines.

Use this pragma with the command-line compiler's `-E` option to make sure that `#line` directives are inserted in the preprocessor output.

By default, this pragma is `off`.

20.8 macro_prepdump

Controls whether the compiler emits `#define` and `#undef` directives in preprocessing output.

Syntax

```
#pragma macro_prepdump on | off | reset
```

Remarks

Use this pragma to help unravel confusing problems like macros that are aliasing identifiers or where headers are redefining macros unexpectedly.

20.9 msg_show_lineref

Controls diagnostic output involving `#line` directives to show line numbers specified by the `#line` directives in error and warning messages.

Syntax

```
#pragma msg_show_lineref on | off | reset
```

Remarks

By default, this pragma is `on`.

20.10 msg_show_realref

Controls diagnostic output involving `#line` directives to show actual line numbers in error and warning messages.

Syntax

```
#pragma msg_show_realref on | off | reset
```

Remarks

By default, this pragma is `on`.

20.11 notonce

Controls whether or not the compiler lets included files be repeatedly included, even with `#pragma once on`.

Syntax

```
#pragma notonce
```

Remarks

If you enable this pragma, files can be repeatedly `#include`d, even if you have enabled `#pragma once on`. For more information, see *once*.

20.12 old_pragma_once

This pragma is no longer available.

20.13 once

Controls whether or not a header file can be included more than once in the same compilation unit.

Syntax

```
#pragma once [ on ]
```

Remarks

Use this pragma to ensure that the compiler includes header files only once in a source file. This pragma is especially useful in precompiled header files.

There are two versions of this pragma:

```
#pragma once
```

and

```
#pragma once on
```

Use `#pragma once` in a header file to ensure that the header file is included only once in a source file. Use `#pragma once on` in a header file or source file to ensure that *any* file is included only once in a source file.

Beware that when using `#pragma once on`, precompiled headers transferred from one host machine to another might not give the same results during compilation. This inconsistency is because the compiler stores the full paths of included files to distinguish between two distinct files that have identical file names but different paths. Use the `warn_pch_portability` pragma to issue a warning message when you use `#pragma once on` in a precompiled header.

Also, if you enable the `old_pragma_once on` pragma, the `once` pragma completely ignores path names.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

20.14 pop, push

Saves and restores pragma settings.

Syntax

```
#pragma push
#pragma pop
```

Remarks

The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings that resulted from the last `push` pragma. For example, see *Example 112*.

Example 112 push and pop example

```
#pragma ANSI_strict on
#pragma push /* Saves all compiler settings. */
#pragma ANSI_strict off
#pragma pop /* Restores ANSI_strict to on. */
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

Note: Pragmas directives that accept `on` | `off` | `reset` already form a stack of previous option values. It is not necessary to use `#pragma pop` or `#pragma push` with such pragmas.

20.15 pragma_prepdump

Controls whether pragma directives in the source text appear in the preprocessing output.

Syntax

```
#pragma pragma_prepdump on | off | reset
```

Remarks

By default, this pragma is `off`.

Note: When submitting bug reports with a preprocessor dump, be sure this option is enabled.

20.16 precompile_target

Specifies the file name for a precompiled header file.

Syntax

```
#pragma precompile_target filename
```

Parameters

`filename`

A simple file name or an absolute path name. If *filename* is a simple file name, the compiler saves the file in the same folder as the source file. If *filename* is a path name, the compiler saves the file in the specified folder.

Remarks

If you do not specify the file name, the compiler gives the precompiled header file the same name as its source file.

Example 113 shows sample source code from a precompiled header source file. By using the predefined symbols `__cplusplus` and the pragma `precompile_target`, the compiler can use the same source code to create different precompiled header files for C and C++.

Example 113 Using #pragma precompile_target

```
#ifdef __cplusplus
  #pragma precompile_target "MyCPPHeaders"
#else
  #pragma precompile_target "MyCHeaders"
#endif
```

This pragma does not correspond to any panel setting.

20.17 simple_prepdump

Controls the suppression of comments in preprocessing output.

Syntax

```
#pragma simple_prepdump on | off | reset
```

Remarks

By default, the compiler adds comments about the current include file being in preprocessing output. Enabling this pragma disables these comments.

By default, this pragma is `off`.

20.18 space_prepdump

Controls whether or not the compiler removes or preserves whitespace in the preprocessor's output.

Syntax

```
#pragma space_prepdump on | off | reset
```

Remarks

This pragma is useful for keeping the starting column aligned with the original source code, though the compiler attempts to preserve space within the line. This pragma does not apply to expanded macros.

By default, this pragma is `off`.

20.19 srcrelincludes

Controls the lookup of `#include` files.

Syntax

```
#pragma srcrelincludes on | off | reset
```

Remarks

When `on`, the compiler looks for `#include` files relative to the previously included file (not just the source file). When `off`, the compiler uses the access paths specified with the `-ir` option.

Use this pragma when multiple files use the same file name and are intended to be included by another header file in that directory. This is a common practice in UNIX programming.

By default, this pragma is `off`.

20.20 syspath_once

Controls how included files are treated when `#pragma once` is enabled.

Syntax

```
#pragma syspath_once on | off | reset
```

Remarks

When this pragma and pragma `once` are set to `on`, the compiler distinguishes between identically-named header files referred to in

`#include <`*`file-name`*`>`

and

`#include "`*`file-name`*`".`

When this pragma is `off` and pragma once is `on`, the compiler will ignore a file that uses a

```
#include <file-name>
```

directive if it has previously encountered another directive of the form

```
#include "file-name"
```

for an identically-named header file.

Example 114 shows an example.

This pragma does not correspond to any panel setting. By default, this setting is `on`.

Example 114 Pragma syspath_once example

```
#pragma syspath_once off
#pragma once on /* Include all subsequent files only once. */
#include "sock.h"
#include <sock.h> /* Skipped because syspath_once is off. */
```

21 Pragmas for code generation

21.1 aggressive_inline

Specifies the size of enumerated types.

Syntax

```
#pragma aggressive_inline on | off | reset
```

Remarks

The IPA-based inliner (-ipa file) will inline more functions when this option

is enabled. This option can cause code bloat in programs that overuse inline

functions. Default is off.

21.2 dont_reuse_strings

Controls whether or not to store identical character string literals separately in object code.

Syntax

```
#pragma dont_reuse_strings on | off | reset
```

Remarks

Normally, C and C++ programs should not modify character string literals. Enable this pragma if your source code follows the unconventional practice of modifying them.

If you enable this pragma, the compiler separately stores identical occurrences of character string literals in a source file.

If this pragma is disabled, the compiler stores a single instance of identical string literals in a source file. The compiler reduces the size of the object code it generates for a file if the source file has identical string literals.

The compiler always stores a separate instance of a string literal that is used to initialize a character array. *Example 115* shows an example.

Although the source code contains 3 identical string literals, `"cat"`, the compiler will generate 2 instances of the string in object code. The compiler will initialize `str1` and `str2` to point to the first instance of the string and will initialize `str3` to contain the second instance of the string.

Using `str2` to modify the string it points to also modifies the string that `str1` points to. The array `str3` may be safely used to modify the string it points to without inadvertently changing any other strings.

By default, this pragma is off.

Example 115 Reusing string literals

```
#pragma dont_reuse_strings off
void strchange(void)
{
```

```
  const char* str1 = "cat";
  char* str2 = "cat";
  char str3[] = "cat";

  *str2 = 'h'; /* str1 and str2 point to "hat"! */
  str3[0] = 'b';
  /* OK: str3 contains "bat", *str1 and *str2 unchanged.
}
```

21.3 enumsalwaysint

Specifies the size of enumerated types.

Syntax

```
#pragma enumsalwaysint on | off | reset
```

Remarks

If you enable this pragma, the C/C++ compiler makes an enumerated type the same size as an `int`. If an enumerated constant is larger than `int`, the compiler generates an error message. Otherwise, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long long`.

Example 116 shows an example.

Example 116 Example of enumerations the same as size as int

```
enum SmallNumber { One = 1, Two = 2 };
  /* If you enable enumsalwaysint, this type is
     the same size as an int. Otherwise, this type is
     the same size as a char. */

enum BigNumber
  { ThreeThousandMillion = 3000000000 };
  /* If you enable enumsalwaysint, the compiler might
     generate an error message. Otherwise, this type is
     the same size as a long long. */
```

By default, this pragma is `off`.

21.4 errno_name

Tells the optimizer how to find the `errno` identifier.

Syntax

```
#pragma errno_name id | ...
```

Remarks

When this pragma is used, the optimizer can use the identifier `errno` (either a macro or a function call) to optimize standard C library functions better. If not used, the optimizer makes worst-case assumptions about the effects of calls to the standard C library.

Note: The MSL C library already includes a use of this pragma, so you would only need to use it for third-party C libraries.

If `errno` resolves to a variable name, specify it like this:

```
#pragma errno_name _Errno
```

If `errno` is a function call accessing ordinarily inaccessible global variables, use this form:

```
#pragma errno_name ...
```

Otherwise, do not use this pragma to prevent incorrect optimizations.

This pragma does not correspond to any panel setting. By default, this pragma is unspecified (worst case assumption).

21.5 explicit_zero_data

Controls the placement of zero-initialized data.

Syntax

```
#pragma explicit_zero_data on | off | reset
```

Remarks

Places zero-initialized data into the initialized data section instead of the BSS section when `on`.

By default, this pragma is `off`.

21.6 float_constants

Controls how floating pointing constants are treated.

Syntax

```
#pragma float_constants on | off | reset
```

Remarks

If you enable this pragma, the compiler assumes that all unqualified floating point constant values are of type `float`, not `double`. This pragma is useful when porting source code for programs optimized for the “`float`” rather than the “`double`” type.

When you enable this pragma, you can still explicitly declare a constant value as double by appending a “D” suffix.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

21.7 instmgr_file

Controls where the instance manager database is written, to the target data directory or to a separate file.

Syntax

```
#pragma instmgr_file "name"
```

Remarks

When the Use Instance Manager option is on, the IDE writes the instance manager database to the project's data directory. If the `#pragma instmgr_file` is used, the database is written to a separate file.

Also, a separate instance file is always written when the command-line tools are used.

Note: Should you need to report a bug, you can use this option to create a separate instance manager database, which can then be sent to technical support with your bug report.

21.8 longlong

Controls the availability of the `long long` type.

Syntax

```
#pragma longlong on | off | reset
```

Remarks

When this pragma is enabled and the compiler is translating "C90" source code (ISO/IEC 9899-1990 standard), the compiler recognizes a data type named `long long`. The `long long` type holds twice as many bits as the `long` data type.

By default, this pragma is `on` for processors that support this type. It is `off` when generating code for processors that do not support, or cannot turn on, the `long long` type.

21.9 longlong_enums

Controls whether or not enumerated types may have the size of the `long long` type.

Syntax

```
#pragma longlong_enums on | off | reset
```

Remarks

This pragma lets you use enumerators that are large enough to be `long long` integers. It is ignored if you enable the `enumsalwaysint` pragma (described in *enumsalwaysint*).

This pragma does not correspond to any panel setting. By default, this setting is enabled.

21.10 min_enum_size

Specifies the size, in bytes, of enumeration types.

Syntax

```
#pragma min_enum_size 1 | 2 | 4
```

Remarks

Turning on the `enumsalwaysint` pragma overrides this pragma. The default is 1.

21.11 pool_strings

Controls how string literals are stored.

Syntax

```
#pragma pool_strings on | off | reset
```

Remarks

If you enable this pragma, the compiler collects all string constants into a single data object so your program needs one data section for all of them. If you disable this pragma, the compiler creates a unique data object for each string constant. While this decreases the number of data sections in your program, on some processors it also makes your program bigger because it uses a less efficient method to store the address of the string.

This pragma is especially useful if your program is large and has many string constants.

Note: If you enable this pragma, the compiler ignores the setting of the `pcrelstrings` *pragma.*

21.12 readonly_strings

Controls whether string objects are placed in a read-write or a read-only data section.

Syntax

```
#pragma readonly_strings on | off | reset
```

Remarks

If you enable this pragma, literal strings used in your source code are output to the read-only data section instead of the global data section. In effect, these strings act like `const char *`, even though their type is really `char *`.

This pragma does not correspond to any IDE panel setting.

21.13 reverse_bitfields

Controls whether or not the compiler reverses the bitfield allocation.

Syntax

```
#pragma reverse_bitfields on | off | reset
```

Remarks

This pragma reverses the bitfield allocation, so that bitfields are arranged from the opposite side of the storage unit from that ordinarily used on the target. The compiler still orders the bits within a single bitfield such that the lowest-valued bit is in the right-most position.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

Note: Limitation: please be aware of the following limitations when this pragma is set to `on`:
- the data types of the bit-fields must be the same data type
- the structure (`struct`) or `class` must not contain non-bit-field members; however, the structure (`struct`) can be the member of another structure

21.14 store_object_files

Controls the storage location of object data, either in the target data directory or as a separate file.

Syntax

```
#pragma store_object_files on | off | reset
```

Remarks

By default, the IDE writes object data to the project's target data directory. When this pragma is on, the object data is written to a separate object file.

Note: For some targets, the object file emitted may not be recognized as actual object data.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

22 Pragmas for optimization

22.1 global_optimizer

Controls whether the Global Optimizer is invoked by the compiler.

Syntax

#pragma global_optimizer on | off | reset

Remarks

In most compilers, this #pragma determines whether the Global Optimizer is invoked (configured by options in the panel of the same name). If disabled, only simple optimizations and back-end optimizations are performed.

Note: This is not the same as `#pragma optimization_level`. The Global Optimizer is invoked even at `optimization_level 0` if `#pragma global_optimizer` is enabled.

This pragma corresponds to the settings in the Global Optimizations panel. By default, this setting is on.

22.2 opt_common_subs

Controls the use of common subexpression optimization.

Syntax

#pragma opt_common_subs on | off | reset

Remarks

If you enable this pragma, the compiler replaces similar redundant expressions with a single expression. For example, if two statements in a function both use the expression

```
a * b * c + 10
```

the compiler generates object code that computes the expression only once and applies the resulting value to both statements.

The compiler applies this optimization to its own internal representation of the object code it produces.

This pragma does not correspond to any panel setting. By default, this settings is related to the *global_optimizer* pragma.

22.3 opt_dead_assignments

Controls the use of dead store optimization.

Syntax

#pragma opt_dead_assignments on | off | reset

Remarks

If you enable this pragma, the compiler removes assignments to unused variables before reassigning them.

This pragma does not correspond to any panel setting. By default, this settings is related to the *global_optimizer* level.

22.4 opt_dead_code

Controls the use of dead code optimization.

Syntax

```
#pragma opt_dead_code on | off | reset
```

Remarks

If you enable this pragma, the compiler removes a statement that other statements never execute or call.

This pragma does not correspond to any panel setting. By default, this settings is related to the *global_optimizer* level.

22.5 opt_lifetimes

Controls the use of lifetime analysis optimization.

Syntax

#pragma opt_lifetimes on | off | reset

Remarks

If you enable this pragma, the compiler uses the same processor register for different variables that exist in the same routine but not in the same statement.

This pragma does not correspond to any panel setting. By default, this settings is related to the *global_optimizer* level.

22.6 opt_loop_invariants

Controls the use of loop invariant optimization.

Syntax

#pragma opt_loop_invariants on | off | reset

Remarks

If you enable this pragma, the compiler moves all computations that do not change inside a loop outside the loop, which then runs faster.

This pragma does not correspond to any panel setting.

22.7 opt_propagation

Controls the use of copy and constant propagation optimization.

Syntax

#pragma opt_propagation on | off | reset

Remarks

If you enable this pragma, the compiler replaces multiple occurrences of one variable with a single occurrence.

This pragma does not correspond to any panel setting. By default, this settings is related to the *global_optimizer* level.

22.8 opt_strength_reduction

Controls the use of strength reduction optimization.

Syntax

#pragma opt_strength_reduction on | off | reset

Remarks

If you enable this pragma, the compiler replaces array element arithmetic instructions with pointer arithmetic instructions to make loops faster.

This pragma does not correspond to any panel setting. By default, this settings is related to the *global_optimizer* level.

22.9 opt_strength_reduction_strict

Uses a safer variation of strength reduction optimization.

Syntax

#pragma opt_strength_reduction_strict on | off | reset

Remarks

Like the *opt_strength_reduction* pragma, this setting replaces multiplication instructions that are inside loops with addition instructions to speed up the loops. However, unlike the regular strength reduction optimization, this variation ensures that the optimization is only applied when the array element arithmetic is not of an unsigned type that is smaller than a pointer type.

This pragma does not correspond to any panel setting. The default varies according to the compiler.

22.10 opt_unroll_loops

Controls the use of loop unrolling optimization.

Syntax

```
#pragma opt_unroll_loops on | off | reset
```

Remarks

If you enable this pragma, the compiler places multiple copies of a loop's statements inside a loop to improve its speed.

This pragma does not correspond to any panel setting. By default, this settings is related to the *global_optimizer* level.

22.11 opt_vectorize_loops

Controls the use of loop vectorizing optimization.

Syntax

```
#pragma opt_vectorize_loops on | off | reset
```

Remarks

If you enable this pragma, the compiler improves loop performance.

Note: Do not confuse loop vectorizing with the vector instructions available in some processors. Loop vectorizing is the rearrangement of instructions in loops to improve performance. This optimization does not optimize a processor's vector data types.

By default, this pragma is `off`.

22.12 optimization_level

Controls global optimization.

Syntax

```
#pragma optimization_level 0 | 1 | 2 | 3 | 4 | reset
```

Remarks

This pragma specifies the degree of optimization that the global optimizer performs.

To select optimizations, use the pragma `optimization_level` with an argument from `0` to `4`. The higher the argument, the more optimizations performed by the global optimizer. The `reset` argument specifies the previous optimization level.

For more information on the optimization the compiler performs for each optimization level, refer to the *Targeting* manual for your target platform.

These pragmas correspond to the settings in the Global Optimizations panel. By default, this pragma is disabled.

22.13 optimize_for_size

Controls optimization to reduce the size of object code.

```
#pragma optimize_for_size on | off | reset
```

Remarks

This setting lets you choose what the compiler does when it must decide between creating small code or fast code. If you enable this pragma, the compiler creates smaller object code at the expense of speed. It also ignores the `inline` directive and generates function calls to call any function declared `inline`. If you disable this pragma, the compiler creates faster object code at the expense of size.

The pragma corresponds to the optimize for size setting on the global optimizations panel.

22.14 optimizewithasm

Controls optimization of assembly language.

Syntax

```
#pragma optimizewithasm on | off | reset
```

Remarks

If you enable this pragma, the compiler also optimizes assembly language statements in C/C++ source code.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

22.15 pack

Stores data to reduce data size instead of improving execution performance.

Syntax

```
#pragma pack()
```

```
#pragma pack(0 | n | push | pop)
```

`n`

One of these integer values: 1, 2, 4, 8, or 16.

Remarks

Use this pragma to align data to use less storage even if the alignment might affect program performance or does not conform to the target platform's application binary interface (ABI).

If this pragma's argument is a power of 2 from 1 to 16, the compiler will store subsequent data structures to this byte alignment.

The `push` argument saves this pragma's setting on a stack at compile time. The `pop` argument restores the previously saved setting and removes it from the stack. Using this pragma with no argument or with 0 as an argument specifies that the compiler will use ABI-conformant alignment.

Not all processors support misaligned accesses, which could cause a crash or incorrect results. Even on processors which allow misaligned access, your program's performance might be reduced. Your program may have better performance if it treats the packed structure as a byte stream, then packs and unpacks each byte from the stream.

Note: Pragma `pack` is implemented somewhat differently by most compiler vendors, especially when used with bitfields. If you need portability, you are probably better off using explicit shift and mask operations in your program instead of bitfields.

22.16 strictheaderchecking

Controls how strict the compiler checks headers for standard C library functions.

Syntax

```
#pragma strictheaderchecking on | off | reset
```

Remarks

The 3.2 version compiler recognizes standard C library functions. If the correct prototype is used, and, in C++, if the function appears in the "`std`" or root namespace, the compiler recognizes the function, and is able to optimize calls to it based on its documented effects.

When this #pragma is `on` (default), in addition to having the correct prototype, the declaration must also appear in the proper standard header file (and not in a user header or source file).

This pragma does not correspond to any panel setting. By default, this pragma is on.

23 eTPU specific features

This chapter describes the eTPU specific features in the compiler:

- *Restrictions on 32-bit variables*
- *Host interface files*
- *eTPU functions structure*
- *pragma ETPU_function*
- *Memory allocation*
- *Channel structure*
- *Tooth Program Register (TPR) structure*
- *struct tpr_struct <varname> @ tpr;*
- *Entry table intrinsic functions*
- *Predefined symbols*
- *Integer types*
- *Fractional types*
- *Inline assembly*
- *#pragma write*
- *#pragma fill*
- *__attribute__((expects_flags))*
- *__attribute__((no_save_registers))*
- *__attribute__((pure_assembly))*
- *eTPU intrinsic functions*

23.1 Restrictions on 32-bit variables

eTPU has limited support for 32 bit variables. This is due to 24 bit natural word size of eTPU. You can declare 32 bit variables, and perform moves (assignments). This allows larger values to be moved around. Math operations are not supported for 32-bit values.

- Pointers to functions are not supported.
- Floating point are not supported.
- Standard libraries are not provided.

23.2 Host interface files

The compiler creates files intended for the CPU program, which configures eTPU. These files are named as:

filename_CPU.letter

Where filename is the base name of the eTPU source file being compiled, and letter is an alphabet letter from a to z.

23.3 eTPU functions structure

eTPU functions have a special structure. An eTPU function consists of several threads. A thread is executed when the eTPU scheduler assigns execution to it. Using the special structure the programmer can associate each thread's code with different conditions in the entry table. The special structure of an eTPU function inludes the following elements:

- A #pragma ETPU_function declaration.
- A void function.
- In the main scope of the function, a series of if()/else statements, each testing one or more elements of the channel condition see restrictions below.
  - - hsr - Host service request 0..7
  - - lsr - Link service request 0/1
  - - m1 - Match1/Transition2 0/1
  - - m2 - Match2/Transition1 0/1
  - - pin - Input pin 0/1
  - - flag0 0/1
  - - flag1 0/1

The threads code itself resides between these if then elses. Using the conditions the compiler associates the thread's code with the correct entry points.

A final else (without if) should be present at the end of this structure to collect all the unused entry points. A warning shall be issued if that else shall be omitted by the user.

In the thread's condition expression, usage of constants is allowed as well as the following operators: ==, !=, &&, ||, !.

There are restrictions on both possible conditions and the order in which they may occur. The following notes give details:

- In expressions that do not explicitly test hsr, an implicit test of hsr==0 is assumed.
- The compiler would issue an error message if the condition expression is not valid given the entry table that was chosen in the ETPU_function pragma. If there is an x in the column of a condition in the table in all qualified entries, the condition cannot be tested in that expression. For example the statement – if (lsr && pin) is legal because after evaluating lsr==1 we have entries 10,11,24-31 qualified and in some of them pin has a specific value.
- But, if (lsr && m1 && pin) is not valid since lsr && m1 qualifies 10,11,30,31 and in all of them pin has an X value and therefore none of them might be selected.
- The order of the conditions and events should be as follows. There are 3 groups. Identifiers from a lower group number should appear first. There is no order that should be enforced within the group:
  a) hsr
  b) lsr, m1,m2
  c) pin, flag0, flag1

An error should be generated when appropriate.

The order of the if then else blocks counts:

```
If (m1)
{
}
Else if (m2)
{
}
```

Would generate a different entry table than

```
If (m2)
{
}
Else if (m1)
{
}
```

since the first implies

```
if (m1)
{
}
else if (!m1 && m2)
{
}
```

and the second implies

```
if (m2)
{
}
else if (!m2 && m1)
{
}
```

- In every expression, one of the following events must be tested and have a non zero value: hsr, lsr, m1, m2.

23.4 pragma ETPU_function

#pragma ETPU_function name [, standard|alternate] [@ func_num];

This pragma tells the compiler that this function is an ETPU function. The compiler will create the needed entry table section for this function so that execution would be given to the different threads according to the condition expressions given in the function and the standard|alternate specification. It would also locate this entry table section in the correct location according to the given func_num.

If (standard|alternate) is not specified, standard is applied.

If a func_num is not specified, the compiler would assign a number automatically.

23.5 Memory allocation

The special architecture of the eTPU is quite different from that of most common architectures. The number of GPRs is small and there are a lot of restrictions using them. There is also no indirect access to memory with an offset from a register. These characteristics lead us to have a non standard memory allocation model.

Global addresses 0-7 are for internal compiler usage and should not be used by the programmer.

ETPU_functions arguments and static variables are allocated in a section that reside in the channel parameter area pointed by the CPBA field. Access to these parameters shall be using the selected channel relative addressing mode. The arguments and static variables of an ETPU_function may occupy up to 512 bytes on the channel relative context. The arguments are treated as static variables and continue to live after the thread ends.

Global variables are allocated in a section that resides in address 8 by default so that access to these variables is done using the absolute addressing mode when possible.

Spilled local variables are allocated along with the global variables and also must be accessed using the absolute addressing mode when possible.

For eTPU2 spilled local variables are allocated in a separate section, which can be accessed using the engine relative accessing mode.

23.6 Channel structure

Channels are represented by a C structure of type chan_struct, which is declared in the standard header file.

Only constants must be assigned to the members. The compiler validates values specific to each field and generates an error when required. When assigning a value to one of the members, the compiler generates an instruction, which corresponds to a field name from the architecture spec, and assigns this field the given constant.

Note: There is a backward compatibility problem as some of the fields where expanded in the HW and have an extra bit in eTPU2 so the users who used CIRC and TDL in their eTPU application would have to check their code.

23.7 Tooth Program Register (TPR) structure

The TPR structure exposes the TPR register fields to manipulation from the C language. This structure is declared in the standard header file.

```
struct tpr_struct {
  int TICKS    : 10;
```

```
int TPR10   : 1;
int HOLD    : 1;
int IPH     : 1;
int MISSCNT : 2;
int LAST    : 1;
} ;
```

The user may use the following syntax to associate a variable with the TPR register:

```
struct tpr_struct <varname> @ tpr;
```

23.8 Entry table intrinsic functions

- Enable_match()/Disable_match(): Sets or clears the ME bit in the entry table entries which are associated with the thread.
- preload_p01()/preload_p23(): Sets or clears the PP bit in the entry table entries, which are associated with the thread.

Note: These functions are implemented only for backward compatibility. It is better not to use them since if they are omitted, the compiler computes the best preload option itself and optimizes the code accordingly.

- read_match(): Loads the values of the Match registers into ERTA and ERTB.

23.9 Predefined symbols

The compiler supports the following predefined symbols:

Table 28. Predefined symbol

Symbol	Description
__DATE__	A string representing the compilation date.
__FILE__	A string representing the name of the file in which the symbol appears.
__LINE__	A string representing the line number in which the symbol appears.
__TIME__	A string representing the compilation time.
__ETPU__	A string representing the compilation for eTPU.
__ETPU2__	A string representing the compilation for eTPU2.

23.10 Integer types

All standard C types are supported.

Two new integer data types are created such as int24, which would be the default for int and unsigned int24, which would be the default for unsigned int.

long and unsigned long are the native types for 32 bits.

The following identifiers are also supported: _Bool, int8, int16, int32, int8_t, int16_t, int24_t, int32_t, uint8, uint16, uint32, uint8_t, uint16_t, uint24_t, uint32_t.

23.11 Fractional types

fract8, fract16, and fract24 types represent fractional numbers of the specified size in bits. Unsigned and signed modifiers can be applied to them. Unsigned fract can represent numbers between 0 and 1. Signed fract can represent numbers between -1 and 1.

23.12 Inline assembly

The inline assembly statement syntax is:

```
asm{"<assembly instructions>"};
```

Both multi line and single line assembly instructions may be omitted using this statement. The <assembly instruction> would be according to the new assembly language which shall be defined by DevTech.

23.12.1 Inline assembler usage

Use the inline assembler in order to write assembler code that is eTPU specific and cannot be expressed using the C language:

```
asm{" add.f p, p, diob"};
```

23.12.2 Specifying variables and labels

You may also use local variable names and labels inside inline assembly statements to reference variables of a C function or targets for change of flow directly.

In the example below, a local variable name needs a @Rn suffix to be recognized. Do not replace @Rn with an actual register number. It has to be the verbatim text made of the letters @, R, and n:

```
asm{" add x@Rn,p,diob"}; // x is a local variable

asm{" jmp label_name"};

asm{" ld p, glob_var"}; // glob_var is a global variable

asm{" ldm p, chan_var"}; // chan_var is a variable allocated on the
channel parameter ram
```

Finally, you can also declare labels using the inline assembler to, for example, mark the beginning for a special assembly loop or branch target:

```
asm{__mysmstart: jmp __mysmstart};
```

23.12.3 Using datatype sizes

You might want to reference the size of a structure from the inline assembler:

```
asm {addi x@Rn, x@Rn, myStruct@sizeof};
```

23.13 #pragma write

```
#pragma write char, (text);
```

This pragma writes information into the host interface files. The created file would have the name of the compiled file with the extension <char>. The <text> information can be either direct text or ::ETPU macros, which are expanded at link time and can give the host application information regarding the code and the data variables location and initialization.

23.14 #pragma fill

```
#pragma fill = list, ...;
#pragma fill [size] @ location = list, ... ;
```

The compiler will fill program memory with <list>. <list> items are any values or strings, defined as constant data separated by commas.

When size is not specified the compiler will simply emit the list in memory. If <size> is specified, the compiler will fill <size> words of memory with the data. The compiler will truncate the list to fit size, or repeat it to fill exactly <size> words.

23.15 __attribute__((expects_flags))

Specifies that a function is using the flags created in the caller function by the assignment into the first argument

Syntax

__attribute__((expects_flags)) function-declaration;

__attribute__((expects_flags)) function-definition;

23.16 __attribute__((no_save_registers))

Specifies that this function do not save and restore its registers. The only thing added in its epilogue is an rtn instruction. This attribute is not recommended and is here only for backwards compatibility with old code.

Syntax

__attribute__((no_save_registers)) function-declaration;

__attribute__((no_save_registers)) function-definition;

23.17 __attribute__((pure_assembly))

Specifies that this function contains only inline assembly instructions and it will not be optimized. It would also not have any prologue or epilogue. This attribute is not recommended and is here only for backwards compatibility with old code.

The recommended way of doing it is simply writing the code in an assembly file and not as C inline assembly.

Syntax

__attribute__((pure_assembly)) function-declaration;

__attribute__((pure_assembly)) function-definition;

23.18 eTPU intrinsic functions

Fraction to integer conversion. Use the following intrinsics to smoothly convert a fraction variable into integer variable without causing it to round to 0 or 1.

`_int_from_fract`

Coverts from signed fraction to signed integer.

`_int_from_ufract`

Converts from unsigned fraction to signed integer.

`_uint_from_fract`

Converts from signed fraction to unsigned integer.

`_uint_from_ufract`

Converts from unsigned fraction to unsigned integer.

24 Revision history

Table 29. Document revision history

Date	Revision	Changes
14-Apr-2011	1	Initial release.
18-Sep-2013	2	Updated Disclaimer.

Please Read Carefully: