Epiphany SDK Reference

Table of Contents

REV 5.13.07.10

REV 5.13.07.10

REV 5.13.07.10

List of Figures

List of Tables

1. Introduction

1.1 SDK Overview

The Epiphany™ architecture defines a multicore, scalable, shared-memory computing fabric. It consists of a 2D array of mesh compute nodes connected by a low-latency mesh network-on-chip. The Epiphany Software Development Kit (eSDK) is a state-of-the art software development environment targeting the Epiphany multicore architecture. The eSDK is based on standard development tools including an optimizing C-compiler, functional simulator, debugger, and multicore integrated development environment (IDE). The eSDK enables out-of-the-box execution of applications written in regular ANSI-C and does not require any C-subset, language extensions, or SIMD style programming. The unparalleled energy efficiency of the Epiphany architecture and the ease of use and fine grain control of the eSDK offer developers best-in-class capabilities for the most demanding real-time applications. The Epiphany SDK framework is illustrated in Figure 1.1 and contains the following key components:

- Optimized ANSI-C compiler (based on gcc)

- Robust multicore Eclipse IDE (on selected platforms)

- Multicore debugger (based on gdb)

- Multicore communication and hardware utility libraries

- Fast functional simulator with instruction trace capability.

Figure 1.1: Epiphany SDK

    REV 5.13.07.10
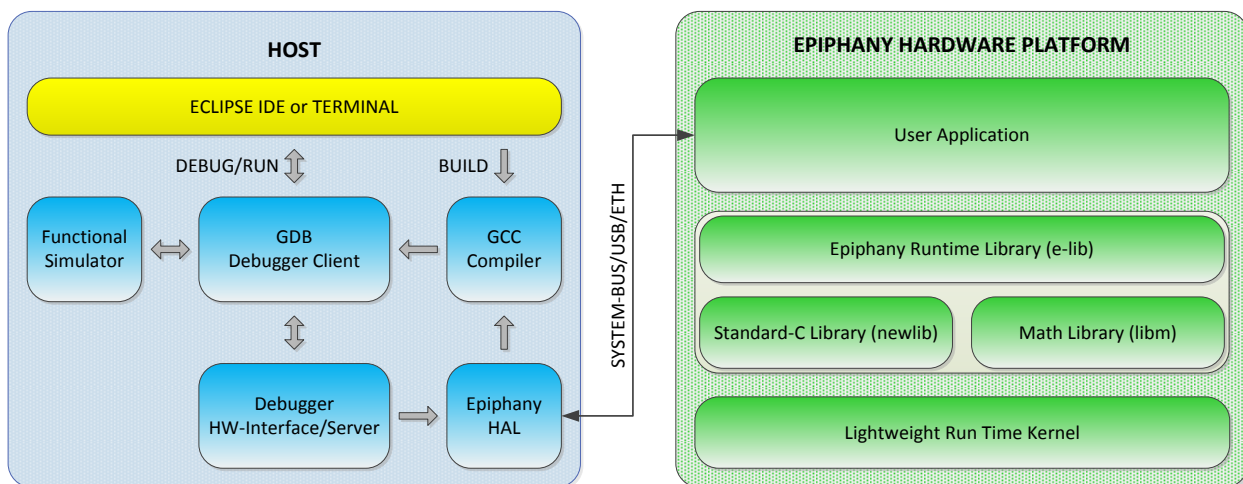
1.2 Epiphany Memory Model

The Epiphany architecture uses a single, flat unprotected address space consisting of 2^{32} 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to $2^{32}-1$. This address space is regarded as consisting of 2^{30} 32-bit words, each of whose addresses is word-aligned, which means that the address is divisible by 4. The word whose word-aligned address is A consists of the four bytes with addresses A, A+1, A+2 and A+3. Each mesh node has a local, aliased, range of local memory that is accessible by the mesh node itself starting at address 0x0. On 32KB chip models, it ends at address 0x00007FFF. Each mesh node also has a globally addressable ID that allows communication with all other mesh nodes in the system. The mesh-node ID consists of 6 row-ID bits and 6 column-ID bits situated at the upper most-significant bits (MSBs) of the address space. The complete memory is shown in Figure 1.2.

Figure 1.2: Epiphany Global Address Map

GLOBAL SPACE

CORE_63_63	0xFFF00000
...	
CORE_63_3	0xFC300000
CORE_63_2	0xFC200000
CORE_63_1	0xFC100000
CORE_63_0	0xFC000000
...	
CORE_1_63	0x07F00000
...	
CORE_1_3	0x04300000
CORE_1_2	0x04200000
CORE_1_1	0x04100000
CORE_1_0	0x04000000
CORE_0_63	0x03F00000
...	
CORE_0_3	0x00300000
CORE_0_2	0x00200000
CORE_0_1	0x00100000
LOCAL MEMORY	0x00000000

LOCAL SPACE

0x000F0000	MEMORY-MAPPED REGISTERS
	RESERVED
0x00006000	INTERNAL MEMORY BANK 3
0x00004000	INTERNAL MEMORY BANK 2
0x00002000	INTERNAL MEMORY BANK 1
0x00000000	INTERNAL MEMORY BANK 0

Data and code can be placed anywhere in the memory space or in external space, except for the memory-mapped register space and reserved space, but performance is optimized when the data and code are placed in separate local-memory banks.

      REV 5.13.07.10

1.3 Epiphany Programming Framework

Each one of the Epiphany processor nodes can run independent programs. Figure 1.3 shows the general programming flow for the Epiphany architecture, highlighting the independent build of programs running on different cores and the use of a common loader to load the complete multicore program onto the chip. The multicore IDE handles the detail of configuring and building multicore projects.

Figure 1.3: Epiphany Program Build Flow
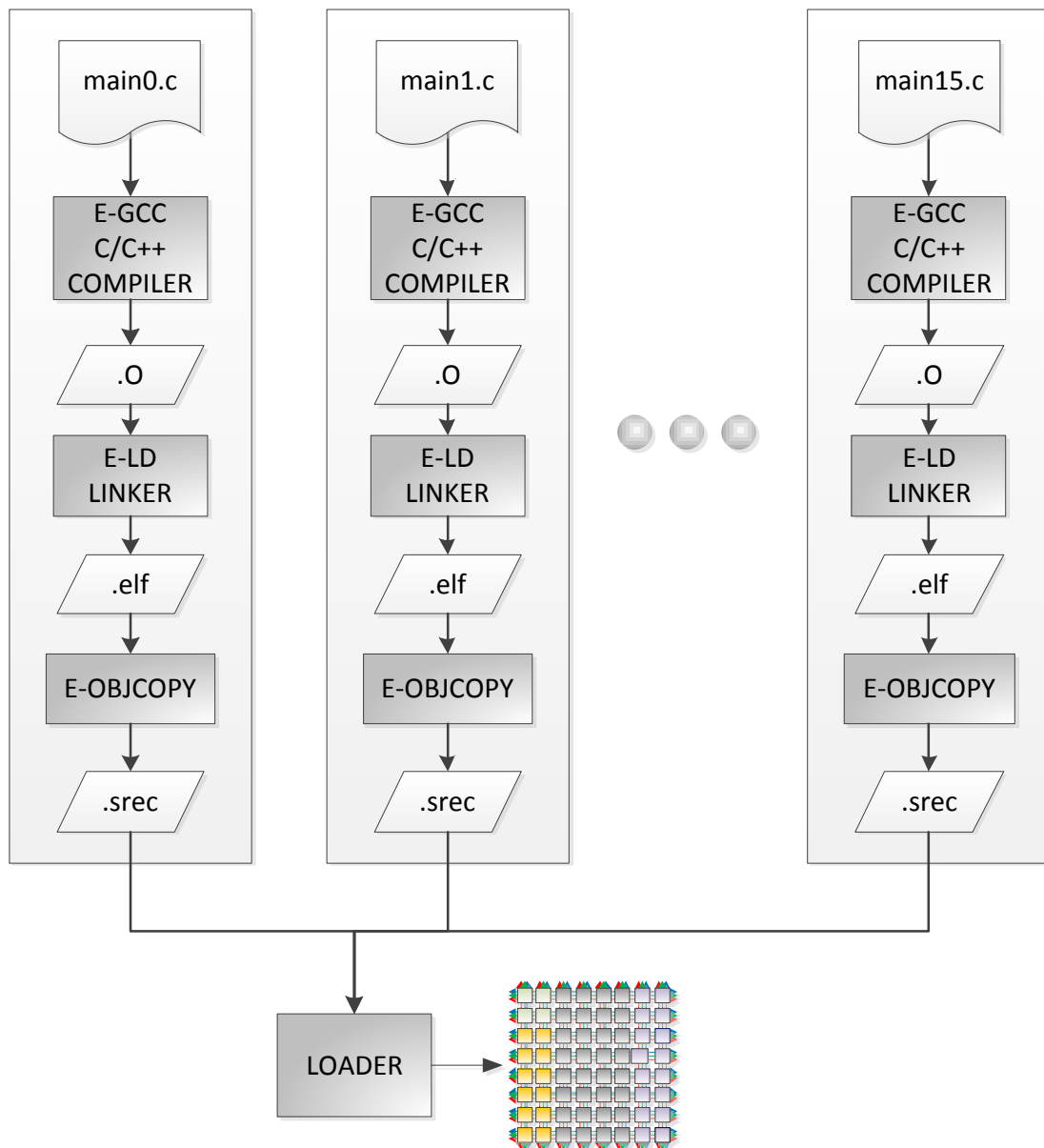
1.4   Epiphany SDK Directory Structure

The Epiphany SDK framework was designed to support multiple platforms, possibly mix-and-matching development platform and deployment platform. Additionally, it is meant to support multiple revisions of the package. As such, a few key components are being referred to by symbolic links instead of their actual name. It is also relocatable across the filesystem, and the root directory discussed here is the default location. Figure 1.4 shows the directory structure of the installed Epiphany SDK.

Figure 1.4: Epiphany SDK Directory Structure
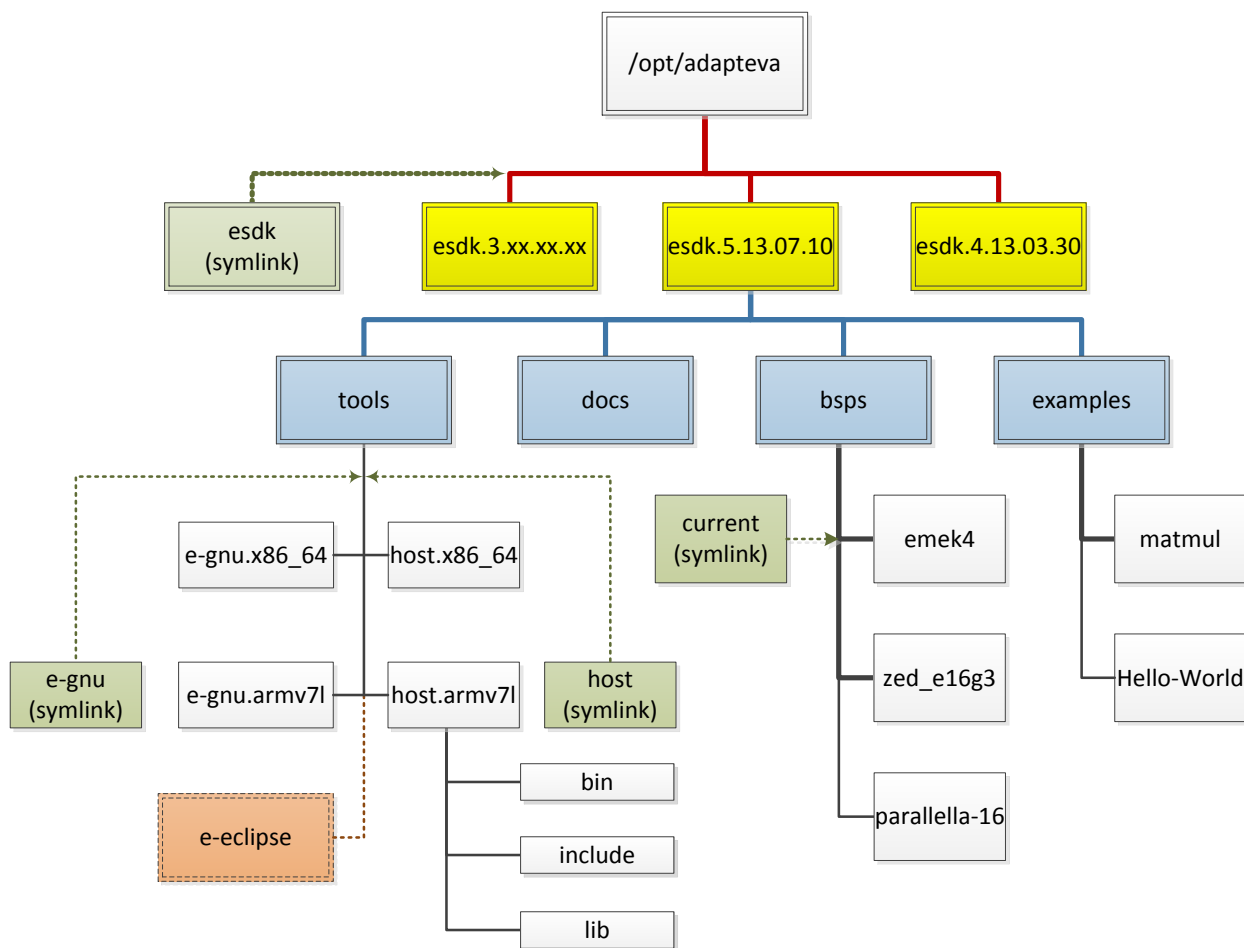
The eSDK framework contains these main components:

tools: The Epiphany GNU toolchain, Epiphany runtime libraries and the host-specific software.

Copyright 2011-2013 Adapteva. All rights reserved   REV 5.13.07.10

`bsps:`	Board Support Packages, with per-platform specific files like default linker scripts and platform definition files.
`docs:`	eSDK documentation, including the eSDK reference and the Architecture Reference.
`examples:`	Some code examples for working with the software framework.
`e-eclipse:`	On some platforms, a Epiphany port for the Eclipse IDE is provided.

1.5   ESDK Installation

The eSDK is provided as a ready-to-install archive, downloadable from the Adapteva FTP site (ftp.adapteva.com, for registered customers) or from the Parallella FPT site (ftp.parallella.org, for Anonymous downloads). By default, the eSDK is installed at `/opt/adapteva`. Any other path can be set as well. After downloading the eSDK archive, extract the file in the `/opt/adapteva` directory (or other target directory) and create or update a symbolic link to the root of the eSDK release. You may need administrator privilege to use the following commands. Use "`sudo`" or equivalent as required. Following is the BASH command sequence:

```
$ REV="4.13.03.30"        # replace with the current revision
$ EDIR="/opt/adapteva"    # replace with preferred directory
$ ftp ...                 # get the eSDK archive tarball
$ mkdir -p ${EDIR}        # create the root directory
$ tar xvf esdk.${REV}.tgz -C ${EDIR} # extract tarball
$ ln -sTf esdk.${REV} ${EDIR}/esdk   # create a symlink to new eSDK
```

Next, library prerequisites have to be installed for the Epiphany compiler to work. On platforms supporting the Eclipse IDE, Java Run-Time library is required as well:

```
$ apt-get install libmpfr-dev libgmp3-dev libmpc-dev openjdk-6-jre
```

If you use a 64-bit PC for cross-build host applications for a 32-bit embedded platform, you may need to install a 32-bit layer support:

```
$ apt-get install ia32-libs
```

      REV 5.13.07.10

In order to use the eSDK, some environment variables have to be defined. Adding the following commands to your shell's login script will define these automatically:

```
$ echo "EPIPHANY_HOME=${EDIR}/esdk"  >> ~/.bashrc
$ echo '. ${EPIPHANY_HOME}/setup.sh' >> ~/.bashrc
```

1.6   Additional Documentation and Resources

This reference manual provides a brief overview of the extensive features available in the GNU distributions and Eclipse IDE. The documentation provided should be enough to enable the successful use of the Epiphany platform for most use cases. Advanced users who are looking for additional features should review the complete manuals found at the following locations:

GCC Compiler: http://gcc.gnu.org/onlinedocs/gcc

LD Linker: http://sourceware.org/binutils/docs/ld

GAS Assembler: http://sourceware.org/binutils/docs/as

ELF Utilities: http://sourceware.org/binutils/docs/binutils

GDB Debugger: http://sourceware.org/gdb/current/onlinedocs/gdb

Newlib Standard C Libraries: http://sourceware.org/newlib

Eclipse-Indigo IDE: http://help.eclipse.org/indigo/index.jsp

                     REV 5.13.07.10

2. Epiphany Multicore Development IDE (ECLIPSE)

Note: The Eclipse based Epiphany IDE is outdated and not in sync with the current generation of Epiphany platforms, which means platforms newer than the EMEK3 and EMEK4 boards. Some of the features presented in this chapter are not applicable to the ZYNQ based system such as Parallella Prototype (ZedBoard) and Parallella computers.

In particular, the debugging and launching of an Epiphany application from the Eclipse IDE is currently not supported. It is still a valuable tool for writing Epiphany applications, especially for the host-accelerator model.

Currently, the IDE is not supported on the ARM based platforms, so one needs an x86 (PC) based machine to use it.

REV 5.13.07.10

2.1   Overview

The Epiphany SDK IDE is based on Eclipse, a multi-language and multi-platform software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. The Eclipse development environment was started by IBM and subsequently released as open source in 2001 and is today one of the world's most popular development environments with millions of users. The Epiphany IDE is multicore ready, and includes all the features a programmer would expect from a state of the art IDE:

Creating, editing, navigating, and building C based projects

Integration with a GDB debugger

Multicore context viewing

Source level debugging

Figure 2.1 The Eclipse IDE

                   REV 5.13.07.10
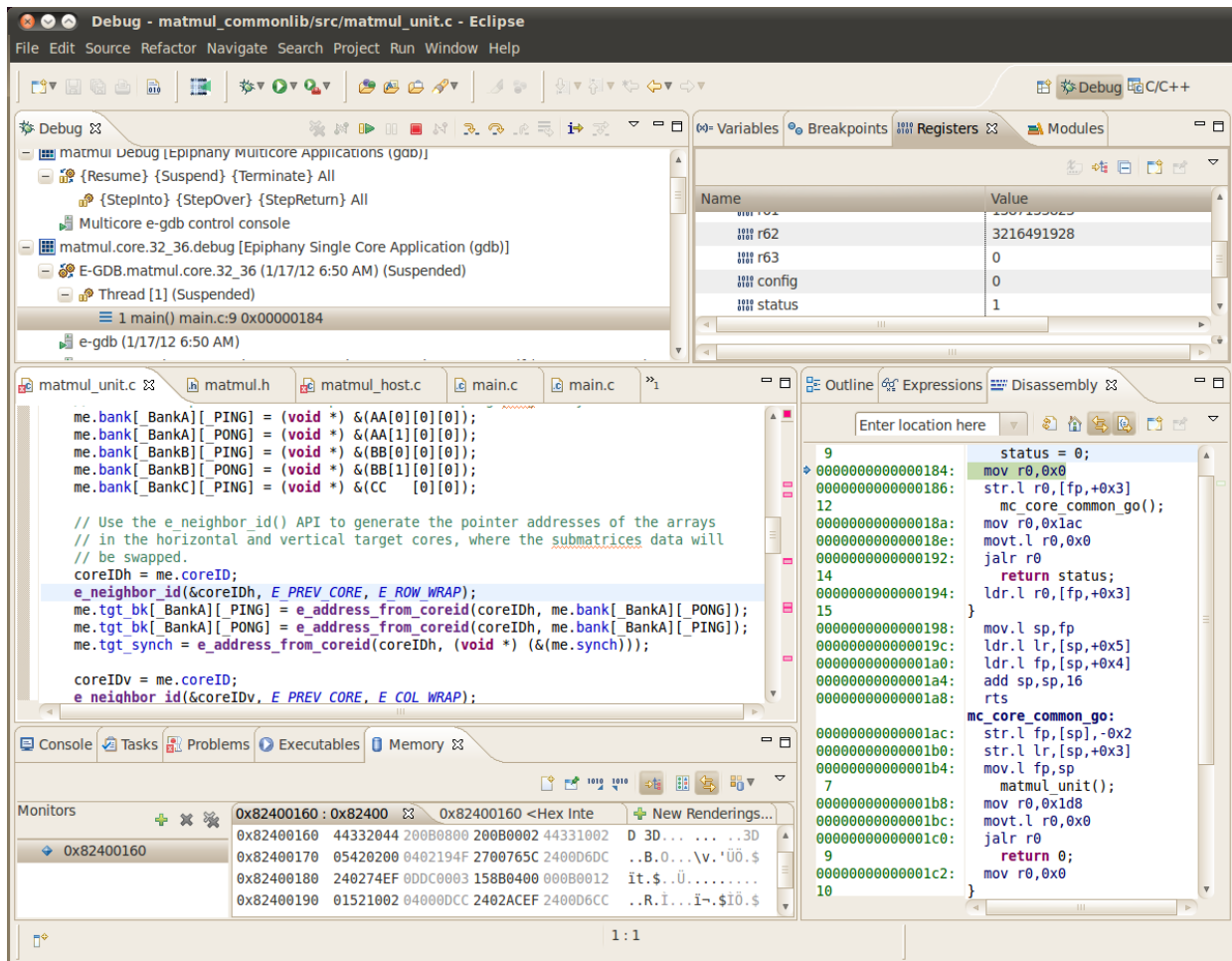
The rest of this chapter will go through a number of tutorials to explain the basic operations of Eclipse.

2.2 Epiphany IDE tutorial

Run the Eclipse environment by typing eclipse at the Linux prompt:

```
$ ${EPIPHANY_HOME}/e-eclipse &
```

The Eclipse window opens. Choose the workspace which is where your projects and environment will be saved:

Figure 2.2: The Workspace Launcher
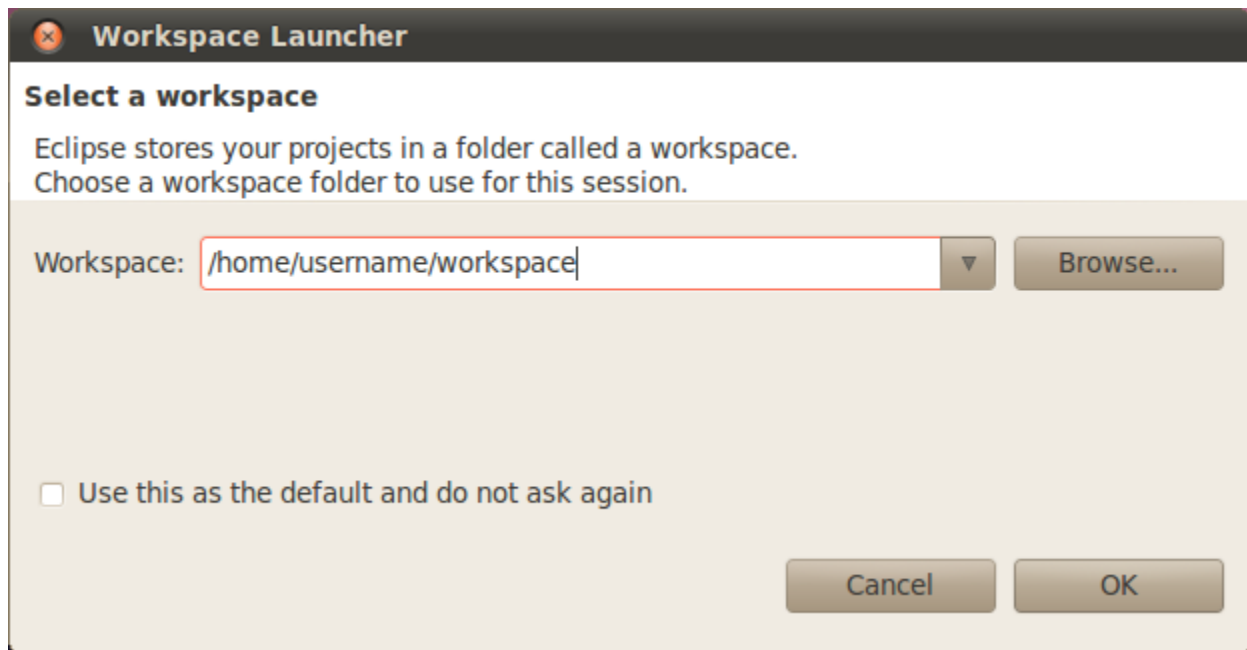
The Eclipse Welcome window opens, ready for making the first project. Select the Workbench icon to get to the project views.

     REV 5.13.07.10
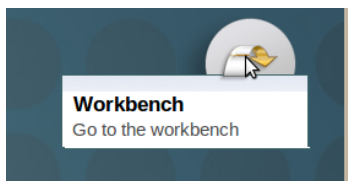
Creating a project

Next, select `File` → `New` → `C Project`. Choose a name for the new project in the `Project name` field and select the `Epiphany Executable (Multi Core)` → `Hello World Epiphany Multicore C-Project` type. The Epiphany tool chain is the only tool chain option for this type of project. This project will generate a simple multicore sample project that can be used as a template for building your system.
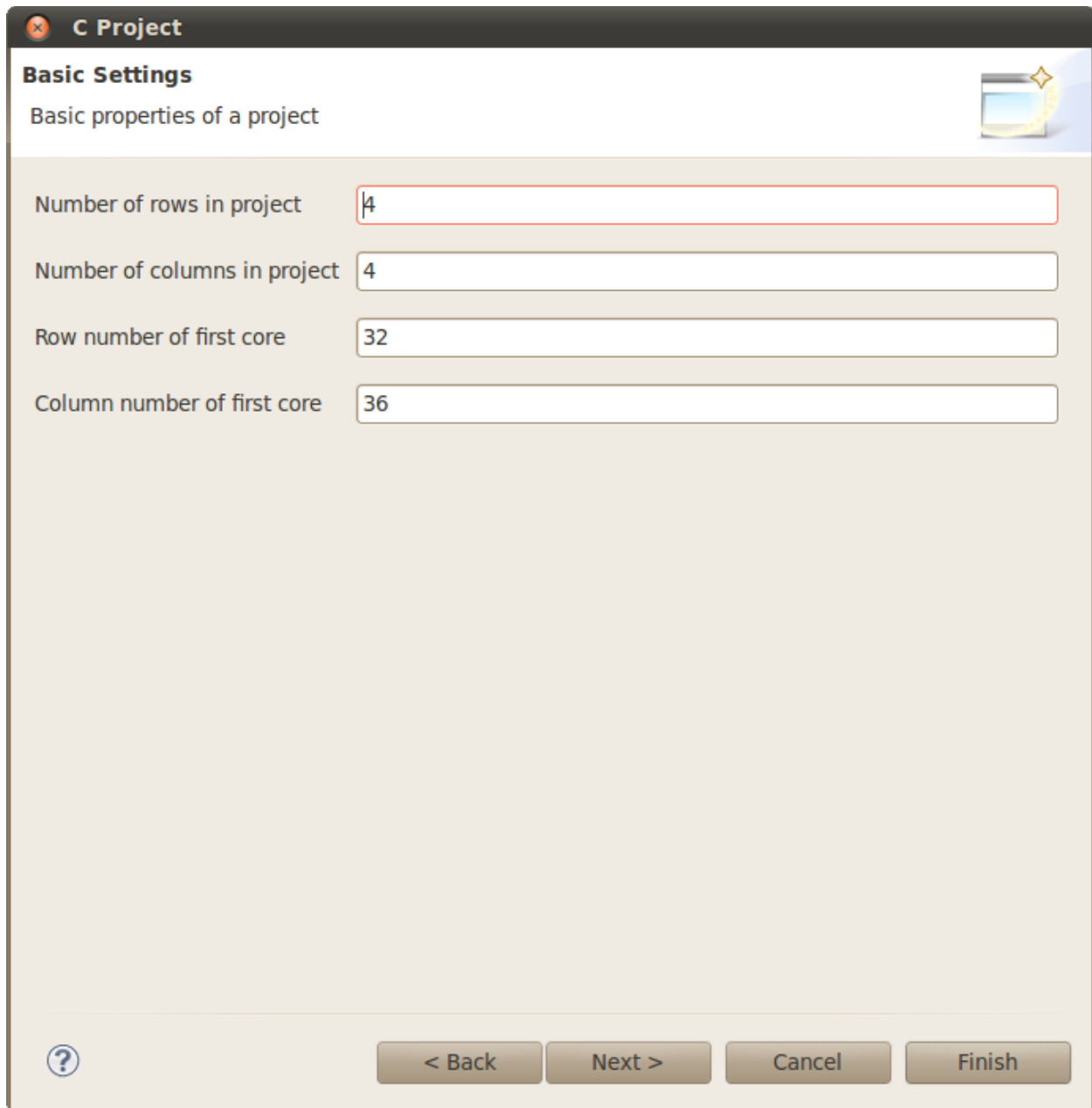
You can also choose an `Empty` or `Hello World ANSI C` project type under `Executable`. The Linux GCC tool chain is used for creating a Linux x86 program that can be run in a native Linux environment as a host application or as a software verification reference.

Figure 2.3: Project Definition Selection
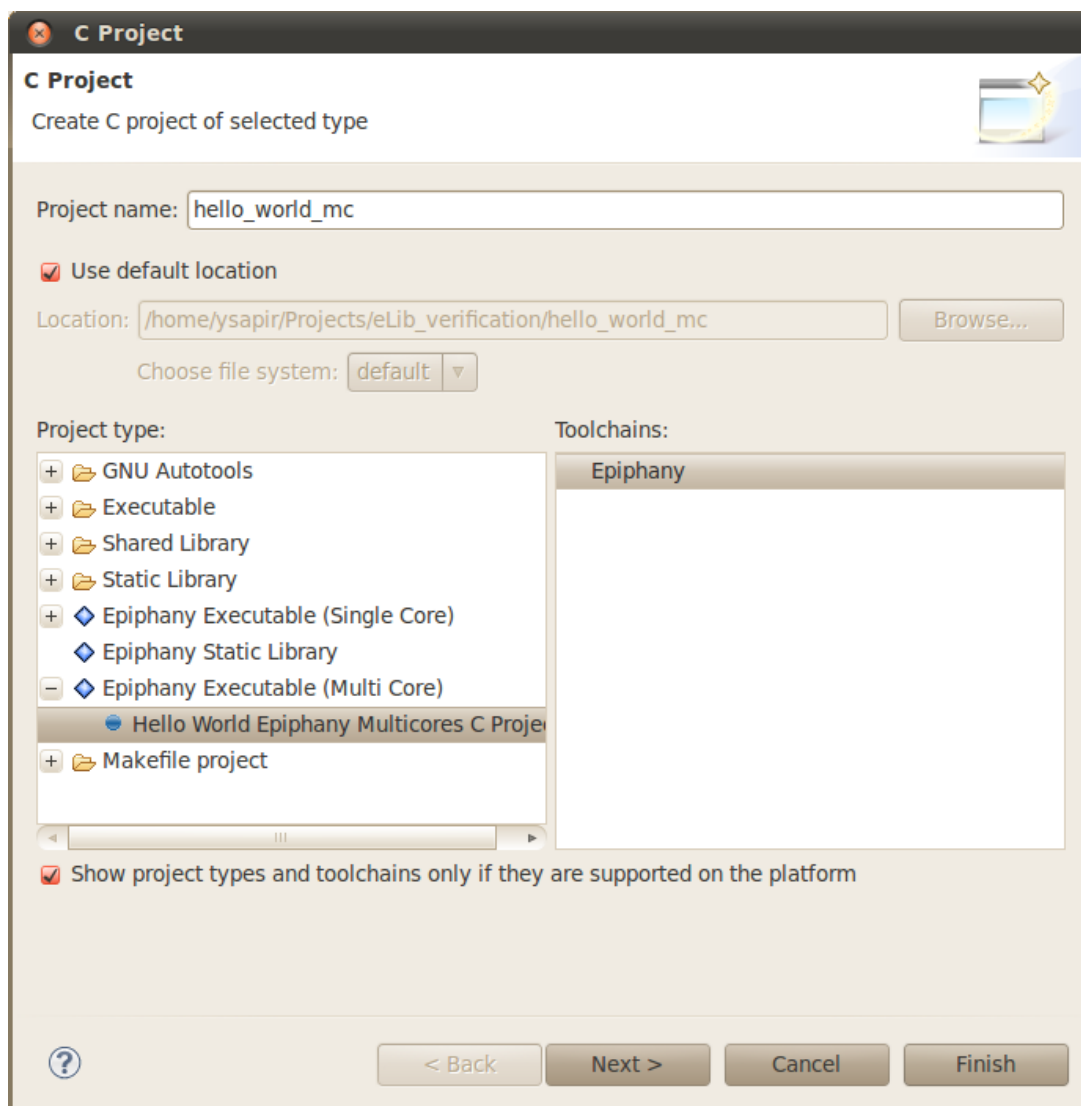
Press the `Next>` button and set the project's basic settings. The number of rows and number of columns set the size of the active portion of the Epiphany core matrix. The start row and start column set the address (2D index) of the first core in the array. Please refer to the board support package reference to get information regarding the actual memory map of a specific platform.

Figure 2.4: Application Target Cores Allocation

Press the `Next>` button twice to approve and then the `Finish` button. The wizard then creates the sample project. Depending on the configuration, this step can take some time to complete.

     REV 5.13.07.10

The number of per-core and the common projects are visible plus an extra, non-source master meta-project.

Eclipse Workbench work flow utilizes the Perspectives concept. A Perspective is the collection of Views and Editors comprising the current state of the Workbench and are related to a certain task (like code edit, application debug, etc.). When launched, the Workbench enters the C/C++ perspective which lets you edit code and build your application. Visible now is a simple multicore Hello World style project which is a skeleton for developing your system. You should add code which is common to all cores in the commonlib project. Code specific to a core should be added to the relevant core's project.

Figure 2.5: C/C++ Perspective for Multicore Application

        REV 5.13.07.10
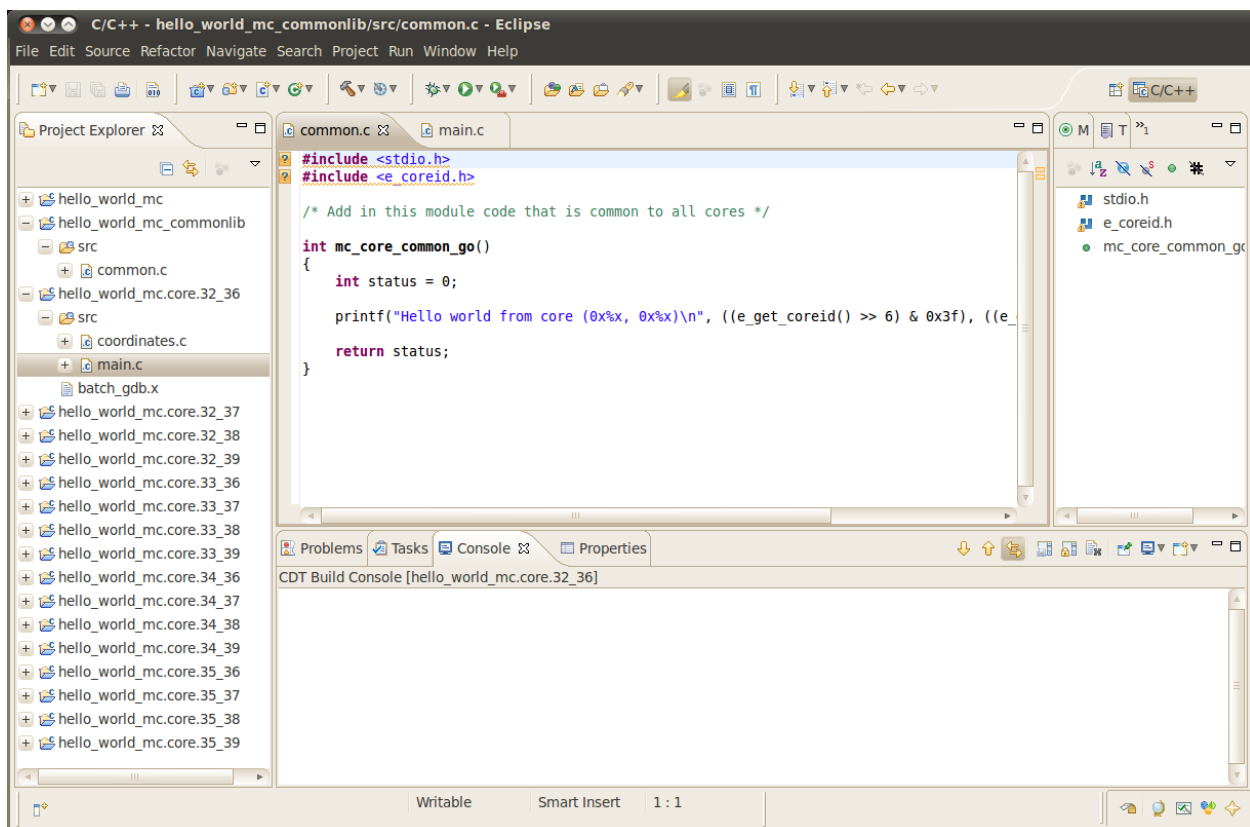
Building and executing

We recommend that you set the `Save automatically before build` option in the `Window` → `Preferences` → `General` → `Workspace` dialog before building the project. Also, before loading and executing an application, make sure the e-server is running, to allow Eclipse communicate with the hardware target.

You may need to set the memory layout of your application. Depending on the size of your code and data, you can place them in the external or internal memory. The newlib code (the Standard C library) and the program stack can be placed internally or externally as well. This can be done using special linker keywords, as described in chapter 5.5. However, the fastest and recommended way is to choose one of the three predefined linker description files that are provided with the hardware target package. Set the LDF option in by selecting one of the per-core projects and from the menu `Project` → `Properties` → `C/C++ Build` → `Settings` → `linker description file` browse for the required `*.ldf` file, located in the `bsps` directory of the eSDK installation. Then, apply the settings to the other core projects by right-clicking on the core project and selecting the action from the context menu.

You can now build and run the application. Select `Project` → `Build All` to build all the projects. To simply load and run the project on the hardware, select the meta-project, then select menu `Run` → `Run Configurations…` to create the context. Right-click on the `Epiphany Multicore Application (loader)` and select `New`. The application item appears like in Figure 2.6. Select the application and press `Run`. The application will now be loaded on the target via the loader, the target will be reset and the application will run.

Debugging a project

After building the project, as described above, select menu `Run` → `Debug Configurations….`. Right-click on the `Epiphany Multicore Applications (gdb)` and select `New`. The application item appears. Select the application item. Make sure that in the `Projects` tab, the `Enable auto build` option is selected and press `Debug`. The Perspective should now change to the `Debug` perspective. Depending on a previous Eclipse configuration, the perspective will not change. You can do this manually by pressing the `Open Perspective` button at the top-right corner.

     REV 5.13.07.10

Figure 2.6: Run Configurations Settings
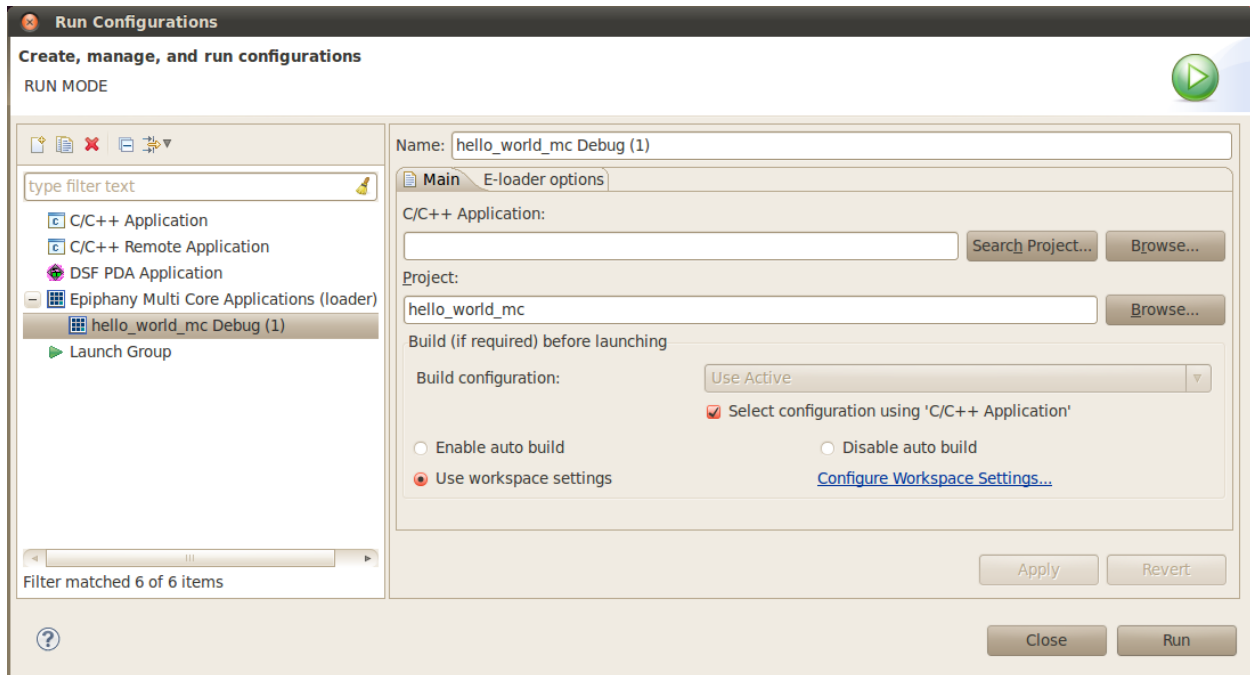
Figure 2.7: Debug Configuration Settings

     REV 5.13.07.10
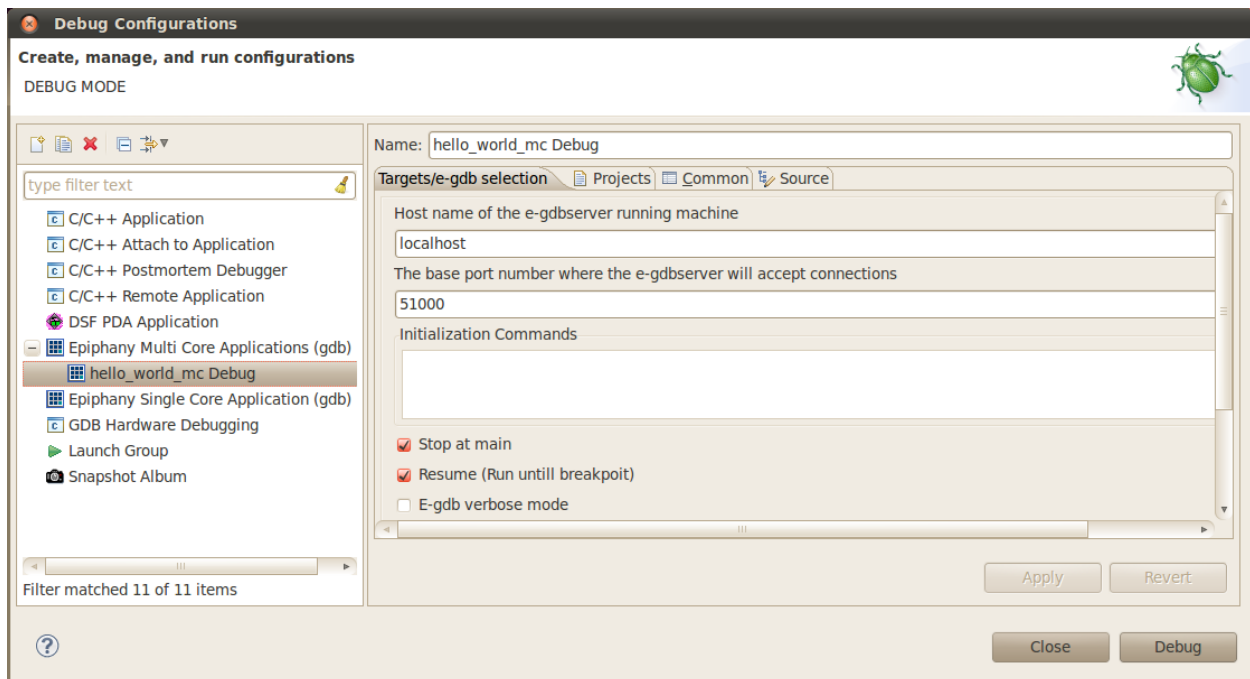
Figure 2.8: Debug Perspective
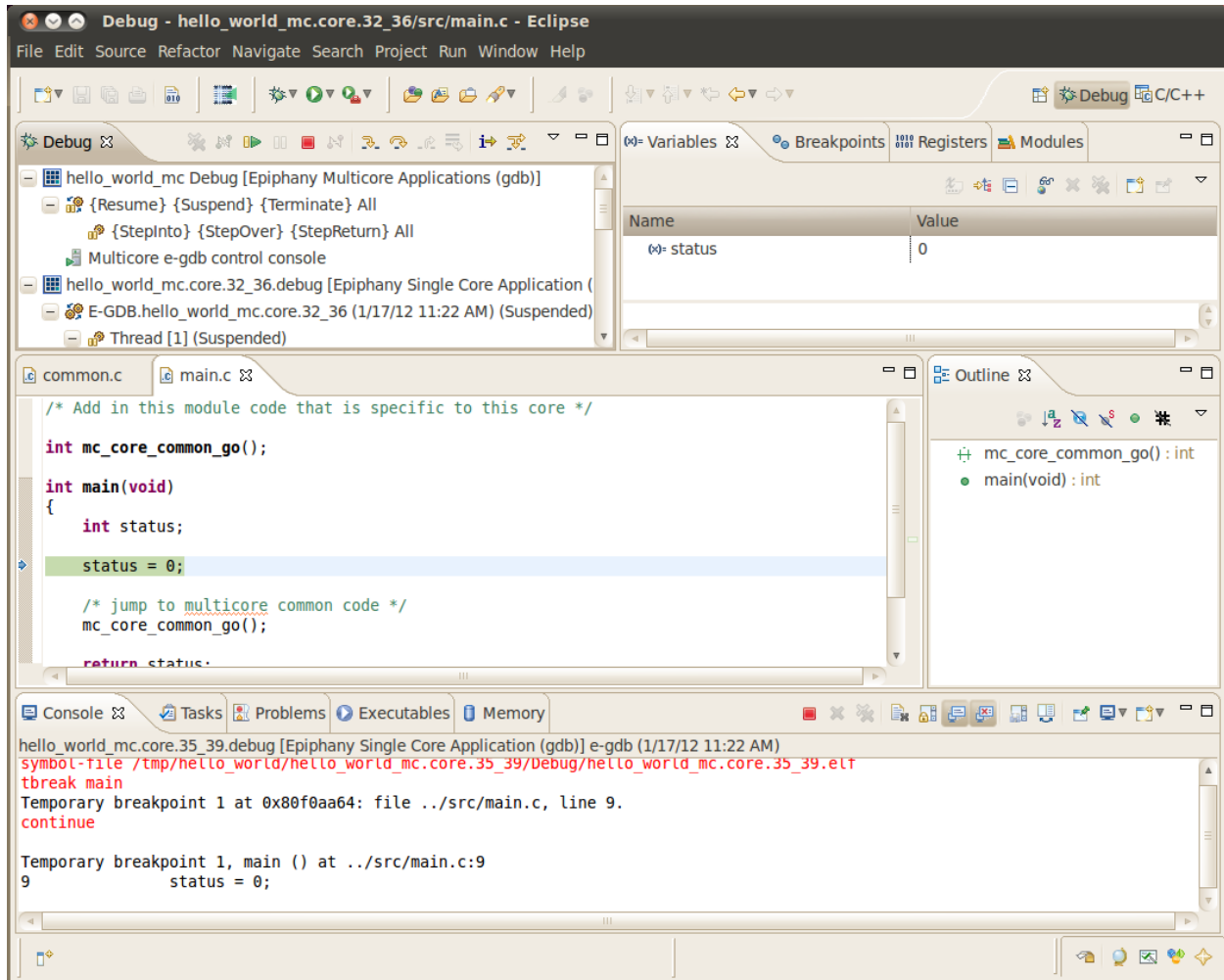
In the Debug view you can see a session for the meta-project and the sessions for the multiple core projects. Each core session includes levels for Target, Threads and Contexts. The first thread is intended for interactive GUI style debugging. The e-gdb thread is intended for Console style debugging. The threads work in conjunction with each-other and actions taken in one affect the other.

In the meta-project session there are two levels as well, where the Target level lets you Resume, Suspend and Terminate all core projects and the Thread level lets you perform the regular step-wise debug operations. Note that multiple Threads and Contexts can be selected (by pressing the Ctrl key while selecting the items) and the debug operations will take place on the multiple

 REV 5.13.07.10

selected sessions. Breakpoints that are set in the common source files will affect all projects, and the execution of each thread will be halted when the breakpoint is reached.

If the application includes standard output and input function calls (like `printf()`), the interaction is made through the Console view. Select the associated `e-gdb` process from the Debug view and the relevant Console becomes active.

Debugging a single core of a multicore project

It is sometimes required, during a development of the application, to debug just one core. In this case, select the binary executable (*.elf file) of the core you want to debug. Right-click and choose `Debug As >` → `Debug Configurations…`. Right-click on `Epiphany Single Core Application (gdb)` and select `New`. Select the new item. You should now change the `Port Number` to the one that is associated with the core and press the `Debug` button.

Note: Debugging and launching an Epiphany application from the Eclipse IDE is currently not supported on all platforms.

       REV 5.13.07.10

Creating a single core application

If the required application should be developed and run on a single core, you can choose the `Epiphany Executable (Single Core)` type from the Project Definition Selection dialog (Figure 2.3) and continue with the similar steps to create, edit, build and debug the application.

Figure 2.9: C/C++ Perspective for Single Core Application

REV 5.13.07.10
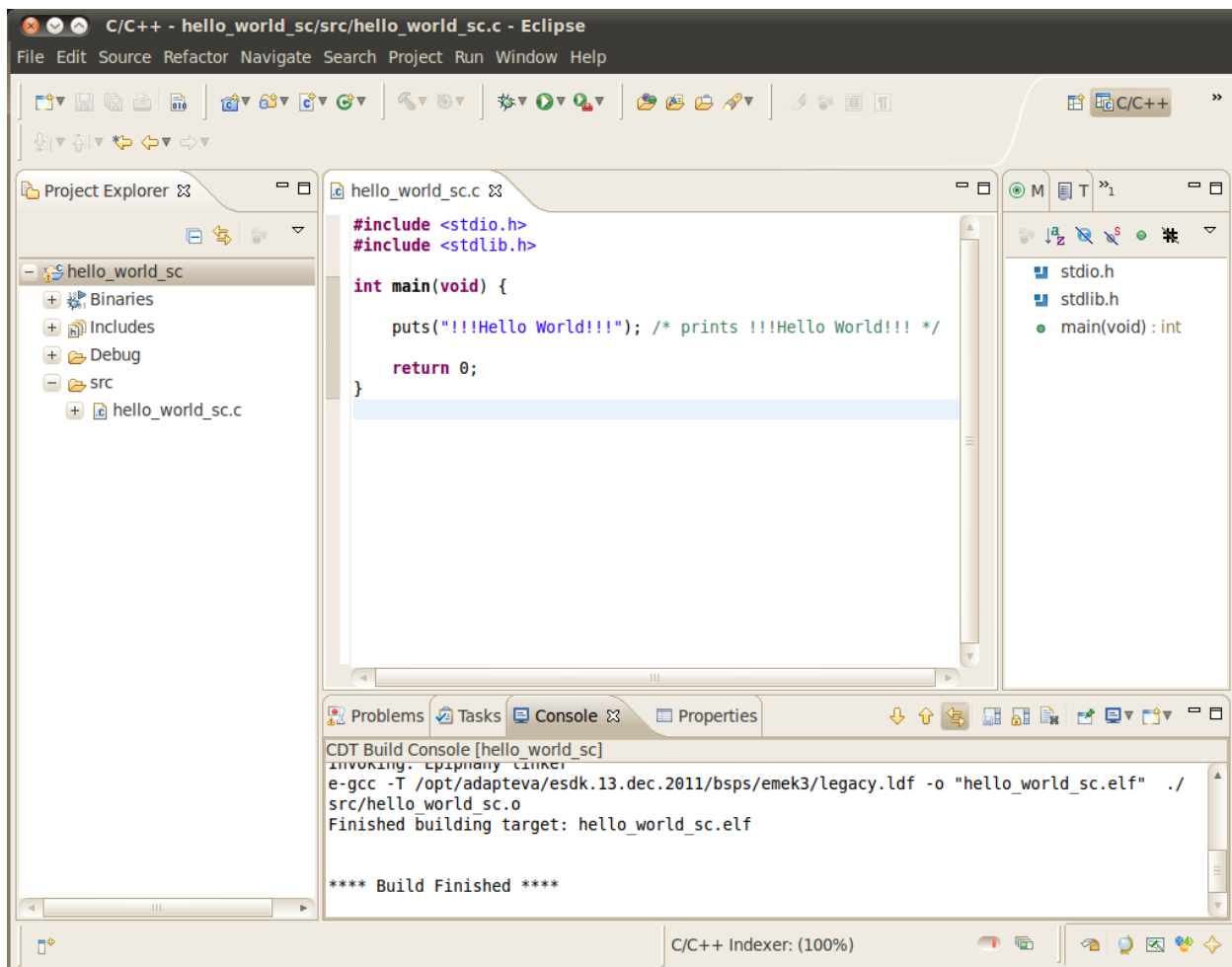
2.3   Online Eclipse Help for Common Tasks

The following topics contain links to further online help guides for some useful general tasks supported by the Eclipse development environment:

Adding Include Paths and Symbols:

http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.user/tasks/cdt_t_proj_paths.htm

Customizing the C/C++ editor:

http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.user/tasks/cdt_t_cust_cpp_editor.htm

Navigating C/C++ declarations:

http://help.eclipse.org/helios/topic/org.eclipse.cdt.doc.user/tasks/cdt_t_open_declarations.htm

Refactoring C/C++ Code:

http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.user/tasks/cdt_t_refactoring.htm

Debugging a program:

http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.user/tasks/cdt_o_debug.htm

     REV 5.13.07.10

2.4 Updating the Eclipse Installation

Updating the Eclipse installation can be done in two ways – manual and automatic. Manual mode involves downloading a plugins pack and extracting it in the `eclipse/plugin` directory. In order to have it done automatically, use Eclipse's Install New Software feature available from the Help menu. A dialog like this opens:

Figure 2.10: Install New Software Dialog

     REV 5.13.07.10
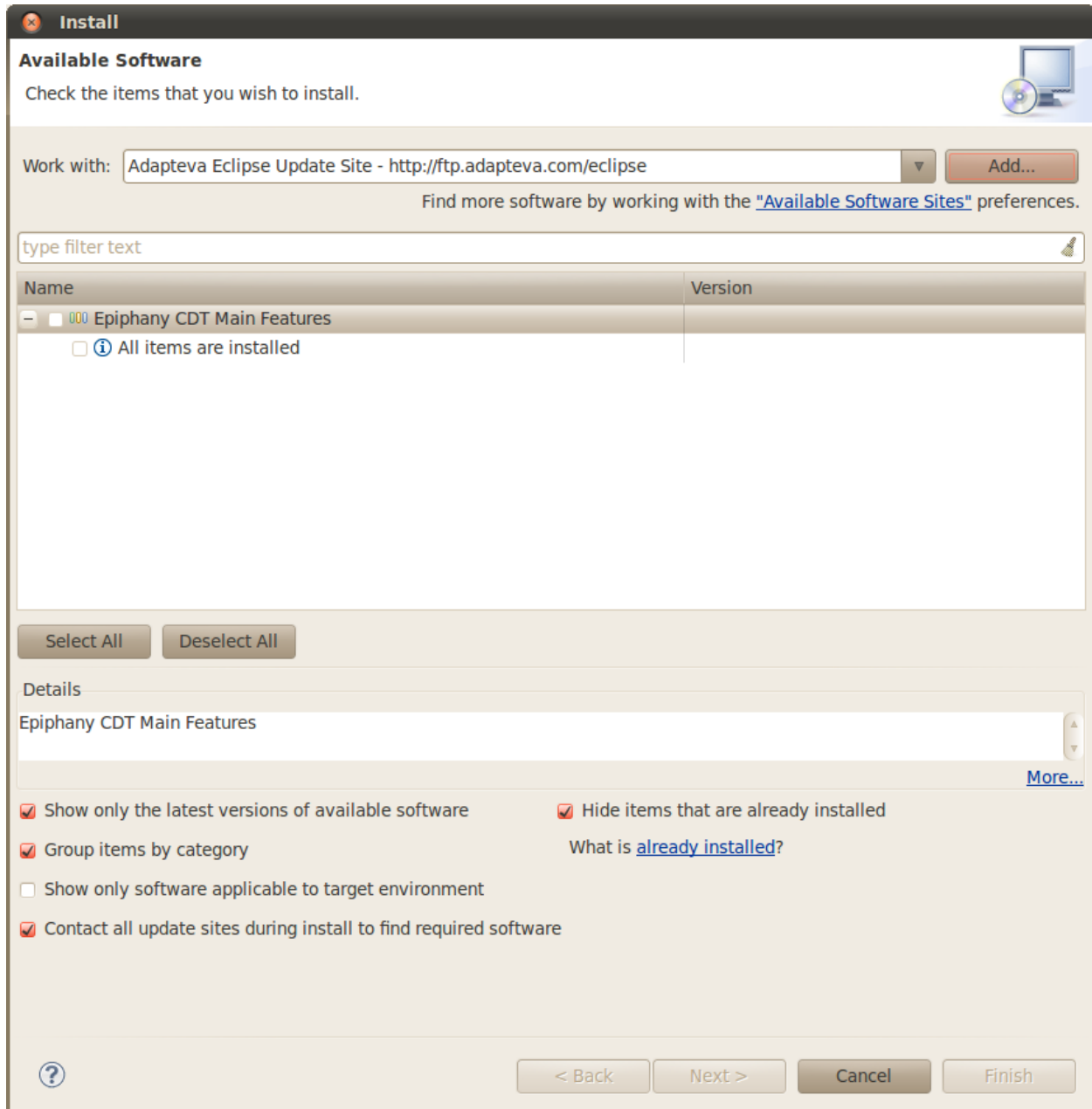
Click `Add…` and fill in "Adapteva Eclipse Update Site" for the name and `http://ftp/adapteva.com/eclipse` for the location. Then, a list of available updates appears. Check the optional checkboxes to filter out irrelevant updates and select the required update. Then click `Next>` to confirm.

     REV 5.13.07.10

3. C/C++ Compiler (E-GCC)

3.1   Overview

This chapter gives a high level overview of the Epiphany compiler, based on the popular GNU GCC compiler. The purpose of this chapter is to summarize the important features and to document additional features that are not included in the baseline GNU distribution.

The GCC compiler supports the following versions of C/C++:

- ISO/IEC 9899:1990 (C89)

- ISO/IEC 9899:1999 (C99)

- ISO/IEC 9899:2011 (C11) *(partial)*

- ISO/IEC 14882:1998 (C++98)

- ISO/IEC 14882:2011 (C++11) *(partial)*

Support for C99 is substantial but not 100% complete. For a review of the minor C99 features not supported in GCC, please refer to: http://gcc.gnu.org/c99status.html. Similarly, C11 and C++11 are new standards and their partial is partial. More info here: http://gcc.gnu.org/onlinedocs/gcc/Standards.html.

3.2   Simple Example

The following example shows how to use the compiler to create an executable from a simple program source file without any optimization.

```
$ e-gcc hello_world.c -o hello_world.elf
```

The following example shows some the usage of some of the compilation switches used to produce executables highly optimized for speed at the expense of code size.

```
$ e-gcc my_fft.c -o my_fft.elf -O3 -ftree-vectorize -funroll-loops \
    -mfp-mode=round-nearest -mfused-madd -ffast-math
```

     REV 5.13.07.10

3.3 Compiler Command-line Options

The GCC compiler supports a wide range of options allowing for fine grain compilation process.

The following tables summarize the compiler most commonly used options grouped by type.

Table 3.1: General Compiler Options

Option	Function
-c	Compile or assemble source code, but do not link
-S	Stop after compilation proper; do not assemble
-E	Stop after the preprocessing stage
-o *file*	Place output in file *file*.
-x *language*	Specify explicit *language* of input file rather than letting GCC determine language based on file name suffix. Valid values for language are: 'c', 'c++'
-std=*standard*	Determine the language standard. To support C99, specify "-std=c99"
-D name=*definition*	The contents of *definition* are tokenized and processed as if they appeared in a '#define' directive.
-Wa,*option*	Pass option as an option to the assembler.
-l*library* -l *library*	Search the library named *library* when linking. It makes a difference where in the command you write this option; the linker searches and processes libraries and object files in the order they are specified. The directories searched include several standard system directories plus any that you specify with '-L'.
--version	Print the version number of the compiler
-I*dir*	Add the directory *dir* to the list of directories to be searched for header files.
-L*dir*	Add directory *dir* to the list of directories to be searched for '-l'.
--help	Print (on the standard output) a description of the command line options understood by gcc.
--help=class	Print (on the standard output) a description of the command line options understood by the compiler that fit into a specific class. The class can be one of 'optimizers', 'warnings', 'target', 'params', or 'language'.
@file	Read command-line options from file.

       REV 5.13.07.10

Table 3.2: Warning Options

Option	Function
-w	Inhibit all warning messages
-Werror	Make all warnings into errors
-Werror=	Make the specified warning into an error
-fsyntax-only	Check the code for syntax errors, but don't do anything beyond that.
-Wfatal-errors	This option causes the compiler to abort compilation on the first error occurred rather than trying to keep going and printing further error messages.
-pedantic	Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++
-Wall	This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid. We highly recommend running with this switch turned on for new code. The switch turns on the following warnings: -Waddress -Warray-bounds (only with '-O2') -Wc++0x-compat -Wchar-subscripts -Wimplicit-int -Wimplicit-function-declaration -Wcomment -Wformat -Wmain (only for C/ObjC and unless '-ffreestanding') -Wmissing-braces -Wnonnull -Wparentheses -Wpointer-sign -Wreorder -Wreturn-type -Wsequence-point -Wsign-compare (only in C++) -Wstrict-aliasing -Wstrict-overflow=1 -Wswitch -Wtrigraphs -Wuninitialized (only with '-O1' and above) -Wunknown-pragmas -Wunused-function -Wunused-label -Wunused-value -Wunused-variable

Table 3.3: Debug Options

Option	Function
-g	Produce debugging information in stabs format.
-pg	Generate extra code to write profile information suitable for the analysis program gprof.

Table 3.4: Linker Options

Option	Function
-nostartfiles	Do not use the standard system startup files when linking.
-nodefaultlibs	Do not use the standard system libraries when linking.
-nostdlib	Do not use the standard system startup files or libraries when linking.

 REV 5.13.07.10

-s	Remove all symbol table and relocation information from the executable.
-static	On systems that support dynamic linking, this prevents linking with the shared libraries. On other systems, this option has no effect.
-shared	Produce a shared object which can then be linked with other objects to form an executable.
-Xlinker *option*	Pass option as an option to the linker.

Table 3.5: Optimization Options

Option	Function
-O0	Reduce compilation time and make debugging produce the expected results. This is the default.
-O1	'-O or –O1' turns on the following optimization flags: -fauto-inc-dec -fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fguess-branch-probability -fif-conversion2 -fif-conversion -finline-small-functions -fipa-pure-const -fipa-reference -fmerge-constants -fsplit-wide-types -ftree-ccp -ftree-ch -ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-fre -ftree-sra -ftree-ter -funit-at-a-time '-O' also turns on '-fomit-frame-pointer'
-O2	GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. It turns on all optimization flags in O1 and the following additional flags: -fthread-jumps -falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize -fexpensive-optimizations -fgcse -fgcse-lm -finline-small-functions -findirect-inlining -fipa-sra -foptimize-sibling-calls -fpartial-inlining -fpeephole2 -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec

	-fschedule-insns -fschedule-insns2 -fstrict-aliasing -fstrict-overflow -ftree-switch-conversion -ftree-pre -ftree-vrp
-O3	Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on: -finline-functions -funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-vectorize -fipa-cp-clone
-Os	Enables all optimization switches of O2, except for the following: -falign-functions -falign-jumps -falign-loops -falign-labels -freorder-blocks -freorder-blocks-and-partition -fprefetch-loop-arrays -ftree-vect-loop-version
-funroll-loops	Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop.
--param name=value	In some places, GCC uses various constants to control the amount of optimization that is done. **max-unrolled-insns:** The maximum number of instructions that a loop should have if that loop is unrolled, and if the loop is unrolled, it determines how many times the loop code is unrolled. **max-unroll-times:** The maximum number of unrollings of a single loop.

Table 3.6: Floating Point Math Options

Option	Function
-fsingle-precision-constant	Treat floating point constant as single precision constant instead of implicitly converting it to double precision constant.
-funsafe-math-optimizations	Enables '-fno-signed-zeros', '-fno-trapping-math', '-fassociative-math' and '-freciprocal-math'.
-freciprocal-math	Allow the reciprocal of a value to be used instead of dividing by the value if this enables optimizations.
-ffast-math	Sets '-fno-math-errno', '-funsafe-math-optimizations', '-ffinite-math-only', '-fno-rounding-math', '-fno-signaling-nans' and '-fcx-limited-range'. This option causes the preprocessor macro __FAST_MATH__ to be defined
-fno-trapping-math	Compile code assuming that floating-point operations cannot generate user visible traps.
-fno-signed-zeros	Allow optimizations for floating point arithmetic that ignore the signedness of zero.

 REV 5.13.07.10

Option	Function
-mprefer-short-insn-regs	Preferentially allocate registers that allow short instruction generation. This can result in increased instruction count, so if this reduces or increases code size might vary from case to case.
-mbranch-cost=*num*	Set the cost of branches to roughly *num* "simple" instructions. This cost is only a heuristic and is not guaranteed to produce consistent results across releases.
-mcmove	Enable the generation of conditional moves.
-mnops=*num*	Emit *num* nops before every other generated instruction.
-mno-soft-cmpsf	For single-precision floating point comparisons, emit an fsub instruction and test the flags. This is faster than a software comparison, but can get incorrect results in the presence of NaNs, or when two different small numbers are compared such that their difference is calculated as zero. The default is -msoft-cmpsf, which uses slower, but IEEE-compliant, software comparisons.
-mno-round-nearest	Make the scheduler assume that the rounding mode has been set to truncating. The default is -mround-nearest.
-mlong-calls	If not otherwise specified by an attribute, assume all calls might be beyond the offset range of the b / bl instructions, and therefore load the function address into a register before performing a (otherwise direct) call. This is the default.
-mshort-calls	If not otherwise specified by an attribute, assume all direct calls are in the range of the B/BL instructions, so use these instructions for direct calls. The default is -mlong-calls.
-msmall16	Assume addresses can be loaded as 16 bit unsigned values. This does not apply to function addresses for which -mlong-calls semantics are in effect.
-mfp-mode=*mode*	Set the prevailing mode of the floating point unit. This determines the floating point mode that is provided and expected at function call and return time. Making this mode match the mode you predominantly need at function start can make your programs smaller and faster by avoiding unnecessary mode switches. *mode* can be set to one the following values: `caller' Any mode at function entry is valid, and retained or restored

     REV 5.13.07.10

when the function returns, and when it calls other functions. This mode is useful for compiling libraries or other compilation units you might want to incorporate into different programs with different prevailing FPU modes, and the convenience of being able to use a single object file outweighs the size and speed overhead for any extra mode switching that might be needed, compared with what would be needed with a more specific choice of prevailing FPU mode.

`truncate'

This is the mode used for floating point calculations with truncating (i.e. round towards zero) rounding mode. That includes conversion from floating point to integer.

`round-nearest'

This is the mode used for floating point calculations with round-to-nearest-or-even rounding mode.

`int'

This is the mode used to perform integer calculations in the FPU, e.g. integer multiply, or integer multiply-and-accumulate.

-mno-postmodify	Code generation tweaks that disable, respectively, splitting of 32 bit loads, generation of post-increment addresses, and generation of post-modify addresses. The defaults are msplit-lohi, -mpost-inc, and -mpost-modify.
-mnovect-double	Change the preferred SIMD mode to SImode. The default is -mvect-double, which uses DImode as preferred SIMD mode.
-max-vect-align=*num*	The maximum alignment for SIMD vector mode types. *num* may be 4 or 8. The default is 8.
-m1reg-*reg*	Specify a register to hold the constant −1, which makes loading small negative constants and certain bitmasks faster. Allowable values for reg are r43 and r63, which specify to use that register as a fixed register, and none, which means that no register is used for this purpose. The default is -m1reg-none.

      REV 5.13.07.10

For the Epiphany architecture, some useful optimization options are `-falign-loops=8` and `-falign-functions=8`. These options direct the compiler to generate code, where the first instructions in the body of a loop or in a function are double-word aligned. Thus, the processor's alignment buffer is kept full from the first fetch after a branch (pipeline flush). `-ffast-math` can make math calculation faster by omitting some floating-point calculation restrictions and relaxing the standard conformance.

3.4 GNU Function Attributes

In GCC, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully. The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses:

alias	**aligned**	**alloc_size**
always_inline	**artificial**	**cold**
const	**constructor**	**deprecated**
destructor	**error**	**externally_visible**
flatten	**format**	**format_arg**
gnu_inline	**hot**	**malloc**
noreturn	**returns_twice**	**noinline**
nonnull	**nothrow**	**pure**
sentinel	**no_instrument_function**	**section**
used	**unused**	**weak**
warn_unused_result	**warning**	

3.5 Epiphany Specific Compiler Attributes

The Epiphany compiler also supports a number of attributes used by the linker descriptor file for easy and explicit memory management. These attributes can be found in the linker chapter of this manual.

    REV 5.13.07.10

4. Assembler (E-AS)

4.1 Overview

The Epiphany assembler, 'e-as', parses a file of assembly code to produce an object file for use by the linker 'e-ld'.

4.2 Simple Example

The following example shows how to use the assembler to create an object file.

```
$ e-as my.s -o my-object-file.o
```

4.3 Command Line Options

Table 4.1: Assembler Command Line Options

Option	Function
-o *file*	Create object file with name *file*.
-W	Suppress warning messages
--version	Print the version number of the assembler
-I*dir*	Add the directory *dir* to the list of directories to be searched for .include files
-g	Generate debugging information
-L	Keep local symbols
--help	Print (on the standard output) a description of the command line options

4.4 General Syntax

The 'e-as' assembler syntax closely follows the "AT&T" style of assembly programming. Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read.

There are four ways of rendering comments to 'e-as':

Anything from '/*' through the next '*/' is a comment. There is no nesting of comments

Anything to the right of the character ';' is a comment

Anything to the right of the character '#' is a comment

Anything to the right of '//' is a comment

     REV 5.13.07.10

A symbol is one or more characters chosen from the set of all letters (both upper and lower case), digits and underscore '_'.

A statement ends at a newline character ('\n') or line separator character. The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements. It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

You may write a statement on more than one line if you put a backslash ('\') immediately in front of any newlines within the statement. When 'e-as' reads a backslashed newline both characters are ignored. You can even put back-slashed newlines in the middle of symbol names without changing the meaning of your source program.

A constant is a number, written so that its value is known by inspection, without knowing any context, as shown in the examples below:

```
.byte 74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.

.ascii "Ring the bell\7" # A string constant.

.octa 0x123456789abcdef0123456789ABCDEF0 # A bignum.

.float 0f-31415926535E-10 # (-PI), a flonum.
```

A string is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to escape these characters: precede them with a backslash '\' character. For example '\\' represents one backslash: the first '\' is an escape which tells as to interpret the second character literally as a backslash (which prevents 'e-as' from recognizing the second '\' as an escape character).

Integer values can be expressed in multiple formats for ease of use:
A binary integer is '0b' or '0B' followed by zero or more of the binary digits '01'.

An octal integer is '0' followed by zero or more of the octal digits ('01234567').

 REV 5.13.07.10

A decimal integer starts with a non-zero digit followed by zero or more digits (`0123456789`).

A hexadecimal integer is `0x` or `0X` followed by one or more hexadecimal digits chosen from `0123456789abcdefABCDEF`.

Integers have the usual values. To denote a negative integer, use the unary minus operator `-`.

     REV 5.13.07.10

4.5   Assembler Syntax Reference

Table 4.2: Assembler Control Directives

Directive	Description	Syntax	Example
.include	Include a file	.include "file"	.include "bsp0.inc"
.balignw	Halfword align the following code to *alignment* byte boundary (**default=4**). Fill skipped words with *fill* (**default=0**).	.balignw {alignment} {,fill}	.balignw 8,0x01a2
.balignl	Word align the following code to *alignment* byte boundary (**default=4**). Fill skipped words with *fill* (**default=0**).	.balignl {alignment} {,fill}	.balignl 8,0x01a2
.end	Marks the end of the assembly file. Data following this directive is not processed.	.end	.end
.fail	Generates errors or warnings during assembly.	.fail *expr*	.fail 1
.err	Generate an error during assembly.	.err	.err
.print	Print a string to standard output during assembly.	.print *string*	.print "Assembly error"
.section	Tell the assembler to assemble the following in section *expr* . *expr* can be either **.text**, **.data**, **.bss**, or any symbol described in the linker description file	.section *expr*	.section data
.text	Tells assembler to process the following as 'text' and tells linker to place it in data memory area	.text {subsection}	.text
.data	Tells assembler to process the following as 'data' and tells linker to place it in data memory area	.data {subsection}	.data
.bss	Tells assembler to process the following as 'bss 'and to place it in bss memory area	.bss {subsection}	.bss
.struct	Tells assembler to assemble the following in an absolute section	.struct *expr*	.struct 0 field1: .struct field1+4 field2:
.org	Following code is inserted at the start of the specified section plus the offset specified as *new-lc*.	.org *new-lc*	.org 0x2000

     REV 5.13.07.10

Table 4.3: Assembler Symbol Directives

Directive	Description	Syntax	Example
.equ | .set	Set value of symbol equal to *expr*	.set *symbol* , *expr*	.set Version, "0.1"
.equiv	Sets the value of symbol equal to *expr* and generates an error if it was previously defined.	.equiv *symbol, expr*	.equiv Version, "0.1"
.global	Makes symbol visible to linker	.global *symbol*	.global MyFunc

Table 4.4: Assembler Constant Directives

Directive	Description	Syntax	Example
.byte	Define byte *expr* (8 bits) in memory	.byte expr{,…}	.byte 25,0x11,031
.hword | .short	Define hword *expr* (32 bits) in memory	.hword expr{,…}	.hword 0x5
.word | .int | .long	Define word *expr* (32 bits) in memory	.word expr{,…}	.word 0x32
.ascii	Define *string*, non-zero terminated array of bytes.	.ascii *expr {....}*	.ascii "Hello"
.asciz | .string	Define *string*, zero terminated array of bytes.	.string *expr {....}*	.string "Hello World!\"
.float	Define 32bit IEEE number in memory	.float *expr*	.float 0f3.14,0f359.1
.double	Define 64bit IEEE number in memory	.double *expr*	.double 0f2e1
.fill	Generate *repeat* copies of *value* that is of size *size and*	.fill *repeat* {*,size* } {*,value*}	.fill 32,4,0xffffffff

Table 4.5: Assembler Looping Directives

Directive	Description	Syntax	Example
.rept	Repeat the sequence of lines between .rept end .endr *count* number of times	.rept *count*	.rept 10
.endr	Ends .rept sequence	.endr	.endr

 REV 5.13.07.10

Table 4.6: Assembler Conditional Directives

Directive	Description	Syntax	Example
.if	Assembles if absolute expression is not zero	.if *{absolute_expr}*	.if(2+4)
.elseif	Used in conjunction with .if	.elseif *{absolute_expr}*	.elseif(2+3)-5
.else	Used in conjunction with .if	.else	.else
.endif	Ends an .if block	.endif	.endif
.ifdef	Assembles if *symbol* exists	.ifdef *symbol*	.ifdef _my_test_
.ifndef	Assembles if *symbol* does not exists	.ifndef *symbol*	.ifndef _my_test_
.ifc	Assemble if *strings* are the same	.ifc *string1,string2*	.ifc "str1","str2"
.ifnc	Assemble if *strings* are not the same	.ifnc *string1,string2*	.ifnc "str1","str2"

Table 4.7: Assembler Macro Directives

Directive	Description	Syntax	Example
.macro	Defines a macro. • A macro can be defined without arguments • Arguments can be accessed by name • Macros can be accessed as ordered list or by reference arguments	..macro *name {args}*	Definition: .macro ArgMacro arg1,arg2 Use: ArgMacro 10,11 ArgMacro arg2=11,arg1=10
.endm	Marks the end of a macro	.endm	.endm
\@	Pseudo variable that contains the macro number . Can be used to generate a unique number on every macro definition	\@	MyLable@

Table 4.8: Assembler Digit Encoding

Type	Base	Prefix	Digits	Example
Decimal Integer	10		0-9	67
Hexadecimal Integer	16	0x or 0X	0-9,A-F	0x43
Octal Integer	8	0	0-7	083
Binary Integer	2	0b or 0B	0-1	0b01000011
Floating Point Number	10	0f or 0F	0-9	0f0.67e+2
Character	n/a	'	ASCII Symbol	'C
String	n/a	" and "	ASCII Symbol(s)	"Sixty Seven"

        REV 5.13.07.10

Table 4.9: Assembler Expression Operators

Operation	Symbol	Precedence
Negate	−	Highest
Compliment	~	
Multiplication	*	
Division	/	
Remainder	%	
Left Shift	<< or <	
Right Shift	>> or >	
Bitwise OR	|	
Bitwise AND	&	
Bitwise XOR	^	
Bitwise OR-NOT	!	
Addition	+	Lowest
Subtraction	−	

     REV 5.13.07.10

5. Linker (E-LD)

5.1 Overview

The Epiphany linker 'e-ld' combines a number of objects and archives, relocates their data and resolves symbol references. The following section gives a brief overview of the operations of the linker.

5.2 Simple Examples

The following example shows how to use the linker to create an elf executable from an object file using the default linker file for simulation using the Epiphany instruction set simulator.

```
$ e-ld my_object.o -o exec.elf
```

The following example shows how to use the linker to create an elf executable for the Epiphany multicore evaluation kit.

```
$ e-ld -T $EPIPHANY_HOME/bsps/zedboard/fast.ldf \
    my_object.o -o exec.elf
```

     REV 5.13.07.10

5.3 Command Line Options

Table 5.1: Linker Command Line Options

Option	Function
-L*searchdir* --library-path=*searchdir*	Adds directory paths to the list of paths that 'e-ld' will search for archive libraries and linker control scripts. The option can be used multiple times, in which case directories are searched in the order in which they are specified on the command line. If *searchdir* begins with =, then the = will be replaced by the sysroot prefix, a path specified when the linker is configured.
-e *entry* --entry=*entry*	The *entry* option can be used to specify the explicit symbol for beginning execution of your program, rather than the default entry point. If there is no symbol named entry, the linker will try to parse *entry* as a number, and use that as the entry address. (Numbers without a prefix will be interpreted in base 10; numbers starting with 0x will be interpreted as base 16.)
-M --print-map	Prints a link map to the standard output, including information about: Object file memory placement, Common symbol, allocation, Symbol value assignments
-o *output* --output=*output*	Specifies name of the output file of the link process. Without this option, the default output name is 'a.out'.
-s --strip-all	Omit all symbol information from the output file.
-S --strip-debug	Omit debugger symbol information (but not all symbols) from the output file.
-T *scriptfile* --script=*scriptfile*	Replaces the default linker script file with *scriptfile*. If script file does not exist in the current directory, the linker looks for it in the directories specified by any `-L' options. Multiple `-T' options accumulate.
-u *symbol* --undefined=*symbol*	Forces *symbol* to be entered in the output file as an undefined symbol.
@*file*	Read command-line options from *file*. Options in *file* are separated by whitespace. The *file* itself may contain additional file may @file options, allowing for modular linker configuration.

 REV 5.13.07.10

5.4   Linker Script Overview

The link process is controlled by a linker script written in the GNU linker command language. The purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. If you do not supply a linker file to 'e-ld', it will use a default linker file.

Executables compiled with the default linker will only execute correctly using the 'e-run' instruction set simulator and will not work correctly when loaded on specific hardware targets. To correctly link for specific hardware targets, you should use the '-T' option to specify one of the board specific linker files that come with the board support package (BSP) or your own custom linker file.

The following list highlights some of the key concepts of the linker:

- The assembler emits an object file (partial program) with an assumed start at address 0. The linker then reads one or more object files and combines their contents to form a runnable program with no program overlap and all addresses completely resolved.

- The linker script contains a number of defined input sections and output sections. These section names can be used within the source code to assist in fine grained code and data placement as explained in the following sections.

- Every object file has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address.

- When compiling a C program into an object file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

5.5   Explicit Code and Data Memory Management

The linker description files that come with the different Epiphany Board Support Packages have a number of key words that allow fine grained management of code and data placement from within the C/C++ source code. The keywords gives the programmer and support libraries complete control of the placement of data and code within the memory system on a per-symbol, per-file, and per -object library. The keywords are derived from section names within the linker descriptor file and can be augmented by the user at his discretion. Table 5.2 gives a summary of the memory placement keywords available in all Epiphany BSPs that can be used within your C code.

Table 5.2: Memory Management Linker Symbols

Keyword	Meaning
__core_row__	The row ID of the core
__core_col__	The column ID of the core
__stack_start__	Stack starting point
__heap_start__	Heap starting point
__heap_end__	Heap ending point

Table 5.3 shows configurations of the three basic linker descriptor files. The 'legacy' scenario is to be used for bringing up code quickly but will run slowly since all data and code is placed in external memory. The 'fast' scenario is used to place user code internally and standard library `code externally. The 'internal' scenario can be used for to effectively place all code and data in the local memory by default. The three descriptor files effectively determine the default placement of all sections and symbols within the objects. The user can override these settings on an individual basis from the C/C++ source code using the attributes defined in Table 5.2 to specify that certain variables and/or functions should be placed in specific program output sections. Note that with all of the predefined LDF's, the heap is allocated externally. This means that use of stdio library will render the program very slow.

Table 5.3: Memory Management Scenarios

File	USER CODE & DATA	STANDARD LIBRARY	STACK	NOTE
legacy.ldf	External SDRAM	External SDRAM	External SDRAM	Use to run any legacy code with up to 1MB of combined code and data.
fast.ldf	Internal SRAM	External SDRAM	Internal SRAM	Places all user code and static data in local memory, including the stack. Use to implement fast critical functions. It is the user's responsibility to ensure that the code fits within the local memory.
internal.ldf	Internal SRAM	Internal SRAM	Internal SRAM	Places all code and static data in local memory, including the stack. Use to implement fastest applications. It is the user's responsibility to ensure that the code fits within the local memory.

Table 5.4: Linker Sections

Section	User Controllable Sections
.text	Application code, read only
.data	Application data (global variables that are not constant)
.rodata	Application data, read only (constants, strings)
.bss	Static variables initialized to zero
.text_bank0	Starts at end of reserved section in bank0
.text_bank1	Starts at the beginning of bank1
.text_bank2	Starts at the beginning of bank2
.text_bank3	Starts at the beginning of bank3
.data_bank0	End of .text_bank0
.data_bank1	End of .text_bank1
.data_bank2	End of .text_bank2
.data_bank3	End of .text_bank3
.code_dram	Code section in external memory
.shared_dram	Data section in external memory
.heap_dram	Heap section in external memory

          REV 5.13.07.10

5.6 Memory Management Examples

The Epiphany SDK gives the programmer complete control over data and code placement through section attributes that can be embedded in the source code. Memory management attributes placed in the source code will be ignored by the standard Linux GCC compiler.

The following examples illustrate some attributes that can be placed inside the source code or in a stub '*.c' file that gets compiled with the rest of the application source file. The general attribute should be placed outside the main routine.

1. How to specify the core where the executable will run:

```
asm(".global __core_row__;");
asm(".set __core_row__,0x20;");
asm(".global __core_col__;");
asm(".set __core_col__,0x24;");
```

2. How to specify where the stack should start:

```
asm(".global __stack_start__;");
asm(".set __stack_start__,0x1ff0;");
```

3. How to force memory placement of a static variable:

```
float data[N] __attribute__ ((section (".data_bank3")));
```

4. How to force memory placement of a function in declaration:

```
int my_fft(int *ptr) __attribute__ ((section (".text_bank0")))
```

5. How to force a long jump attribute on a function situated in external SDRAM:

```
int dump_memory(int *ptr) __attribute__ ((long call));
```

      REV 5.13.07.10

6. ELF Utilities

6.1 Overview

The Epiphany SDK includes several utilities that can be used to effectively manipulate binary object files. This chapter provides a brief overview of these utilities.

6.2 Utility Summary

Table 6.1 explains the major binary manipulation utilities within the Epiphany SDK and their respective arguments. All utilities use the --help switch to print out a complete set of arguments.

Table 6.1 ELF Manipulation Programs

Utility	Note	Usage + Arguments
e-ar	Creates and manipulates archive content	-r Replace existing or insert new files into archive
e-nm	Lists the symbols in an object file	
e-objcopy	Copies a binary file, possibly transforming it in the process	*e-objcopy [options] in-file [out-file]* -S Remove all symbol and relocation information -g Remove all debugging information --srec-forceS3 Generate srec type output --gap-fill <val> Fill gaps between sections with <val> --set-start <addr> Set the start address to <addr> -W <name> Force <name> symbol to be marked weak --strip-unneeded Remove all unneeded symbols --prefix-sections <prefix> Add prefix to section names --prefix-symbols <prefix> Add prefix to symbol names
e-objdump	Displays information about the content of object files	*e-objdump [options] file* -x Display the contents of all headers -d Display content of all executable sections -D Display content of all sections -t Display content of of the symbol tabe -T Display content of dynamic symbol table -r Display relocation entries in file ---section=NAME Only display section NAME
e-size	Lists the section sizes within an object file	*e-size [options] file* -o|-d|-x Display numbers in octal, decimal, or hex
e-strip	Strips symbols from object files	*e-strip [options] file* --remove-section=<name> Remove section <name> -g Remove all debug symbols and sections -s Remove all symbols and relocation information -o <file> Place stripped output into <file>

     REV 5.13.07.10

7. Instruction Set Simulator (E-RUN)

7.1   Overview

The Epiphany Instruction Set Simulator (ISS) is an accurate and fast functional representation of the Epiphany Instruction Set Architecture. The simulator accurately models the instruction set and register map of a single Epiphany core, but does not model pipeline behavior or any of the non-CPU hardware mechanisms such as the eMesh Network-On-Chip, DMA, or timers. The simulator runs in a host Linux environment, takes a binary ELF file as an input and supports standard I/O. To simplify program debugging and profiling, the simulator supports outputting program traces.

7.2   Simple Example

The following example shows how to simulate the execution of an Epiphany elf executable using the ISS within a Linux host platform.

```
$ e-run hello_world.elf
```

The simulator will print out "Hello World!"

To get an instruction trace of the executed program, use the '-t' option before the hello_world.elf argument as follows:

```
$ e-run -t hello_world.elf
```

7.3   Command Line Options

Note that the elf file should be the last argument given at the command line.

Table 7.1: Simulator Command Line Options

Option	Function
-t, --trace	Output simulated instruction trace to scree
--memory-region ADDRESS,SIZE	Defines a memory region as valid for simulator. Default is to allow 0x0→1MB
--help	Prints help

     REV 5.13.07.10

8. Hardware Connection Server (E-SERVER)

8.1 Overview

The GDB client runs on the Linux host machine and communicates with the Epiphany GDB server using GDB's internal RSP (remote serial protocol) over TCP/IP ports. The e-server responds to the GDB client or Loader requests and controls the hardware or hardware emulation model. Each core in the system needs a separate GDB client and connects to the GDB server using a unique TCP/IP port. By default cores connect to the e-server starting at port 51000. The e-server responds to the e-loader requests in the dedicated port.

An illustration of the GDB server/client operations is shown below:

Figure 8.1: The eServer Client-Target Connection Concept
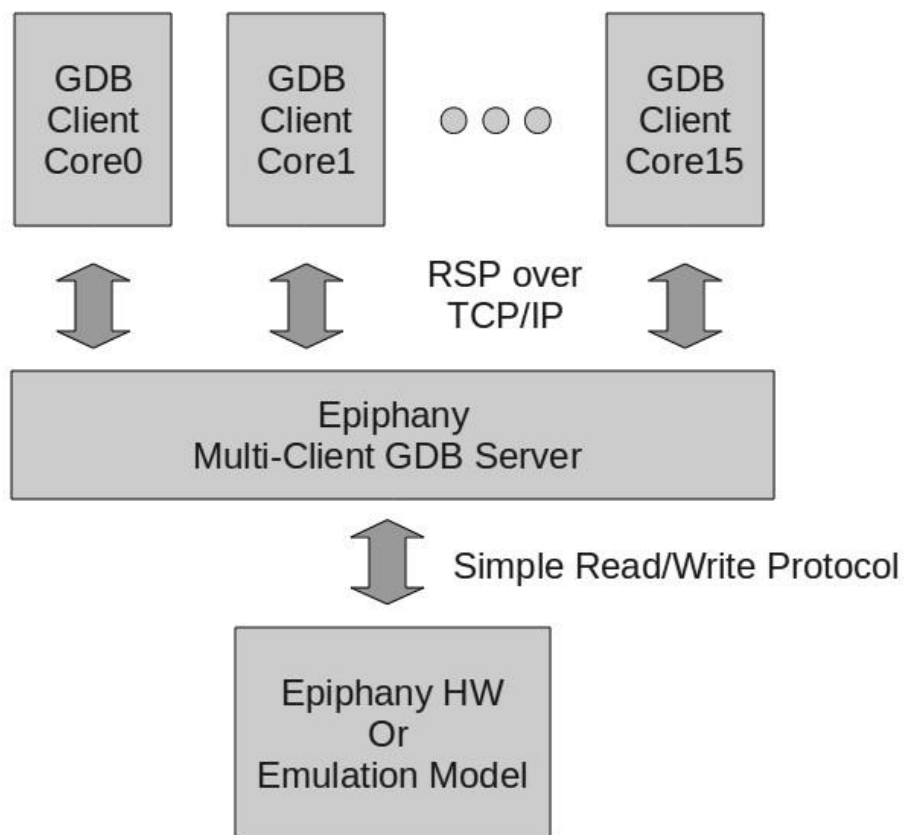
8.2 Simple Example

To start the Target Server open a terminal window and type the following in the command line:

```
$ e-server –hdf ${EPIPHANY_HOME}/bsps/zedboard/zed_E16G3_512mb.xml
```

The debug server now responds with status messages regarding the connection process and the results of the memory test, if performed. The output should be similar to the following:

```
Using the HDF file: zed_E16G3_512mb.xml
Listening for RSP on port 51000
Listening for RSP on port 51001
Listening for RSP on port 51002
Listening for RSP on port 51004
Listening for RSP on port 51005
:
Listening for RSP on port 51006
Listening for RSP on port 51007
Listening for RSP on port 51010
Listening for RSP on port 51009
Listening for RSP on port 51011
```

8.3 Command Line Options

Table 8.1: eServer Command Line Options

Argument	Note
-hdf *file*	Mandatory argument, specifies the platform specific description file containing platform definitions, normally located in the esdk/bsps/{platform} directory
-p <port_base>	Assign base socket port number for the server. The default is 51000
--show-memory-map	Print out the supported memory map
--test-memory	Test the memory before serving the debugger clients [*only for select platforms]
-Wpl <options>	Pass comma-separated <options> on to the platform driver
-Xpl <arg>	Pass <arg> on to the platform driver
--version	Display the version number and copyright info

8.4 Target Server Connection API

The run-time connection from a host to the epiphany target is performed via the eHAL library. For documentation on this library, see chapter 14 of this book, "*Epiphany Host Library (eHAL)*".

 REV 5.13.07.10

9. Debugger (E-GDB)

9.1 Overview

The Epiphany debugger (e-gdb) is based on the popular GNU GDB debugger. It allows the programmer to see what is going on inside a program while it executes. Some of the powerful debug features enabled by the debugger include:

- Interactive program load

- Stopping program on specific conditions (usually a breakpoint placed in source code)

- Examine complete state of machine and program once program has stopped.

- Continuing program one instruction at a time or until the next stop condition is met.

The Epiphany debugger supports program debugging using the functional simulator as a target or the hardware platform as a target using the 'e-server'. The only difference between the two modes of debugging is the argument specified with the 'target' command within the debugger client. The simulator only supports debugging programs running on a single Epiphany CPU core and is not multi-core aware.

Note: In order to enable program debugging, the compiler should be invoked with the -g option.

 REV 5.13.07.10

9.2 Simple Examples

The following example shows how to debug a simple "Hello World" program with the Epiphany instruction set simulator.

In a Linux shell, start an e-gdb session using your executable.

```
$ e-gdb hello_world.elf
```

Inside e-gdb, connect to the instruction set simulator debugging target.

```
(gdb) target sim
```

Load the program to the core memory.

```
(gdb) load
```

Place a breakpoint at the main function entry point.

```
(gdb) b main
```

Run the functional simulator.

```
(gdb) run
```

Continue program execution from breakpoint.

```
(gdb) c
```

Program then runs until completion and displays.

```
"Hello World!"
```

Exit debugger

```
(gdb) q
```

     REV 5.13.07.10

The following example shows how to debug a program running on an Epiphany based hardware target.

Make sure that a connection has been established with the hardware using the e-server program:

```
$ e-server –hdf ${EPIPHANY_HOME}/bsps/emek3/emek3.xml -test-memory
```

In a Linux shell, start a e-gdb session using your executable (same as for the simulator).

```
$ e-gdb hello_world.elf
```

Inside e-gdb, connect to the TCP/IP socket connected to core that you want to debug.

```
(gdb) target remote:51000
```

Load the program the core memory.

```
(gdb) load
```

Place a breakpoint at the main function entry point.

```
(gdb) b main
```

Continue program execution from breakpoint.

```
(gdb) c
```

Program then runs until completion and displays.

```
"Hello World!"
```

Exit debugger

```
(gdb) q
```

             REV 5.13.07.10

9.3   Command Line Options

Invoke the debugger by running the program 'e-gdb'. Once started, 'e-gdb' reads commands from the terminal until you tell it to exit. You can also run 'e-gdb' with a variety of arguments and options, to specify more of your debugging environment at the outset.

The most common way to start 'e-gdb' is to simply specify the program as the only argument:

```
$ e-gdb program.elf
```

Table 9.1: Debugger Command Line Options

Option	Function
-x *file*	Execute gdb commands from file *file*.
-d *directory*	Add *directory* to the path to search for source files.
-quiet | q -silent	"Quiet". Do not print the introductory and copyright messages.
-batch	Run in batch mode. Exit with status 0 after processing all the command files specified with '-x' (and all commands from initialization files, if not inhibited with '-n'). Exit with nonzero status if an error occurs in executing the gdb commands in the command files.
-nowindows -nw	"No windows".
-windows -w	If gdb includes a GUI, then this option requires it to be used if possible.
-async	Use the asynchronous event loop for the command-line interface. gdb processes all events, such as user keyboard input, via a special event loop.
-version	This option causes gdb to print its version number and exit

9.4   Quitting GDB

To quit 'e-gdb', enter 'q' or 'quit' at the 'e-gdb' command line. An interrupt (often *Ctrl-C*) does not exit from 'e-gdb', but rather terminates the action of any 'e-gdb' command that is in progress and returns to the 'e-gdb' command level.

     REV 5.13.07.10

9.5   Shell I/O

If you need to execute occasional shell commands during your debugging session, there is no need to leave or suspend e-gdb. You can just use the shell command:

```
shell command string
```

You may want to save the output of e-gdb commands to a file. There are several commands
To control e-gdb's logging, use:

```
set logging [on/off]
```

To control name of current logfile:

```
set logging file
```

To overwrite existing logfile instead of appending:

```
set logging overwrite [on|off]
```

To specify that log should only go to file:

```
set logging redirect [on|off]
```


9.6   Getting Help

You can use `help` (abbreviated `h`) with no arguments to display a short list of named classes of Commands.

9.7   Command Syntax

An e-gdb command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. Any text from "#" to the end of the line is a comment, which can be useful in command files.

     REV 5.13.07.10

9.8　Command Summary

Table 9.2: Debugger Commands

Command	Function
info	Give information about specific keyword
breakpoint	Sets a breakpoint.
break *function*	Sets a breakpoint at entry to function
break -offset break +offset	Sets a breakpoint some lines forward or back from position
break linenum	Set a breakpoint at line *linenum* in the current source file. The current source file is the last file whose source text was printed.
break *filename*:*linenum*	Set a breakpoint at line *linenum* in source file *filename*.
break filename:function	Set a breakpoint at line *linenum* in source file *filename*.
break **address*	Set a breakpoint at address *address*.
break	When called without any arguments, break sets a breakpoint at the next instruction to be executed in the selected stack frame
info breakpoints [*n*] info break [*n*] info watchpoints [*n*]	Print a table of all breakpoints, watchpoints, and catchpoints set and not deleted, with the following columns for each breakpoint:
watch expr	Set a watchpoint for an expression.
clear	Delete any breakpoints at the next instruction to be executed in the selected stack frame
clear *function* clear *filename*:*function*	Delete any breakpoints set at entry to the function *function*.
clear *linenum* clear *filename*:*linenum*	Delete any breakpoints set at or within the code of the specified line.
delete [breakpoints] [*range*...]	Delete the breakpoints, watchpoints, or catchpoints of the breakpoint ranges specified as arguments. If no argument is specified, delete all breakpoints
disable [breakpoints] [*range*...]	Disable the specified breakpoints—or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten.
enable [breakpoints] [*range*...]	Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.
continue [*ignore-count*] c [*ignore-count*] fg [*ignore-count*]	Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of ignore
step	Continue running your program until control reaches a different source line, then stop it and return control to gdb. *Warning:* If you use the step command while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, It will not step into a function which is compiled without debugging information. To step through functions without debugging information, use the stepi command, described below
step *count*	Continue running as in step, but do so *count* times.

 　　REV 5.13.07.10

next [*count*]	Continue to the next source line in the current (innermost) stack frame. This is similar to step, but function calls that appear within the line of code are executed without stopping.
set step-mode on	The set step-mode on command causes the step command to stop at the first instruction of a function which contains no debug line information rather than stepping over it.
set step-mode off	Causes the step command to step over any functions which contains no debug information. This is the default.
finish	Continue running until just after function in the selected stack frame returns.
until	Continue running until a source line past the current line, in the current stack frame, is reached. This command is used to avoid single stepping through a loop more than once.
stepi stepi *arg* si	Execute one machine instruction, then stop and return to the debugger.
nexti nexti *arg* ni	Execute one machine instruction, but if it is a function call, proceed until the function returns.
handle *signal keywords*	The keywords allowed by the handle command can be abbreviated. Their full names are: nostop gdb should not stop your program when this signal happens. It may still print a message telling you that the signal has come in. stop gdb should stop your program when this signal happens. This implies the print keyword as well. print gdb should print a message when this signal happens. noprint gdb should not mention the occurrence of the signal at all. This implies the nostop keyword as well. Pass noignore gdb should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled. pass and noignore are synonyms. Nopass ignore gdb should not allow your program to see this signal. nopass and ignore are synonyms.
b backtrace	Print a backtrace of the entire stack: one line per frame for all frames in the stack.
backtrace n bt n	Similar, but print only the innermost n frames.
frame n f n	Select frame number n. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is the one for main.
list linenum	Print lines centered around line number linenum in the current source file.
set listsize count	Make the list command display count source lines
list first,last	Print lines from first to last.
disassemble disassemble /m disassemble /r	This specialized command dumps a range of memory as machine instructions. It can also print mixed source + disassembly by specifying the /m modifier and print the raw instructions in hex as

	well as in symbolic form by specifying the /r. The default memory range is the function surrounding the program counter of the selected frame.
print expr print /f expr	expr is an expression (in the source language). By default the value of expr is printed in a format appropriate to its data type;
x/nfu addr x addr x	Use the x command to examine memory. n, f, and u are all optional parameters that specify how much memory to display and how to format it; addr is an expression giving the address where you want to start displaying memory. If you use defaults for nfu, you need not type the slash '/'. For example, 'x/3uh 0x54320' is a request to display three halfwords (h) of memory, formatted as unsigned decimal integers ('u'), starting at address 0x54320. All the defaults for the arguments to x are designed to make it easy to continue scanning memory with minimal specifications each time you use x. For example, after you have inspected three machine instructions with 'x/3i addr', you can inspect the next seven with just 'x/7'.
compare-sections [section-name]	Compare the data of a loadable section section-name in the executable file of the program being debugged with the same section in the remote machine's memory, and report any mismatches.
display expr	Add the expression expr to the list of expressions to display each time your program stops.
$_	The variable $_ is automatically set by the x command to the last address examined
info all-registers	Print the names and values of all registers including floating-point and vector registers (in the selected stack frame).
info registers regname ...	Print the relativized value of each specified register regname. There are four standard registers in GDB: pc,sp,fp,ps(status). Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict.
info mem	Print a table of all defined memory regions, with the following columns for each region:
find [/sn] start_addr, +len, val1 [, val2, ...] find [/sn] start_addr, end_addr, val1 [, val2, ...]	Search memory for the sequence of bytes specified by val1, val2, etc. The search begins at address start addr and continues for either len bytes or through to end addr inclusive.
set var symb=lvalue	Setting a symbol value from GDB
jump linespec jump location	Resume execution at line linespec or at address given by location. Execution stops again immediately if there is a breakpoint there.
target remote host:port target remote tcp:host:port	Debug using a TCP connection to port on host. The host may be either a host name or a numeric ip address; port must be a decimal number. The host could be the target machine itself, if it is directly connected to the net, or it might be a terminal server which in turn has a serial line to the target.
target sim	Builtin Epiphany CPU simulator.

 REV 5.13.07.10

set debug monitor	Enable or disable display of communications messages between the debugger and the remote monitor.
detach	When you have finished debugging the remote program, you can use the detach command to release it from debugger control.
disconnect	The disconnect command behaves like detach, except that the target is generally not resumed.

9.9   Epiphany GDB Limitations

The Epiphany implementation of GDB currently lacks support for:

1. Tracing

2. Hardware assisted watchpoints

REV 5.13.07.10

10.   Epiphany SDK Utilities (E-UTILS)

10.1 Overview

The Epiphany SDK is provided with a group of command-line utility programs. These programs are used to perform Epiphany system related tasks during program development and debugging. The e-utils programs include:

```
e-reset
e-loader
e-read
e-write
e-hw-rev
```

10.2 Reset Utility (E-RESET)

The Epiphany reset utility (e-reset) is used to reset the Epiphany subsystem, in case it gets stuck due to some unstable situation, or in order to bring it to a known state.

10.2.1   Example

At the command prompt, type:

```
$ e-reset
```

10.3 Loader Utility (E-LOADER)

The Epiphany loader (e-loader) is responsible for loading programs onto the hardware platform. The input to the loader is a compiled and linked Epiphany program, that was generated by e-gcc/e-ld. Currently, the loader supports binary images formatted as a text file with a standard S-record (known as SREC) file format. This format is an ASCII hexadecimal ("hex") text encoding for binary data. The S-record is an output of the binary utility 'e-objcopy'.

     REV 5.13.07.10

When loading a binary image on the chip there is a need to translate the internal core addresses to global space addresses. During compile time, the build tools do not know what core will be the target of the executable. This information is known only at load time. Thus, the insertion of the core ID data has to be done prior to sending the SREC file to the e-loader. When loading images of more than one core, each partial SREC has to be pre-processed separately.

10.3.1   Command Line Options

```
e-loader [-s|--start] [-r|--reset] <e-program> [row col [rows cols]]
```

Table 10.1: Loader Command Line Options

Option	Function
-r, --reset	Perform a full hardware reset of the Epiphany platform before loading the program.
-s, --start	With this option set, the loaded programs are started after finished loading on all cores in workgroup.
<e-program>	Path to the program image to load on the core workgroup.
row, col	Absolute coordinates of first core in workgroup to be loaded. The default values are the platform's first physical core.
rows, cols	Size of cores workgroup to be loaded. The default values are (1,1)
-h, --help	Display a help message.

10.3.2   Example

After building an Epiphany elf program, translate from elf to S-record program load format:

```
$ e-objcopy --srec-forceS3 --output-target srec main1.elf main1.srec
```

Load program onto the target, on a 4×4 block of cores starting at core 0x808 (32,8) and start it immediately after:

```
$ e-loader --start main.srec 32 8 4 4
```

      REV 5.13.07.10

Or, perform a system reset and load the program onto a single core at the chip's origin. Then wait for host command to start the program.

```
$ e-loader --reset main.srec
```

10.4 Memory Read Utility (E-READ)

The Epiphany memory read utility (e-read) is used to read words from memory locations on the Epiphany chip(s) or the External Memory.

10.4.1   Command Line Options

```
e-read <row> [<col>] <address> [<num-words>]
```

Table 10.2: e-read Command Line Options

Option	Function
<row>	Row coordinate of the target core. To read data from External Memory, enter -1. The core coordinates are relative to the platform's chip bounding box. That is, all of the Epiphany chips are considered one workgroup, where the first core of the first chip is at coordinates (0,0).
[<col>]	Row coordinate of the target core. When reading from External Memory, this parameter is omitted.
<address>	The start address of the read data. Address is given as local space when reading from a core memory, or as an offset from the platform's External Memory base. Address should be in hexadecimal format and is rounded down to the word (4-bytes) alignment.
[<num-words>]	Number of words to read from the target, starting at <address>. If this parameter is omitted, a single word is read.

10.4.2   Example

The following command will read the STATUS register of the 3rd core in the 1st row (0,2):

```
$ e-read 0 2 0xf0404
```

The following command will read 16 words from the External Memory, starting at offset 0x4000:

```
$ e-read -1 0x4000 16
```

10.5 Memory Write Utility (E-WRITE)

The Epiphany memory write utility (e-write) is used to write words to memory locations on the Epiphany chip(s) or the External Memory.

10.5.1   Command Line Options

```
e-write <row> [<col>] <address> [<val0> <val1> ...]
```

Table 10.3: e-write Command Line Options

Option	Function
<row>	Row coordinate of the target core. To read data from External Memory, enter -1. The core coordinates are relative to the platform's chip bounding box. That is, all of the Epiphany chips are considered one workgroup, where the first core of the first chip is at coordinates (0,0).
[<col>]	Row coordinate of the target core. When reading from External Memory, this parameter is omitted.
<address>	The start address of the read data. Address is given as local space when reading from a core memory, or as an offset from the platform's External Memory base. Address should be in hexadecimal format and is rounded down to the word (4-bytes) alignment.
[<val0> <val1> …]	Number of words to write to the target, starting at <address>. If this parameter is omitted, the input is taken in interactive mode from the standard input, one word at a time, until an empty line is entered. Values are entered as 32-bit hexadecimal numbers.

     REV 5.13.07.10

10.5.2   Example

The following command will program the CONFIG register of the 3rd core in the 1st row (0,2) such that the core works in the "truncate" rounding mode:

```
$ e-write 0 2 0xf0400 0x1
```

The following command will write the first eight elements of the Fibonacci Series to a buffer in the External Memory, starting at offset 0x120:

```
$ e-write -1 0x120 1 1 2 3 5 8 d 15
```

10.6 Hardware Revision Utility (E-HW-REV)

The hardware revision utility (e-hw-rev) returns the current revision number of the Epiphany subsystem programmed logic.

10.6.1   Example

At the command prompt, type:

```
$ e-hw-rev
```

                REV 5.13.07.10

11.   Standard Library Support

11.1 Overview

The Epiphany SDK includes a set of libraries based on the newlib distribution of Standard C and Standard Math libraries. These libraries are bundled with the e-gcc compiler. Usage of these libraries does not require the use of the –L switch for specifying library location search path.

11.2 Standard C Libraries

Table 11.1 shows the key components within the Standard C library, with complete support for file I/O. For a complete explanation of the C functions, please refer to the newlib manual at: http://sourceware.org/newlib/libc.html

Table 11.1: Key Standard C Library Components

Library	Function
<stdlib.h>	Standard utility functions
<ctype.h>	Character classification macros
<stdio.h>	Input/output functions
<strings.h>	String handling functions
<signals.h>	Event handling

The Standard C Library functions are reentrant whenever possible.

REV 5.13.07.10

11.3 Standard Math Library (math.h)

The Standard Math library is based on the newlib math library (libm). The complete newlib math library reference manual can be located at: http://sourceware.org/newlib/libm.html

Alternative declarations of the mathematical functions, which exploit specific machine capabilities to operate faster—but generally have less error checking and may reflect additional limitations on some machines—are available when you include fastmath.h instead of math.h.

When a libm function detects an exceptional case, `errno` may be set, the `matherr()` function may be called, and an error message may be written to the standard error stream. This behavior may not be reentrant.

      REV 5.13.07.10

12. Epiphany System Programming Model

When programming an application for the Epiphany system, a few parallel programming models are applicable. The Epiphany Run-Time library and Host library are designed to support programming in the relocateable core workgroup model.

The Epiphany platform is comprised of the Epiphany chip arrangement and External Memory. Individual eCores are referenced in a context of Workgroups. External Memory regions are addressed relative to the External Memory region base. The physical allocation of the Epiphany chips and External Memory are defined in the provided Hardware Description File (HDF). This way, an application can be easily migrated between platforms where the location of physical chips and External Memory differ.

A Workgroup is a collection of adjacent eCores on the Epiphany chip(s), normally intended for the parallel execution of a computational task. The workgroup is rectangular and its size and origin are defined at run-time by the host. Its parameters are set by a host library function and are maintained in a workgroup object.

The collection of chips in the Epiphany system form a rectangular area, which is the minimal bounding box including all the physical chips in the platform. A workgroup is defined in terms of its row and column coordinates relative to this bounding box, and its size in terms of number of rows and columns of cores. Thus, the first core of the first chip is defined as located at coordinates (0, 0). For example, consider a system comprised of 4 Epiphany-16 (E16) chips arranged in two rows containing two chips each. A workgroup of size 2 by 2 cores, originating at the 4th row and 6th column will have the following parameters: (row, col, rows, cols) = (3, 5, 2, 2).

An eCore member of the workgroup is referenced in terms of its row and column coordinates relative to the workgroup's origin. Thus, the first core in the workgroup is at coordinates (0, 0).

An example of coordinates and references designation can be seen in Figure 12.1.

 REV 5.13.07.10

Figure 12.1: Platform, Workgroup and eCore coordinates

7c7	7c8	7c9	7ca	7cb	7cc
807	808 - CoreID (32,8) - System Origin (4,8) - System Size	809	80a	80b	80c
847	848	849	84a	84b	84c
887	888	889 (2,1) - Workgroup Origin (2,3) - Workgroup Size	88a	88b (0,2) - eCore Coordinates in Workgroup	88c
8c7	8c8	8c9	8ca	8cb	8cc
907	908	909	90a	90b	90c

Epiphany Space

Epiphany E16 Chip

eCore

Workgroup
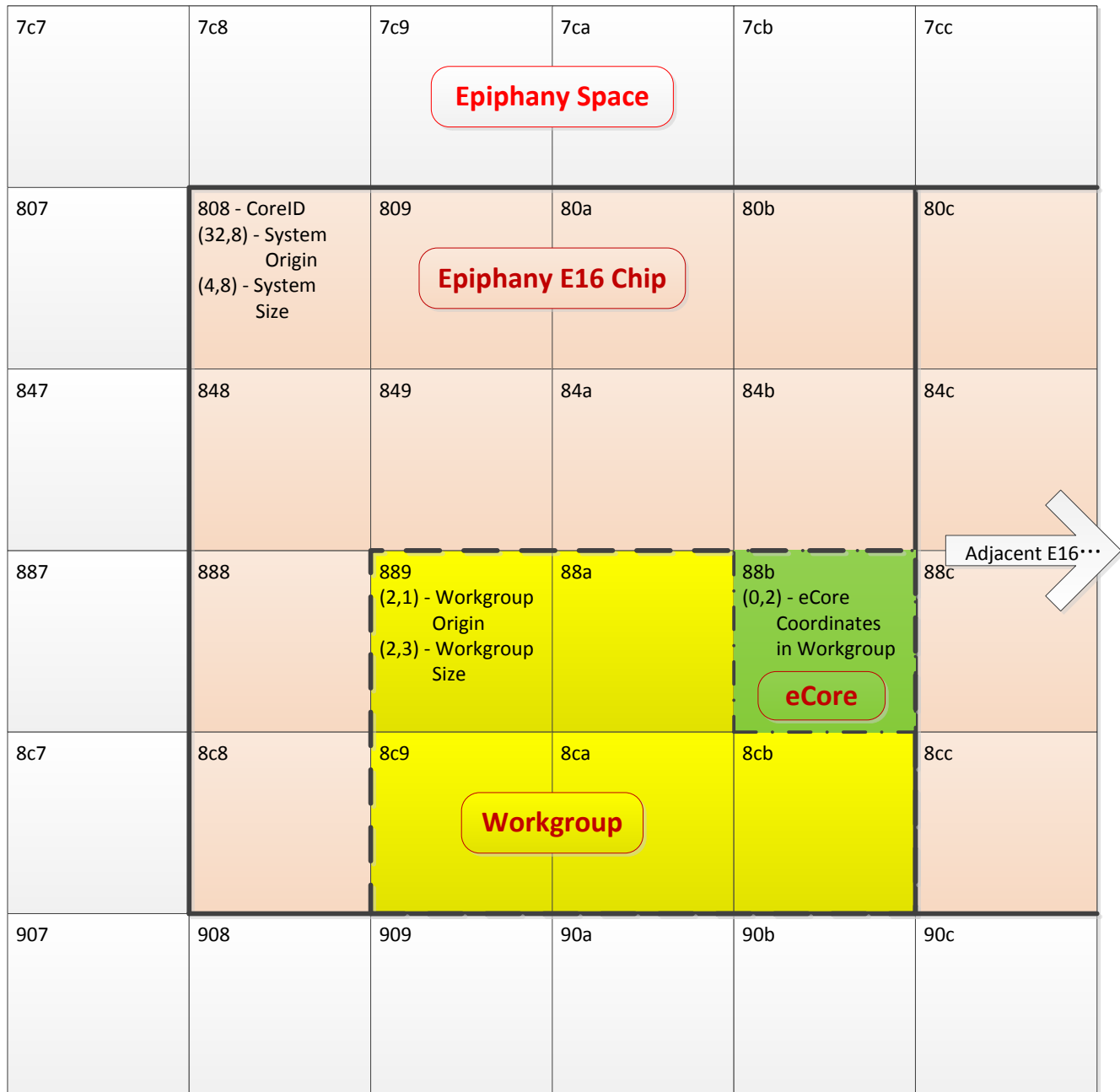
Adjacent E16···

Here the platform is comprised of two adjacent E16 chips, one is located at coordinates (32, 8) and the other at (32, 12). The Platform's total size is 4 by 8 cores. A Workgroup of size 2 by 3 is defined starting at core (34, 9). Hence, its relative position is (2, 1). The 3rd eCore on the workgroup's 1st row has CoreID 0x88b, which is at absolute coordinates (34, 11). Hence, its relative coordinates are (0, 2).

 REV 5.13.07.10

13.   Epiphany Hardware Utility Library (eLib)

13.1 Overview

The Epiphany Hardware Utility library provides functions for configuring and querying the Epiphany hardware resources. These routines automate many common programming tasks that are not provided by the C and C++ languages and are specific to the Epiphany architecture.

In the following sections, the various eLib functions are described. Each section provides details on a family of APIs. The master header file for the eLib, which includes all the per-family headers, is the "`e-lib.h`" header file. Include this header file at the beginning of a program that uses the eLib functions and objects.

```
#include "e-lib.h"
```

In order to use this library to build an Epiphany program, use the e-gcc compiler option `-le-lib` on the build command line.

Each core on the platform is referenced via a definition of a workgroup. Two global objects are available at each core's space. One object, called `e_group_config`, contains the information about the chip type, the workgroup's position and size, and the core's position in the containing workgroup. Its members are:

```
e_group_config.chiptype   - Type of chip containing the core
e_group_config.group_id   - CoreID of first core in Workgroup
e_group_config.group_row  - Origin of Workgroup
e_group_config.group_col
e_group_config.group_rows - Size of Workgroup
e_group_config.group_cols
e_group_config.core_row   - Coordinates of core
e_group_config.core_col
```

The other object, called `e_emem_config`, contains information about the External Memory base address. Its member is:

```
e_emem_config.base        - Absolute address of base of ext. mem.
```

     REV 5.13.07.10

In addition to the function prototypes and specific type enumerations, the following definitions are provided: Two macro shortcuts for the "align", "packed" and "section" function and variable attributes are defined as:

```
#define ALIGN(x)    __attribute__ ((aligned (x)))
#define PACKED      __attribute__ ((packed))
#define SECTION(x)  __attribute__ ((section (x)))
```

The e_bool_t type is defined as follows:

```
typedef enum {
     E_FALSE,
     E_TRUE,
} e_bool_t;
```

The e_return_stat_t type defined the eLib functions return values:

```
typedef enum {
     E_OK,
     E_ERR,
     E_WARN,
} e_return_stat_t;
```

       REV 5.13.07.10

13.2 System Register Access Functions

13.2.1   Overview

The system register access functions enable the reading from and writing to the hardware special registers.

Functions definition summary

```
unsigned e_reg_read(e_reg_id_t reg_id);
void e_reg_write(e_reg_id_t reg_id, unsigned val);
```

Enumerated constants and macros

```
// General Purpose Registers
typedef enum
{
    E_REG_R0,      E_REG_R8,      E_REG_R16,      E_REG_R24,
    E_REG_R1,      E_REG_R9,      E_REG_R17,      E_REG_R25,
    E_REG_R2,      E_REG_R10,     E_REG_R18,      E_REG_R26,
    E_REG_R3,      E_REG_R11,     E_REG_R19,      E_REG_R27,
    E_REG_R4,      E_REG_R12,     E_REG_R20,      E_REG_R28,
    E_REG_R5,      E_REG_R13,     E_REG_R21,      E_REG_R29,
    E_REG_R6,      E_REG_R14,     E_REG_R22,      E_REG_R30,
    E_REG_R7,      E_REG_R15,     E_REG_R23,      E_REG_R31,

    E_REG_R32,     E_REG_R40,     E_REG_R48,      E_REG_R56,
    E_REG_R33,     E_REG_R41,     E_REG_R49,      E_REG_R57,
    E_REG_R34,     E_REG_R42,     E_REG_R50,      E_REG_R58,
    E_REG_R35,     E_REG_R43,     E_REG_R51,      E_REG_R59,
    E_REG_R36,     E_REG_R44,     E_REG_R52,      E_REG_R60,
    E_REG_R37,     E_REG_R45,     E_REG_R53,      E_REG_R61,
    E_REG_R38,     E_REG_R46,     E_REG_R54,      E_REG_R62,
    E_REG_R39,     E_REG_R47,     E_REG_R55,      E_REG_R63,
} e_gp_reg_id_t;


// eCore Special Registers
typedef enum
{
    // Control Registers
    E_REG_CONFIG,              E_REG_IRET,
    E_REG_STATUS,              E_REG_IMASK,
    E_REG_FSTATUS,             E_REG_ILAT,
    E_REG_PC,                  E_REG_ILATST,
    E_REG_DEBUGSTATUS,         E_REG_ILATCL,
    E_REG_DEBUGCMD,            E_REG_IPEND,
```

                          REV 5.13.07.10

```c
    E_REG_LC,
    E_REG_LS,
    E_REG_LE,

    // DMA registers
    E_REG_DMA0CONFIG,           E_REG_DMA1CONFIG,
    E_REG_DMA0STRIDE,           E_REG_DMA1STRIDE,
    E_REG_DMA0COUNT,            E_REG_DMA1COUNT,
    E_REG_DMA0SRCADDR,          E_REG_DMA1SRCADDR,
    E_REG_DMA0DSTADDR,          E_REG_DMA1DSTADDR,
    E_REG_DMA0AUTODMA0,         E_REG_DMA1AUTODMA0,
    E_REG_DMA0AUTODMA1,         E_REG_DMA1AUTODMA1,
    E_REG_DMA0STATUS,           E_REG_DMA1STATUS,

    // Event Timer Registers
    E_REG_CTIMER0,              E_REG_CTIMER1,

    // Processor Control Registers
    E_REG_MEMPROTECT,
    E_REG_MESH_CONFIG,
    E_REG_COREID,
    E_REG_CORE_RESET,
} e_core_reg_id_t;


// Chip Registers
typedef enum
{
    E_REG_IO_LINK_MODE_CFG,
    E_REG_IO_LINK_TX_CFG,
    E_REG_IO_LINK_RX_CFG,
    E_REG_IO_LINK_DEBUG,
    E_REG_IO_GPIO_CFG,
    E_REG_IO_FLAG_CFG,
    E_REG_IO_SYNC_CFG,
    E_REG_IO_HALT_CFG,
    E_REG_IO_RESET,
} e_chip_reg_id_t;
```

     REV 5.13.07.10

13.2.2   *e_reg_read()*

Synopsis

```
#include "e-lib.h"
unsigned e_reg_read(e_core_reg_id_t reg_id);
```

Description

Reads value from one of the system registers within the caller core.

Return value

Return the current value read from one of the system registers as identified by `reg_id`.

                       REV 5.13.07.10

13.2.3   e_reg_write()

Synopsis

```
#include "e-lib.h"
void e_reg_write(e_core_reg_id_t reg_id, unsigned val);
```

Description

Set the value of the system register identified by `reg_id` within the caller core, to `val`.

Return value

None.

               REV 5.13.07.10

13.3 Interrupt Service Functions

13.3.1   Overview

The Interrupt Service functions handle system interrupt control and generation. It is possible to generate interrupts in the local core or in a remote core.

Functions definition summary

```
void e_irq_attach(e_irq_type_t irq, sighandler_t handler);
void e_irq_global_mask(e_bool_t state);
void e_irq_mask(e_irq_type_t irq, e_bool_t state);
void e_irq_set(unsigned row, unsigned col, e_irq_type_t irq);
void e_irq_clear(unsigned row, unsigned col, e_irq_type_t irq);
```

Enumerated constants, macros and types

```
typedef void (*sighandler_t)(int);


typedef enum
{
    E_SYNC,
    E_SW_EXCEPTION,
    E_MEM_FAULT,
    E_TIMER0_INT,
    E_TIMER1_INT,
    E_MESSAGE_INT,
    E_DMA0_INT,
    E_DMA1_INT,
    E_USER_INT,
} e_irq_type_t
```

      REV 5.13.07.10

13.3.2   e_irq_attach()

Synopsis

```
#include "e-lib.h"
void e_irq_attach(e_irq_type_t irq, sighandler_t handler);
```

Description

This function attaches (registers) an interrupt handler function (ISR), given by `handler`, to a specific entry in the IVT (Interrupt Vector Table), specified by `irq`.

Using this function, ISR for a specific event type can be assigned and replaced in run-time. It uses an indirect handler attachment, which may impose a slight delay on the execution of the handler in case of an event.

The ISR should be compiled using the `interrupt` function attribute in order to apply proper entry and exit sequences, guaranteeing safe context switching.

Return value

None.

               REV 5.13.07.10

13.3.3   e_irq_global_mask()

Synopsis

```
#include "e-lib.h"
void e_irq_global_mask(e_bool_t state);
```

Description

Globally enable or disable interrupts on caller core. When `state` is `E_TRUE`, the `GID` bit of the core's `STATUS` register is set and consequent interrupt events are masked. When `state` is `E_FALSE`, the `GID` bit is cleared and consequent interrupt events are tested against the other masking mechanisms and pending interrupts.

Return value

None.

                             REV 5.13.07.10

13.3.4   *e_irq_mask()*

Synopsis

```
#include "e-lib.h"
void e_irq_mask(e_irq_type_t irq, e_bool_t state);
```

Description

Disable or enables a single interrupt event type, specified by irq, by setting its respective bit in the core's IMASK register according to state. If state is E_TRUE, then consequent interrupt events of type irq are masked. If state is E_FALSE, this interrupt type is not masked.

Return value

None.

    REV 5.13.07.10

13.3.5   e_irq_set()

Synopsis

```
#include "e-lib.h"
void e_irq_set(unsigned row, unsigned col, e_irq_type_t irq);
```

Description

Generate an interrupt event by setting its ILAT register bit specified by `irq`. The event is generated on the core with relative coordinates (`row`, `col`) in a core workgroup.

Return value

None.

        REV 5.13.07.10

13.3.6   e_irq_clear()

Synopsis

```
#include "e-lib.h"
void e_irq_clear(unsigned row, unsigned col, e_irq_type_t irq);
```

Description

Clears pending interrupt request by clearing its ILAT register bit specified by `irq`. The request is cleared from the core with relative coordinates (`row`, `col`) in a core workgroup.

Return value

None.

      REV 5.13.07.10

13.4 Timer Functions

13.4.1 Overview

The Timer functions interface the system timers (two per core) and the read, write and manipulation of their operation.

Functions definition summary

```
unsigned e_ctimer_get(e_ctimer_id_t timerid);

unsigned e_ctimer_set(e_ctimer_id_t timerid, unsigned val);

unsigned e_ctimer_start(e_ctimer_id_t timerid,
    e_ctimer_config_t config);

unsigned e_ctimer_stop(e_ctimer_id_t timerid);

void e_wait(e_ctimer_id_t timerid, unsigned clicks);
```

Enumerated constants and macros

```
typedef enum
{
    E_CTIMER_0,
    E_CTIMER_1,
} e_ctimer_id_t;

typedef enum
{
    E_CTIMER_OFF,
    E_CTIMER_CLK,
    E_CTIMER_IDLE,
    E_CTIMER_IALU_INST,
    E_CTIMER_FPU_INST,
    E_CTIMER_DUAL_INST,
    E_CTIMER_E1_STALLS,
    E_CTIMER_RA_STALLS,
    E_CTIMER_EXT_FETCH_STALLS,
    E_CTIMER_EXT_LOAD_STALLS,
} e_ctimer_config_t;

#define E_CTIMER_MAX
```

 REV 5.13.07.10

13.4.2   *e_ctimer_get()*

Synopsis

```
#include "e-lib.h"
unsigned e_ctimer_get(e_ctimer_id_t timerid);
```

Description

Read value of the core's timer specified by `timerid`. Note that the core counters decrement on events and stop counting at zero.

Return value

Returns the current value of timer `timerid`.

     REV 5.13.07.10

13.4.3   *e_ctimer_set()*

Synopsis

```
#include "e-lib.h"
unsigned e_ctimer_set(e_ctimer_id_t timerid, unsigned val);
```

Description

Sets value of the core's timer specified by `timerid` to `val`. Note that the core counters decrement on events and stop counting at zero. Use `E_CTIMER_MAX` to set `val` to the maximum allowed value.

and the initial value of the ctimer count register to `val`. Note that the core counters decrement on events and stop counting at zero.

Return value

Returns the new value of timer `timerid`.

                           REV 5.13.07.10

13.4.4   e_ctimer_start()

Synopsis

```
#include "e-lib.h"
unsigned e_ctimer_start(e_ctimer_id_t timerid,
    e_ctimer_config_t config);
```

Description

Causes the ctimer specified by `timerid` to begin counting down upon events. The type of events to be counted is specified by `config`. The function sets the ctimer configuration field `CTIMERxCFG` in the core's `CONFIG` register to `config`.

Return value

Returns the current value of timer `timerid`.

        REV 5.13.07.10

13.4.5   *e_ctimer_stop()*

Synopsis

```
#include "e-lib.h"
unsigned e_ctimer_stop(unsigned timerid);
```

Description

Causes the ctimer specified with `timerid` to stop counting down by setting the ctimer configuration field `CTIMERxCFG` in the core's `CONFIG` register to `E_CTIMER_OFF`.

Return value

Returns the current value of the stopped timer.

          REV 5.13.07.10

REV 5.13.07.10

13.4.6   e_wait()

Synopsis

```
#include "e-lib.h"
void e_wait(e_ctimer_id_t timerid, unsigned clicks);
```

Description

Pauses the execution of the program for the number of clock cycles specified by `clicks`.

This function utilizes ctimer `timerid` for counting the clocks. Consequently, it will override whatever counting process is currently being performed by ctimer `timerid`. Make sure to store the old value before calling `e_wait()` if required later.

Note that as this function counts clock cycles, the actual time (wall-clock) depends on the clock rate of the Epiphany chip.

Return value

None.

                               REV 5.13.07.10

13.5 DMA and Data Movement Functions

13.5.1   Overview

The DMA functions control the two DMA channels included in each core. Functionality is provided for querying status, configuring and copying memory using the DMA engine.

Functions definition summary

```
void *e_read(void *remote, void *dst, unsigned row, unsigned col,
    const void *src, size_t bytes);

void *e_write(void *remote, const void *src, unsigned row,
    unsigned col, void *dst, size_t bytes);

int e_dma_copy(void *dst, void *src, size_t bytes);

int e_dma_start(e_dma_desc_t *descriptor, e_dma_id_t chan);

int e_dma_busy(e_dma_id_t chan);

void e_dma_wait(e_dma_id_t chan);

void e_dma_set_desc(e_dma_id_t chan,
    unsigned config, e_dma_desc_t *next_desc,
    unsigned stride_i_src, unsigned stride_i_dst,
    unsigned count_i, unsigned count_o,
    unsigned stride_o_src, unsigned stride_o_dst,
    void *addr_src, void *addr_dst, e_dma_desc_t *descriptor);
```

Enumerated constants, macros and types

```
typedef enum
{
    E_DMA_0,
    E_DMA_1
} e_dma_id_t;


typedef struct
{
    unsigned config;
    unsigned inner_stride;
    unsigned count;
    unsigned outer_stride;
    void    *src_addr;
    void    *dst_addr;
} e_dma_desc_t;


typedef enum
{
    E_DMA_ENABLE,
```

       REV 5.13.07.10

```
        E_DMA_MASTER,
        E_DMA_CHAIN,
        E_DMA_STARTUP,
        E_DMA_IRQEN,
        E_DMA_BYTE,
        E_DMA_HWORD,
        E_DMA_WORD,
        E_DMA_DWORD,
        E_DMA_MSGMODE,
        E_DMA_SHIFT_SRC_IN,
        E_DMA_SHIFT_DST_IN,
        E_DMA_SHIFT_SRC_OUT,
        E_DMA_SHIFT_DST_OUT,
} e_dma_config_t;
```

      REV 5.13.07.10

13.5.2   e_read()

Synopsis

```
#include "e-lib.h"
void *e_read(void *remote, void *dst, unsigned row, unsigned col,
    const void *src, size_t bytes);
```

Description

Copy `bytes` bytes of data from a remote source `src` to a local destination `dst`. The remote source can be either a core on the caller core's workgroup, or an External Memory buffer. The `remote` parameter must be either `e_group_config` or `e_emem_config`, specifying the nature if the source.

If the `remote` parameter is `e_group_config`, then the source core is specified by its (`row`, `col`) coordinates in the caller core's workgroup. If the `src` address is a global address, then it is used unmodified.

If the `remote` parameter is `e_emem_config`, then the source address is given relative to the External Memory base address. In this case, the `row` and `col` parameters are ignored.

Return value

None.

          REV 5.13.07.10

13.5.3   e_write()

Synopsis

```
#include "e-lib.h"
void *e_write(void *remote, const void *src, unsigned row,
    unsigned col, void *dst, size_t bytes);
```

Description

Copy `bytes` bytes of data from a local source `src` to a remote destination `dst`. The remote destination can be either a core on the caller core's workgroup, or an External Memory buffer. The `remote` parameter must be either `e_group_config` or `e_emem_config`, specifying the nature if the destination.

If the `remote` parameter is `e_group_config`, then the destination core is specified by its (`row`, `col`) coordinates in the caller core's workgroup. If the `dst` address is a global address, then it is used unmodified.

If the `remote` parameter is `e_emem_config`, then the destination address is given relative to the External Memory base address. In this case, the `row` and `col` parameters are ignored.

Return value

None.

          REV 5.13.07.10

13.5.4   e_dma_copy()

Synopsis

```
#include "e-lib.h"
int e_dma_copy(void *dst, void *src, size_t bytes);
```

Description

Copy `bytes` bytes of data from `src` to `dst` using the DMA engine `DMA1`. If the DMA channel is busy when calling this function, it waits until the previous transfer is concluded. After initiating the DMA transfer process it waits until the transfer is finished (blocking DMA).

This is generally a faster alternative to the standard `memcpy()` function. However, utilizing the DMA, it has some limitations that the standard function does not impose, like some restrictions on the source and destination addresses. Please consult the Epiphany Architecture Reference Manual for more details.

Return value

Returns `0` if successful.

                       REV 5.13.07.10

13.5.5   e_dma_start()

Synopsis

```
#include "e-lib.h"
int e_dma_start(e_dma_desc_t *descriptor, e_dma_id_t chan);
```

Description

Start a DMA on channel `chan` as described by the `descriptor` descriptor. Use the `e_dma_desc_t` constants to populate the `descriptor` descriptor fields.

Return value

Returns 0 if successful.

     REV 5.13.07.10

13.5.6 e_dma_busy()

Synopsis

```
#include "e-lib.h"
int e_dma_busy(e_dma_id_t chan);
```

Description

Queries the status of the state machine of dma channel `chan`.

Return value

Return `0` if the DMA channel identified by `chan` is idle, otherwise return DMA channel status.

 REV 5.13.07.10

13.5.7   *e_dma_wait()*

Synopsis

```
#include "e-lib.h"
void e_dma_wait(e_dma_id_t chan);
```

Description

Halts the execution of the program and waits as long as DMA channel `chan` is busy.

Return value

None.

     REV 5.13.07.10

13.5.8   *e_dma_set _desc()*

Synopsis

```
#include "e-lib.h"

void e_ dma_set_desc(e_dma_id_t chan,
    unsigned config, e_dma_desc_t *next_desc,
    unsigned stride_i_src, unsigned stride_i_dst,
    unsigned count_i, unsigned count_o,
    unsigned stride_o_src, unsigned stride_o_dst,
    void *addr_src, void *addr_dst, e_dma_desc_t *descriptor);
```

Description

Sets the DMA descriptor `descriptor` of DMA channel `chan` with the various members:

config   - 16-bit configuration field. This field is comprised of the Logical OR of the
                  `e_dma_config_t` constants.

next_desc   - 16-bit address of DMA descriptor, loaded when current transfer ends, when
                  `E_DMA_CHAIN` mode was set.

stride_i_src   - 16-bit stride of the inner loop of the source address generator.

stride_i_dst   - 16-bit stride of the inner loop of the destination address generator.

count_i   - 16-bit count of the inner loop transactions. This value must be positive.

count_o   - 16-bit count of the outer loop transactions. This value must be positive.

stride_o_src   - 16-bit stride of the outer loop of the source address generator.

stride_o_dst   - 16-bit stride of the outer loop of the destination address generator.

addr_src   - 32-bit start address of source data.

addr_dst   - 32-bit start address of destination data.

Use this function to make sure that the DMA channel is idle when programming the descriptor.

Return value

None.

          REV 5.13.07.10

13.6 Mutex and Barrier Functions

13.6.1   Overview

A mutex is an object which allows locking of a shared resource, enabling exclusive access to just one agent. When an access to the shared resource is required, first the associated mutex is checked. If the mutex is cleared, then resource is free. The mutex is then set and access is granted to the querying agent.

A barrier is a means for synchronizing parallel executing threads. When a program reaches a barrier, it will wait until all other threads reached the barrier as well. Only then will the program (and all the other programs) continue their execution.

Functions definition summary

```
void e_mutex_init(unsigned row, unsigned col, e_mutex_t *mutex,
    e_mutexattr_t *attr);
void e_mutex_lock(unsigned row, unsigned col, e_mutex_t *mutex);
unsigned e_mutex_trylock(unsigned row, unsigned col,
    e_mutex_t *mutex);
void e_mutex_unlock(unsigned row, unsigned col, e_mutex_t *mutex);
void e_barrier_init(volatile e_barrier_t bar_array[],
    e_barrier_t *tgt_bar_array[]);
void e_barrier(volatile e_barrier_t *bar_array,
    e_barrier_t *tgt_bar_array[]);
```

Enumerated constants, macros and types

```
typedef int  e_mutex_t;
typedef int  e_mutexattr_t;
typedef char e_barrier_t;
```

                     REV 5.13.07.10

13.6.2   e_mutex_init()

Synopsis

```
#include "e-lib.h"
void e_mutex_init(unsigned row, unsigned col, e_mutex_t *mutex,
    e_mutexattr_t *attr);
```

Description

This function initializes the mutex referenced by `mutex`, on the core at coordinates (`row`, `col`) in the caller core's workgroup. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

The initialization attribute, specified by `attr`, is reserved for future use. When calling the function, use `NULL` for `attr`.

Return value

Returns `0` upon successful initialization; otherwise, a nonzero error value is returned.

        REV 5.13.07.10

13.6.3   *e_mutex_lock()*

Synopsis

```
#include "e-lib.h"
void e_mutex_lock(unsigned row, unsigned col, e_mutex_t *mutex);
```

Description

This function tries to lock the mutex object referenced by `mutex`, on the core at coordinates (`row`, `col`) in the caller core's workgroup. If the mutex is already locked, the calling thread will be blocked until the mutex becomes available.

Return value

If successful, the function returns `0`; otherwise, a nonzero error value is returned.

                   REV 5.13.07.10

13.6.4   e_mutex_trylock()

Synopsis

```
#include "e-lib.h"
unsigned e_mutex_trylock(unsigned row, unsigned col,
    e_mutex_t *mutex);
```

Description

This function tries to lock the mutex object referenced by `mutex`, on the core at coordinates (`row`, `col`) in the caller core's workgroup. If the mutex is already locked, the function returns with a failure code.

Return value

If successful, the function returns `0`; otherwise, nonzero value is returned, which is the Core ID of the agent that holds the associated resource.

13.6.5   *e_mutex_unlock()*

Synopsis

```
#include "e-lib.h"
void e_mutex_unlock(unsigned row, unsigned col, e_mutex_t *mutex);
```

Description

This function unlocks the mutex object referenced by `mutex`, on the core at coordinates (`row`, `col`) in the caller core's workgroup.

Return value

If successful, the function returns `0`; otherwise, a nonzero error value is returned.

          REV 5.13.07.10

13.6.6   e_barrier_init()

Synopsis

```
#include "e-lib.h"
void e_barrier_init(volatile e_barrier_t bar_array[],
    e_barrier_t *tgt_bar_array[]);
```

Description

Initialize a workgroup barrier. The `bar_array` and `tgt_bar_array` parameters are defined as arrays of size equal to the number of cores in the workgroup. The barrier is mutual to all cores in the workgroup, so care must be taken w

Return value

None.

        REV 5.13.07.10

13.6.7   *e_barrier()*

Synopsis

```
#include "e-lib.h"
void e_barrier(volatile e_barrier_t *bar_array,
    e_barrier_t *tgt_bar_array[]);
```

Description

Set a workgroup barrier point (a sync point) across the workgroup cores. When the program reaches the barrier point, it will halt and wait until all cores in the workgroup reached that point as well.

The `bar_array` and `tgt_bar_array` parameters are defined as arrays of size equal to the number of cores in the workgroup, and must be initialized by `e_barrier_init()`.

The barrier is mutual to all cores in the workgroup, so care must be taken when placing the `e_barrier()` call, to prevent deadlock conditions.

Return value

None.

         REV 5.13.07.10

13.7 Core ID and Workgroup Functions

13.7.1   Overview

The Core ID is a number which identifies a core in the system. Each core is associated with a unique number that is related to the core's coordinates in the global mesh. The ID is a 12-bit number where the 6 high order bits are the core row coordinate and the 6 low order bits are the core column coordinates. This number also indicates the core's 1MB slice in the global memory space, where it comprises the most significant bits of the core's globally addressable space.

Functions definition summary

```
e_coreid_t e_get_coreid(void);

void *e_get_global_address(unsigned row, unsigned col,
    const void *ptr);

e_coreid_t e_coreid_from_coords(unsigned row, unsigned col);

void e_coords_from_coreid(e_coreid_t coreid, unsigned *row,
    unsigned *col);

e_bool_t e_is_on_core(const void *ptr);

void e_neighbor_id(e_coreid_wrap_t dir, e_coreid_wrap_t wrap,
    unsigned *row, unsigned *col);
```

Enumerated constants and macros

```
typedef unsigned int e_coreid_t;


#define E_SELF


typedef enum
{
    // neighboring cores wrap topology
    E_CHIP_WRAP, // all chip cores form a ring
    E_ROW_WRAP,  // core rows form rings
    E_COL_WRAP,  // core columns form rings
    // neighboring cores direction
    E_NEXT_CORE, // neighbor core with the next coreID
    E_PREV_CORE, // neighbor core with the prev coreID
} e_coreid_wrap_t
```

      REV 5.13.07.10

```
typedef enum {
    E_E16G301,
    E_E64G401,
} e_chiptype_t;

typedef struct {
    e_chiptype_t chiptype;
    e_coreid_t   group_id;
    unsigned     group_row;
    unsigned     group_col;
    unsigned     group_rows;
    unsigned     group_cols;
    unsigned     core_row;
    unsigned     core_col;
    unsigned     alignment_padding;
} e_group_config_t;

typedef struct {
    unsigned base;
} e_emem_config_t;
```

          REV 5.13.07.10

13.7.2   e_get_coreid()

Synopsis

```
#include "e-lib.h"
e_coreid_t e_get_coreid(void);
```

Description

Reads coreid from the hardware register.

Return value

Returns a 12-bit coreid value, aligned to lsb of the result.

13.7.3   e_get_global_address ()

Synopsis

```
#include "e-lib.h"
void *e_get_global_address(unsigned row, unsigned col,
     const void *ptr);
```

Description

Transforms a local pointer `ptr` into the matching address on a neighbor core, referred to by coordinates (`row`, `col`), in the caller core's workgroup.

Note that for `ptr` values that point to a global address not local to `coreid`, the function returns an unmodified version of `ptr`.

If either `row` or `col` are `E_SELF`, or they are equal to the caller core's own coordinates, then the function calculates the global version of the local address. That is, the returned address is the same address as would be referenced from outside of the core.

If `ptr` points to a global address (that is, its 12-bit msb's are nonzero), then it is returned unmodified.

Return value

Returns a 32-bit absolute global address.

13.7.4   e_coreid_from_coords()

Synopsis

```
#include "e-lib.h"
e_coreid_t e_coreid_from_coords(unsigned row, unsigned col);
```

Description

Returns the `coreid` value of the neighbor core, referred to by coordinates (`row`, `col`), in the caller core's workgroup.

Return value

Returns 12-bit wide `coreid` value.

      REV 5.13.07.10

13.7.5   e_coords_from_coreid()

Synopsis

```
#include "e-lib.h"
void e_coords_from_coreid(e_coreid_t coreid, unsigned *row,
    unsigned *col);
```

Description

Calculate the row and column coordinates (`row`, `col`) of the core specified by `coreid`, in the caller's workgroup.

Note that no check is made for a `coreid` value outside of the workgroup. In such case, the return coordinates may be either bigger than the workgroup's size or negative.

Return value

None.

          REV 5.13.07.10

13.7.6   *e_is_oncore()*

Synopsis

```
#include "e-lib.h"
e_bool_t e_is_on_core(const void *ptr);
```

Description

This function checks whether an address (either global or local) is within the memory space of the caller core.

Return value

The function returns E_FALSE if the address is not in the caller's space. Otherwise, it returns E_TRUE.

    REV 5.13.07.10

13.7.7   e_neighbor_id()

Synopsis

```
#include "e-lib.h"
void e_neighbor_id(e_coreid_wrap_t dir, e_coreid_wrap_t wrap,
    unsigned *row, unsigned *col);
```

Description

This function calculates the (`row`, `col`) coordinates of the neighboring core, according to a specified topology.

Cores can be logically chained in one linear string across the whole chip, from north-west core to south-east core in a raster scan fashion. The cores can also be chained in a row-wise fashion or column-wise fashion, such that rows or columns create parallel rings.

The `dir` argument (one of `E_NEXT_CORE, E_PREV_CORE`) specifies whether the next or previous cores in the chain are required. The function will always calculate the ID of another core on the same chip, wrapping on a row, column, or chip boundary as specified by the `wrap` argument (one of `E_ROW_WRAP, E_COL_WRAP, E_CHIP_WRAP`). This calculated ID is returned in the `coreid` parameter.

This function is limited to workgroup dimensions (rows and columns) which are powers of 2, i.e., 2, 4, 8, etc.

Return value

None.

14.   Epiphany Host Library (eHAL)

14.1 Overview

The Epiphany Hardware Abstraction Layer (eHAL) library provides functionality for communicating with the Epiphany chip when the application runs on a host. The host can be a PC or an embedded processor. The communication is performed using memory writes to and reads from shared buffers that the applications on both sides should define. The library interface is defined in the `e-hal.h` header file.

In order to use this library in your application, the compiler and linker must be configured with the paths to the header file and the library binary. In your tools options use the following configurations:

```
$ gcc -I${EPIPHANY_HOME}/tools/host/include \
      -L${EPIPHANY_HOME}/tools/host/lib -le-hal ...
```

Basic mode of operation

As described in an earlier chapter, the standard mode of operation of the eHAL API is working in eCore workgroups. A workgroup is a rectangular mesh of eCore nodes that are allocated for performing a computational task. It is possible to load the group with identical copies of the same program (SPMD style), or load subgroups, or even single cores with different programs. It is the user's responsibility to make sure that tasks are not allocated to a previously allocated group cores.

External (shared) memory architecture

The host application can communicate with the Epiphany device by either accessing the eCore's private memory space, or by using shared buffers in the device external memory.

In a platform which implements such shared memory (for example, a bulk of DRAM accessible by the host via system bus or other connection, and by the Epiphany via the eLinks), there may be a different mapping of the physical address space of this memory, as seen from the host side and from the Epiphany side. For example, the Parallella platform is configured by default with 32MB of DRAM used as device memory. The physical address of this memory segment is

     REV 5.13.07.10

`0x1e000000÷0x1ffffffff`. However, to overcome some system limitations, this range is aliased to address `0x8e000000÷0x8ffffffff`, as seen from the Epiphany side. For example, when a buffer of 8KB is allocated at offset 64KB on that segment, the host sees this buffer as occupying addresses `0x1e010000÷0x1e012000`. For accessing the buffer from the Epiphany program, this range is aliased to `0x8e010000÷0x8e012000`.

The base addresses of the external shared memory space (the real and the aliased) are defined in the Hardware Description File (HDF) so the eHAL is aware of the difference. The aliased base address is also defined in the Epiphany program's Linker Description File (LDF).

Enumerated constants and macros

```
typedef enum {
    E_FALSE,
    E_TRUE,
} e_bool_t;


typedef enum {
    E_OK,
    E_ERR,
    E_WARN,
} e_return_stat_t;
```

The following symbols are defined and can be used as addresses to access eCore and Epiphany system registers using the `e_read()` and `e_write()` API's:

```
// General Purpose Registers
// (see Epiphany Architecture Manual for details)
typedef enum
{
    E_REG_R0,     E_REG_R8,     E_REG_R16,     E_REG_R24,
    E_REG_R1,     E_REG_R9,     E_REG_R17,     E_REG_R25,
    E_REG_R2,     E_REG_R10,    E_REG_R18,     E_REG_R26,
    E_REG_R3,     E_REG_R11,    E_REG_R19,     E_REG_R27,
    E_REG_R4,     E_REG_R12,    E_REG_R20,     E_REG_R28,
    E_REG_R5,     E_REG_R13,    E_REG_R21,     E_REG_R29,
    E_REG_R6,     E_REG_R14,    E_REG_R22,     E_REG_R30,
    E_REG_R7,     E_REG_R15,    E_REG_R23,     E_REG_R31,

    E_REG_R32,    E_REG_R40,    E_REG_R48,     E_REG_R56,
    E_REG_R33,    E_REG_R41,    E_REG_R49,     E_REG_R57,
    E_REG_R34,    E_REG_R42,    E_REG_R50,     E_REG_R58,
    E_REG_R35,    E_REG_R43,    E_REG_R51,     E_REG_R59,
```

 REV 5.13.07.10

```
    E_REG_R36,    E_REG_R44,    E_REG_R52,    E_REG_R60,
    E_REG_R37,    E_REG_R45,    E_REG_R53,    E_REG_R61,
    E_REG_R38,    E_REG_R46,    E_REG_R54,    E_REG_R62,
    E_REG_R39,    E_REG_R47,    E_REG_R55,    E_REG_R63,
} e_gp_reg_id_t;


// eCore Special Registers
typedef enum
{
    // Control Registers
    E_REG_CONFIG,              E_REG_IRET,
    E_REG_STATUS,              E_REG_IMASK,
    E_REG_FSTATUS,             E_REG_ILAT,
    E_REG_PC,                  E_REG_ILATST,
    E_REG_DEBUGSTATUS,         E_REG_ILATCL,
    E_REG_DEBUGCMD,            E_REG_IPEND,
    E_REG_LC,
    E_REG_LS,
    E_REG_LE,


    // DMA registers
    E_REG_DMA0CONFIG,          E_REG_DMA1CONFIG,
    E_REG_DMA0STRIDE,          E_REG_DMA1STRIDE,
    E_REG_DMA0COUNT,           E_REG_DMA1COUNT,
    E_REG_DMA0SRCADDR,         E_REG_DMA1SRCADDR,
    E_REG_DMA0DSTADDR,         E_REG_DMA1DSTADDR,
    E_REG_DMA0AUTODMA0,        E_REG_DMA1AUTODMA0,
    E_REG_DMA0AUTODMA1,        E_REG_DMA1AUTODMA1,
    E_REG_DMA0STATUS,          E_REG_DMA1STATUS,


    // Event Timer Registers
    E_REG_CTIMER0,             E_REG_CTIMER1,


    // Processor Control Registers
    E_REG_MEMPROTECT,
    E_REG_MESH_CONFIG,
    E_REG_COREID,
    E_REG_CORE_RESET,
} e_core_reg_id_t;


// Chip Registers
// (see Epiphany Chip Datasheets for details)
typedef enum
{
    E_REG_IO_LINK_MODE_CFG,
    E_REG_IO_LINK_TX_CFG,
    E_REG_IO_LINK_RX_CFG,
    E_REG_IO_LINK_DEBUG,
    E_REG_IO_GPIO_CFG,
    E_REG_IO_FLAG_CFG,
    E_REG_IO_SYNC_CFG,
    E_REG_IO_HALT_CFG,
```

```
    E_REG_IO_RESET,
} e_chip_reg_id_t;


// Epiphany system registers
// (see Board manual for details)
typedef enum
{
    E_SYS_CONFIG,
    E_SYS_RESET,
    E_SYS_VERSION,
    E_SYS_FILTERL,
    E_SYS_FILTERH,
    E_SYS_FILTERC,
} e_sys_reg_id_t
```

          REV 5.13.07.10

14.2 Platform Configuration Functions

14.2.1   Overview

These functions are used to initialize and prepare the Epiphany system for working with the Host application. It also enables the query and retrieval of platform information.

Functions definition summary

```
int e_init(char *hdf);
int e_get_platform_info(e_platform_t *platform);
int e_finalize();
```

     REV 5.13.07.10

14.2.2   e_init()

Synopsis

```
#include "e-hal.h"
int e_init(char *hdf);
```

Description

This function initializes the HAL data structures, and establishes a connection to the Epiphany platform. The platform parameters are read form a Hardware Description File (HDF), whose path is given at the function argument.

If the `hdf` parameter is a `NULL` pointer, then the file location is read from the `EPIPHANY_HDF` environment variable. This variable is normally set on your system startup file (`~/.bashrc` in Linux), and reflects the structure of the underlying Epiphany platform. For example:

```
EPIPHANY_HDF="${EPIPHANY_HOME}/bsps/parallella/parallella.xml"
```

If the `EPIPHANY_HDF` variable is not set, then the function will try to locate the `platform.hdf` file located in the current BSP directory.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

Note: At the time of the release, the XML parser was not yet fully integrated into the driver. Instead of an XML description file, the library now uses a simplified (flat) text file listing the platform components. Please use the provided files or create your own accordingly:

```
EPIPHANY_HDF="${EPIPHANY_HOME}/bsps/parallella/parallella.hdf"
```

14.2.3   e_get_platform_info()

Synopsis

```
#include "e-hal.h"
int e_get_platform_info(e_platform_t *platform);
```

Description

The Epiphany platform information is stored internally in an `e_platform_t` type object. It contains the data on the various chips, external memory segments and geometry comprising the system. Some of this data can be retrieved through this function.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

Note: The data that is currently made available through this function is:

```
char *version        - platform version string
unsigned row, col    - coordinates of effective chip area
unsigned rows, cols  - size of effective chip area
int num_chips        - number of Epiphany chips in platform
int num_emems        - number of external memory segments
```

If necessary for the application, the internal object can be accessed using this declaration:
```
extern e_platform_t e_platform;
```
However, this practice should be normally avoided, and if used, absolutely no modification of the data is allowed, or the integrity of the driver system may be broken. Additionally, because the variable is currently exposed (the `extern` keyword is not really necessary), there should be no user-defined object of this name in the application!

      REV 5.13.07.10

14.2.4   *e_finalize()*

Synopsis

```
#include "e-hal.h"
int e_finalize();
```

Description

Use this function to finalize the connection with the Epiphany system. Some resources that were allocated in the `e_init()` call are released here.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

     REV 5.13.07.10

14.3 Workgroup and External Memory Allocation Functions

14.3.1   Overview

These functions are used to assign and allocate the eCore workgroups and external memory buffers resources.

Functions definition summary

```
int e_open(e_epiphany_t *dev, unsigned row, unsigned col,
    unsigned rows, unsigned cols);
int e_close(e_epiphany_t *dev);
int e_alloc(e_mem_t *mbuf, off_t base, size_t size);
int e_free(e_mem_t *mbuf);
```

     REV 5.13.07.10

14.3.2   e_open()

Synopsis

```
#include "e-hal.h"
int e_open(e_epiphany_t *dev, unsigned row, unsigned col,
    unsigned rows, unsigned cols);
```

Description

This function defines an eCore workgroup. The workgroup is defined in terms of the coordintaes relative to the platform's effective chip area. The arguments `row` and `col` define the place of the group's origin eCore. The origin is set relative to the Epiphany platform's origin, defined in the `e_init()` call. The arguments `rows` and `cols` give the group's size, defining the work rectangle. A work group can be as amall as a single core or as large as the whole available effective chip area. The core group data is saved in the provided `e_epiphany_t` type object `dev`.

Subsequent accesses to the core group (e.g., for read and write of data) are done using a reference to the `dev` object.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

       REV 5.13.07.10

14.3.3   e_close()

Synopsis

```
#include "e-hal.h"
int e_close(e_epiphany_t *dev);
```

Description

The function closes the eCore workgroup. The resources allocated by the `e_open()` call are released here. Use this function before re-allocating an eCore to a new workgroup.

Return value

If successful, the function returns E_OK. On a failure it returns E_ERR.

14.3.4   *e_alloc()*

Synopsis

```
#include "e-hal.h"
int e_alloc(e_mem_t *mbuf, off_t base, size_t size);
```

Description

This function defines a buffer in external memory. The buffer is defined in terms of the relative from the beginning of the external memory segment, defined in the `e_init()` call. The argument `base` defines the offset, starting at 0. The argument and `size` gives the buffer's size. The external memory buffer data is saved in the provided `e_mem_t` type object `mbuf`.

Subsequent accesses to the buffer (e.g., for read and write of data) are done using a reference to the `mbuf` object.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

          REV 5.13.07.10

14.3.5   *e_free()*

Synopsis

```
#include "e-hal.h"
int e_free(e_mem_t *mbuf);
```

Description

The resources allocated by the e_alloc() call are released here. Use this function before re-allocating an external memory space to a new buffer.

Return value

If successful, the function returns E_OK. On a failure it returns E_ERR.

14.4 Data Transfer Functions

14.4.1   Overview

These functions are used to read and write data from and to Epiphany eCore workgroups and external memory buffers.

Functions definition summary

```
ssize_t e_read(void *dev, unsigned row, unsigned col,
    off_t from_addr, void *buf, size_t size);
ssize_t e_write(void *dev, unsigned row, unsigned col,
    off_t to_addr, const void *buf, size_t size);
```

     REV 5.13.07.10

14.4.2   e_read()

Synopsis

```
#include "e-hal.h"
ssize_t e_read(void *dev, unsigned row, unsigned col,
    off_t from_addr, void *buf, size_t size);
```

Description

This function reads data of length `size` from a workgroup core or an external memory buffer to the local byte buffer `buf`. The argument `dev` specifies the target from which to read the data. It can be of either types `e_epiphany_t` or `e_mem_t`.

If an object of type `e_epiphany_t` is given, then the `row` and `col` arguments specify the relative target eCore coordinates in the workgroup.

If an object of type `e_mem_t` is given, then the `row` and `col` arguments are ignored.

In both cases, the from_addr parameter specifies the write offset relative to the buffer's start, or to the eCore's internal space.

To access system registers, the `to_addr` parameter can be one of the register symbols of the types `e_gp_reg_id_t`, `e_core_reg_id_t`, `e_chip_reg_id_t`, `e_sys_reg_id_t`.

Return value

If successful, the function returns the number of bytes read. On a failure it returns `E_ERR`.

14.4.3 e_write()

Synopsis

```
#include "e-hal.h"
ssize_t e_write(void *dev, unsigned row, unsigned col,
    off_t to_addr, const void *buf, size_t size);
```

Description

This function writes data of length `size` from the local byte buffer `buf` to a workgroup core or an external memory buffer. The argument `dev` specifies the target on which to write the data. It can be of either types `e_epiphany_t` or `e_mem_t`.

If an object of type `e_epiphany_t` is given, then the `row` and `col` arguments specify the relative target eCore coordinates in the workgroup.

If an object of type `e_mem_t` is given, then the `row` and `col` arguments are ignored.

In both cases, the `to_addr` parameter specifies the write offset relative to the buffer's start, or to the eCore's internal space.

To access system registers, the `to_addr` parameter can be one of the register symbols of the types `e_gp_reg_id_t`, `e_core_reg_id_t`, `e_chip_reg_id_t`, `e_sys_reg_id_t`.

Return value

If successful, the function returns the number of bytes written. On a failure it returns `E_ERR`.

     REV 5.13.07.10

14.5 System Control Functions

14.5.1   Overview

These functions provide some means to control different aspects of the system and a program execution.

Functions definition summary

```
int e_reset_system();
int e_reset_core(e_epiphany_t *dev, unsigned row, unsigned col);
int e_start(e_epiphany_t *dev, unsigned row, unsigned col);
int e_start_group(e_epiphany_t *dev);
int e_signal(e_epiphany_t *dev, unsigned row, unsigned col);
int e_halt(e_epiphany_t *dev, unsigned row, unsigned col);
int e_resume(e_epiphany_t *dev, unsigned row, unsigned col);
```

     REV 5.13.07.10

14.5.2   e_reset_system()

Synopsis

```
#include "e-hal.h"
int e_reset_system();
```

Description

Use this function to perform a full hardware reset of the Epiphany platform, including the Epiphany chips and the FPGA glue logic.

Special care must be taken when using this function in a multiprocessing environment not to disrupt working tasks, possibly launched by other applications.

Return value

If successful, the function returns E_OK. On a failure it returns E_ERR.

          REV 5.13.07.10

14.5.3   e_reset_core()

Synopsis

```
#include "e-hal.h"
int e_reset_core(e_epiphany_t *dev, unsigned row, unsigned col);
```

Description

Use this function to perform a soft reset of a workgroup core. The `row` and `col` parameters specify the target eCore coordinates, relative to the workgroup given by the `dev` argument.

Special care must be taken when using this function, as resetting the eCore when memory transactions, that were generated with a core read instruction from the global memory space (either `LDR` instruction or an instruction fetch from outside of the core) are not concluded can bring the system to an undefined state.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

          REV 5.13.07.10

14.5.4   *e_start()*

Synopsis

```
#include "e-hal.h"
int e_start(e_epiphany_t *dev, unsigned row, unsigned col);
```

Description

This function writes the SYNC signal to the workgroup core's ILAT register. It causes the core to jump to the IVT entry #0. Normally, this will be used after loading a program on the core.

The row and col parameters specify the target eCore coordinates, relative to the workgroup given by the dev argument.

Return value

If successful, the function returns E_OK. On a failure it returns E_ERR.

          REV 5.13.07.10

14.5.5   e_start_group()

Synopsis

```
#include "e-hal.h"
int e_start_group(e_epiphany_t *dev);
```

Description

This function writes the SYNC signal to the workgroup cores' ILAT registers. It causes the workgroup cores to jump to their IVT entry #0. Normally, this will be used after loading a program on the core.

Return value

If successful, the function returns E_OK. On a failure it returns E_ERR.

          REV 5.13.07.10

14.5.6   e_signal()

Synopsis

```
#include "e-hal.h"
int e_signal(e_epiphany_t *dev, unsigned row, unsigned col);
```

Description

This function writes the USER_INT (soft interrupt) signal to the workgroup core's ILAT register. It causes the core to jump to the IVT entry #9.

The row and col parameters specify the target eCore coordinates, relative to the workgroup given by the dev argument.

Return value

If successful, the function returns E_OK. On a failure it returns E_ERR.

                   REV 5.13.07.10

14.5.7   *e_halt()*

Synopsis

```
#include "e-hal.h"
int e_halt(e_epiphany_t *dev, unsigned row, unsigned col);
```

Description

This function halts the workgroup core's program execution. It may be useful for debug purposes.

The `row` and `col` parameters specify the target eCore coordinates, relative to the workgroup specified by the `dev` argument.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

14.5.8   e_resume()

Synopsis

```
#include "e-hal.h"
int e_resume(e_epiphany_t *dev, unsigned row, unsigned col);
```

Description

This function resumes a workgroup core's program execution that was previously stopped with a call to `e_halt()`.

The `row` and `col` parameters specify the target eCore coordinates, relative to the workgroup specified by the `dev` argument.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

                REV 5.13.07.10

14.6 Program Load Functions

14.6.1   Overview

These loader functions load an Epiphany program on an eCore or an eCore workgroup in a SPMD manner. Optionally, the loaded programs can be started immediately after loading the group.

Functions definition summary

```
int e_load(char *executable, e_epiphany_t *dev, unsigned row,
    unsigned col, e_bool_t start);

int e_load_group(char *executable, e_epiphany_t *dev, unsigned row,
    unsigned col, unsigned rows, unsigned cols, e_bool_t start);
```

      REV 5.13.07.10

14.6.2   e_load()

Synopsis

```
#include "e-hal.h"
int e_load(char *executable, e_epiphany_t *dev, unsigned row,
    unsigned col, e_bool_t start);
```

Description

This function loads an Epiphany program onto a workgroup core. The `executable` string specifies the path to the program's image. The target core workgroup is specified by the `dev` argument. The target core is specified by the `row` and `col` coordinates, relative to the workgroup.

Optionally, a loaded program can be started immediately after loading, according to the `start` parameter. When the `start` parameter is `e_true`, the program is launched after load. If it is `e_false`, the program is not launched.

Program load should be performed only when the core is in an idle or halt state. A safe way to achieve this is to use the `e_reset_system()` or `e_reset_core()` API's before the load.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`.

Note: Currently, the eHAL supports loading executable images in the form of SREC file format. Use the `e-objcopy` utility to generate an SREC image from the binary ELF executable, as described in chapters 6 and 9 of this book.

14.6.3   *e_load_group()*

Synopsis

```
#include "e-hal.h"
int e_load_group(char *executable, e_epiphany_t *dev, unsigned row,
    unsigned col, unsigned rows, unsigned cols, e_bool_t start);
```

Description

This function loads an Epiphany program onto a subgroup of a workgroup. The `executable` string specifies the path to the program's image. The target workgroup is specified by the `dev` argument. The target cores subgroup for loading the image is specified by the `row` and `col` coordinates, relative to the workgroup origin. The `rows` and `cols` parameters specify the size of the subgroups. All cores in the subgroup are loaded with the same program image.

Optionally, the loaded programs can be started immediately after loading on all cores in the subgroup, according to the `start` parameter. When the `start` parameter is `e_true`, the programs are launched after load. If it is `e_false`, the programs are not launched.

Program load should be performed only when the core is in an idle or halt state. A safe way to achieve this is to use the `e_reset_system()` or `e_reset_core()` API's before the load.

Return value

If successful, the function returns `E_OK`. On a failure it returns `E_ERR`. Some non-fatal erroneous image content generates an `E_WARN` return value. The SREC parser ignores the errors and continues the program load.

Note: Currently, the eHAL supports loading executable images in the form of SREC file format. Use the e-objcopy utility to generate an SREC image from the binary ELF executable, as described in chapters 6 and 9 of this book.

14.7 Utility Functions

14.7.1   Overview

These is a set of utility functions, provided for easing some Host application programming tasks.

Functions definition summary

```
unsigned e_get_num_from_coords(e_epiphany_t *dev, unsigned row,
    unsigned col);

void e_get_coords_from_num(e_epiphany_t *dev, unsigned corenum,
    unsigned *row, unsigned *col);

e_bool_t e_is_addr_on_chip(void *addr);

e_bool_t e_is_addr_on_group(e_epiphany_t *dev, void *addr);

e_hal_diag_t e_set_host_verbosity(e_hal_diag_t verbose);

e_loader_diag_t e_set_loader_verbosity(e_loader_diag_t verbose);
```

     REV 5.13.07.10

14.7.2   *e_get_num_from_coords()*

Synopsis

```
#include "e-hal.h"
unsigned e_get_num_from_coords(e_epiphany_t *dev, unsigned row,
    unsigned col);
```

Description

Convert a workgroup's eCore coordinates to a core number. The workgroup is defined by the `dev` argument. The core numbering is done in a "raster scan" manner, starting at the groups origin as core #0 and continuing row-wise. Thus, the number of the first core in the second row equals to the group's `cols` parameter, and the last core in the third row equals to ($3 \cdot cols-1$). The last core in the group is numbered ($rows \cdot cols-1$).

Return value

The function returns the selected core's number.

          REV 5.13.07.10

14.7.3   *e_get_coords_from_num()*

Synopsis

```
#include "e-hal.h"
void e_get_coords_from_num(e_epiphany_t *dev, unsigned corenum,
    unsigned *row, unsigned *col);
```

Description

Convert a workgroup's eCore number to core's coordinates, relative to the group origin. The workgroup is defined by the `dev` argument. The core numbering is done in a raster scan manner, starting at the groups origin as core #0 and continuing column-wise. Thus, the (`row`, `col`) coordinates of the core #0 are (`0,0`), core #`cols` coordinates are (`1,0`), and core #(3·`cols-1`) coordinates are (`2,cols-1`). The last core in the group, numbered (`rows·cols-1`), has coordinates (`rows-1,cols-1`).

Return value

The function returns the selected core's coordinates.

                   REV 5.13.07.10

14.7.4   e_is_addr_on_chip()

Synopsis

```
#include "e-hal.h"
e_bool_t e_is_addr_on_chip(void *addr);
```

Description

This function checks whether a global, 32-bit address, given by argument `addr`, is within a physical Epiphany chip's space.

Return value

The function returns `e_true` if an address is on a chip and `e_false` otherwise.

    REV 5.13.07.10

14.7.5   e_is_addr_on_group()

Synopsis

```
#include "e-hal.h"
e_bool_t e_is_addr_on_group(e_epiphany_t *dev, void *addr);
```

Description

This function checks whether a global, 32-bit address, given by argument addr, is within a core workgroup's space. The workgroup is specified by the dev argument.

Return value

The function returns e_true if an address is on a workgroup and e_false otherwise.

      REV 5.13.07.10

14.7.6   e_set_host_verbosity()

Synopsis

```
#include "e-hal.h"
e_hal_diag_t e_set_host_verbosity(e_hal_diag_t verbose);
```

Description

This function sets the verbosity level of the eHAL function calls. The levels defined from `H_D0` to `H_D4`. Level `H_D0` means no diagnostics are emitted, and any higher level designates more detailed diagnostics. This function is meant for diagnostics and debug purposes.

Return value

The function returns the old diagnostics level value.

       REV 5.13.07.10

14.7.7   *e_set_loader_verbosity()*

Synopsis

```
#include "e-hal.h"
e_loader_diag_t e_set_loader_verbosity(e_loader_diag_t verbose);
```

Description

This function sets the verbosity level of the program loader function calls, on top of the other eHAL calls diagnostics.. The levels defined from `L_D0` to `L_D4`. Level `L_D0` means no diagnostics are emitted, and any higher level designates more detailed diagnostics. This function is meant for diagnostics and debug purposes.

Return value

The function returns the old diagnostics level value.

    REV 5.13.07.10

Appendix A: Application Binary Interface (EABI)

A.1 Overview

This chapter is intended for library developers and users who develop custom assembly routines that can be called from the Epiphany C-compiler. The Epiphany ABI (EABI) is an agreement between developers that ensures interoperability between different libraries. The EABI defines a common procedure call standard and restrictions on data types and alignment. Some of the details defined by the EABI include:

- How the program (caller) should set up the machine state before calling a procedure.

- How the called procedure (callee) should preserve program state across the call.

- The right of the called procedure to alter the program state of its caller.

Conformance to this standard requires that:

- At all times, stack limits and basic stack alignment are observed.

- The routines of publicly visible interface conform to the procedure call standard.

- The data elements of publicly visible interface conform to the data layout rules. Data elements include: parameters to routines named in interfaces, static data, and all data addressed by pointer values across interfaces.

REV 5.13.07.10

A.2   Data Types and Alignment Restrictions

A.2.1   Arithmetic Data Types

Table A.1 shows the fundamental data types supported by the Epiphany architecture. Memory can be considered as an array of bytes, with each byte separately addressable by the software. The memory layout accepted is little-endian data. The least significant bit of an object is always bit 0.

Table 14.1: Arithmetic Data Types

C/C++ Type	Machine Type	Size (Bytes)	Restriction
char	Signed byte	1	
unsigned char	Unsigned byte	1	
signed char	Signed byte	1	
signed short	Signed half-word	2	Must be half-word aligned in memory
unsigned short	Unsigned half-word	2	Must be half-word aligned in memory
signed int	Signed word	4	Must be word aligned in memory
unsigned int	Unsigned word	4	Must be word aligned in memory
signed long	Signed word	4	Must be word aligned in memory
unsigned long	Unsigned word	4	Must be word aligned in memory
signed long long	Signed double word	8	Must be double-word aligned in memory
unsigned long long	Unsigned double word	8	Must be double-word aligned in memory
float	IEEE754 Single-Precision Floating Point	4	Must be word aligned in memory
double	IEEE754 Double-Precision Floating Point	8	Must be double-word aligned in memory

     REV 5.13.07.10

A.2.2   Composite Types

In additional to the fundamental data types described previously, the Epiphany supports composite types, which are a collection of one or more fundamental data types that can be processed as a single entity during procedure calls. Each one of the composite types may contain composite types and/or fundamental data types as members.

Aggregates

An aggregate is a type with members that are laid out sequentially in memory. The alignment of the aggregate shall be the alignment of its most aligned component. The size of the aggregate shall be the smallest multiple of its alignment that is sufficient to hold all of its members when they are laid out according to these rules.

Unions

A union is a composite type, where each of the members has the same address. The alignment of a union shall be the alignment of its most-aligned component. The size of a union shall be the smallest multiple of its alignment that is sufficient to hold its largest member. Structures and unions are laid out according to the fundamental data types of which they are composed. All members are laid out in declaration order.

Arrays

An array is a repeated sequence of some other type (its base type). The alignment of an array shall be the alignment of its base type. The size of an array shall be the size of the base type multiplied by the number of elements in the array.

             REV 5.13.07.10

A.3   Procedure Call Standard

A.3.1   Overview

This chapter defines the protocol for defining and using procedures in a functional language. It includes rules for stack management, register usage, and argument passing.

A.3.2   Register Usage

The Epiphany architecture includes 64 general word length purpose register. Table A.2 below shows the register usage convention in the EABI. The register usage convention acts as a contract to guarantee that a caller and callee function can work together with predictable results.

Table 14.2: Register Usage and Procedure Call Standard

Name	Synonym	Role in the Procedure Call Standard	Saved By
R0	A1	Argument/result/scratch register #1	Caller saved
R1	A2	Argument/result/scratch register #2	Caller saved
R2	A3	Argument/result/scratch register #3	Caller saved
R3	A4	Argument/result/scratch register #4	Caller saved
R4	V1	Register variable #1	Callee saved
R5	V2	Register variable #2	Callee saved
R6	V3	Register variable #3	Callee saved
R7	V4	Register variable #4	Callee saved
R8	V5	Register variable #5	Callee saved
R9	V6	Register variable #6	Callee saved
R10	V7	Register Variable #7	Callee saved
R11	V8/FP	Variable Register #8/Frame Pointer	Callee saved
R12	–	Intra-procedure call scratch register	Caller saved
R13	SP	Stack Pointer	N/A
R14	LR	Link Register	Callee saved
R15		General use	Callee saved
R16–R27		General use	Caller saved

    REV 5.13.07.10

R28-R31		Reserved for constants	N/A
R32-R43		General use	Callee saved
R44-R63		General use	Caller saved

The first four registers R0-R3 (A1-A4) are used to pass argument values into a subroutine and to return a result value from a function. They may also be used to hold intermediate values within a routine (but, in general, only between subroutine calls).

Typically, the registers R4-R11, R14-R15, and R32-R43 are used to hold the values of a routine's local variables.

A subroutine must preserve the contents of the registers R4-R11, R14-R15, and R32-R43

A.3.3   Handling Large Data Types

Fundamental types larger than 32 bits may be passed as parameters to, or returned as the result of, function calls. A double-word sized type is passed in two consecutive registers (e.g., R0 and R1, or R2 and R3).

A.3.4   Stack Management

The stack is a contiguous area of memory that may be used for storage of local variables and for passing additional arguments to subroutines when there are insufficient argument registers available. The stack implementation is full-descending, with the current extent of the stack held in the register SP (R13). The stack will, in general, have both a base and a limit though in practice an application may not be able to determine the value of either.

The stack may have a fixed size or be dynamically extendable (by adjusting the stack-limit downwards).The rules for maintenance of the stack are divided into two parts: a set of constraints that must be observed at all times, and an additional constraint that must be observed at a public interface. At all times the following basic constraints must hold:

- Stack-limit < SP <= stack-base. The stack pointer must lie within the extent of the stack.

- `(SP mod 4)=0`. The stack pointer must at all times be aligned to a word boundary (where `SP` is the value of register `R13`).

- A process may only access (for reading or writing) the closed interval of the entire stack delimited by [`SP`, (stack-base – 1)].

- The stack frame must be double-word aligned.

A.3.5 Subroutine Calls

The Epiphany includes '`BL`' and '`JALR`' instructions for calling subroutines. These instructions transfer the sequentially next value of the program counter—the return address —into the link register (`LR`) and the destination address into the program counter (`PC`). The result is to transfer control to the destination address, passing the return address in `LR` as an additional parameter to the called subroutine. Control is returned to the instruction following the `BL/JALR` when the return address is loaded back into the `PC` using the `JR/RTS` instruction.

A.3.6 Procedure Result Return

The manner in which a result is returned from a procedure is determined by the type of the result.

A data type that is smaller than 4 bytes is zero or sign-extended to a word and returned in `r0`.

A word-sized data type (e.g., `int`, `float`) is returned in `r0`.

A double-word sized data type (e.g., `long long, double`) is returned in `r0` and `r1`.

A Composite Type not larger than 4 bytes is returned in `r0`. The format is as if the result had been stored in memory at a word-aligned address and then loaded into `r0` with an `LDR` instruction. Any bits in `r0` that lie outside the bounds of the result have unspecified values.

A Composite Type larger than 4 bytes, or whose size cannot be determined statically by both caller and callee, is stored in memory at an address passed as an extra argument when the function was called.

 REV 5.13.07.10

A.3.7   Parameter Passing

The base standard provides for passing arguments in core registers (`r0-r3`) and on the stack. For subroutines that take a small number of parameters, only registers are used, greatly reducing procedure call overhead.

         REV 5.13.07.10

Appendix B: Board Support Packages

B.1   Board Support Package Descriptor File

The Epiphany uses an XML format that allows the flexibility of defining custom boards so that the Epiphany SDK can be automatically configured to work correctly. Standard boards and evaluation kits officially supported by Adapteva are distributed with XML board support packages (BSP) pre-written.

The following example code shows the XML configuration file for the Epiphany Multicore Evaluation Kit (EMEK).

```xml
<?xml version="1.0"?>
<platform version="1" name="AAHM" lib="libftdi_target.so" libinitargs="">
   <chips>
      <chip version="3" id="(32,36)" rows="4" cols="4" host_base="0x12000000"
                                                core_memory_size="0x8000">
         <ioregs col="2" row="2"/>
      </chip>
      <chip version="5" id="(32,32)" rows="1" cols="1" host_base="0x12000000"
                                                core_memory_size="0x8000" />
   </chips>
   <external_memory>
      <bank name="EXTERNAL_DRAM_0" start="0x80000000" size="0x01000000" />
      <bank name="EXTERNAL_DRAM_1" start="0x81000000" size="0x01000000" />
      <bank name="EXTERNAL_SRAM"   start="0x920f0000" size="0x00010000" />
   </external_memory>
</platform>
```

The following sections define the different tags used in the ESDK XML configuration format.

The `<platform>` Element

The `<platform>` tag will serve as the root tag for the document and has the following attributes:

name

> This attribute provides a human-readable name for the platform being described by the document. An example value would be "AAHM" for the Epiphany Multicore Evaluation Kit. This name is used to acknowledge a successful startup sequence by the 'e-server'.

lib

> The lib attribute defines the name of the library containing the software needed to access the platform from a Linux host. In the case of the EMEK, a lib might be "ftdilib_target".

libinitargs

> This attribute defines a string that will be passed to the library's `init_platform()` function. The string is actually embedded in a structure containing other needed information, and a pointer to this structure is passed to `init_platform()`. For the EMEK, the string will be `NULL`, but other systems may require additional information provided by this string.

The `<chips>` Element

The `<chips>` tag will have the following sub-elements:

`<chips>`

> This tag serves as a container for at least one `<chip>` tag. It has the following sub-elements:

`<chip>`

> This tag defines a single chip within the platform. It has the following attributes:

version

> This required attribute defines the version number of the chip, and should be unique for each incarnation of the chip.

id

> This attribute defines the chip's row and column id. The values specified here should match the YID and XID pin settings of the device, meaning it does not include the portions of the coordinates that are internal to the chip. In the case of a 16-core device, YID and XID are four bits each.

rows

> This attribute defines the number of rows in the device.

cols

> This attribute defines the number of columns in the device.

host_base

> This attribute gives the library a base address that it may use for addressing the device's host. In the case of the AAHM on an S3 Devkit, it would be equal to 0x12000000. This is the beginning of an address range within the FPGA on the S3 Devkit containing control registers (such as reset) and an area of memory used as a readback destination. Use of this field is specific to the library.

     REV 5.13.07.10

`core_memory_size`

This attribute defines the amount of memory (in bytes) internal to each core in the device. If omitted, it defaults to 32K bytes.

The `<chip>` tag may optionally have the following sub-elements:

`<cores>`

This tag defines an array of <core> tags. If this array is omitted, the parser will assume that all the cores within the device (as defined by the `rows` and `cols` attributes) are present.

`<core>`

The core tag can be used to explicitly identify the cores within the device. It has the following attribute:

`id`

This attribute identifies the local coordinates of the core within the chip. That is, the internal portion of the core's row and column coordinates.

`<ioregs>`

This subelement is used to identify the location of any I/O registers within the chip, such as link port control or GPIO registers. If it is omitted, the device is assumed to have no I/O registers. If present, it requires the following attributes:

col

This attribute defines the column in which the north and south control registers lie. The north registers will reside in row 0, and the south registers will reside in the chip's maximum row number.

row

This attribute defines the row in which the east and west control registers lie. The east registers will reside in column 0, and the west registers will reside in the chip's maximum column number.

     REV 5.13.07.10

The `<external_memory>` Element

The `<external_memory>` tag may have the following optional sub-elements:

`<external_memory>`
>   This element defines an array of external memory banks. It requires at least one `<bank>` sub-element.
>
>   `<bank>`
>>   This tag identifies one bank of external memory available to the chip. It has the following attributes, all of which are required.
>>
>>   `name`
>>>   This attribute names the memory bank, and could be used by a linker script code generator.
>>
>>   `start`
>>>   This attribute defines the starting address of the memory bank (as dereferenced by an Epiphany core).
>>
>>   `size`
>>>   This attribute defines the size (in bytes) of the memory bank.

Note: The current XML parser is implemented in C++. In order to keep the eHAL library C compatible when processing the HDF, we could not use this parser, and plan to replace it with a C implementation.

Until the replacement is integrated in, the eHAL does not support the XML file format. Instead, a simplified text file was defined. This file has the `.hdf` extension, and includes information similar to that of the XML HDF files. Here's what a sample file looks like:

```
// Platform description for the
// ZedBoard/512MB/E16G3
PLATFORM_VERSION    0x00000300
ESYS_REGS_BASE      0x808f0f00

NUM_CHIPS                     1
CHIP                    E16G301
CHIP_ROW                     32
CHIP_COL                      8

NUM_EXT_MEMS                  1
EMEM                   ext-DRAM
EMEM_BASE_ADDRESS    0x1e000000
EMEM_EPI_BASE        0x8e000000
EMEM_SIZE            0x02000000
EMEM_TYPE                  RDWR
```

                REV 5.13.07.10

Structure Definition

The XML file will be parsed by the top-level code in whatever tool needs the information, and its data will be captured in a C-language structure. A pointer to this structure can then be passed to any of the tool's subcomponents (such as libftdi_target in the case of gdbserver). The structures required are defined below:

```
// Structure describing each chip in the system
typedef struct
{
  char    *version;         // version of the chip
  unsigned yid;             // chip coordinates (YID[0:3] pins)
  unsigned xid;             // chip coordinates (XID[0:3] pins)
  unsigned ioreg_row;       // row within chip where I/O registers are located
  unsigned ioreg_col;       // column within chip where I/O registers are located
  unsigned num_rows;        // number of rows in the chip
  unsigned num_cols;        // number of cols in the chip
  void    *host_base;       // base address of host (for reset, readback, etc)
  size_t   core_memory_size; // bytes of internal memory in each core
} chip_def_t;

// Structure describing each external memory segment available to the chips.
typedef struct
{
  char  *name; // name of the memory segment (can be used in linker script)
  void  *base; // base address of memory segment
  size_t size; // number of bytes in the memory segment
} mem_def_t;

// Structure containing the data parsed from the XML file and
// passed to a subordinate function.
typedef struct
{
  char        *name;        // name of the platform (i.e. "AAHM")
  char        *lib;         // name of platform library (i.e. "libftdi_target")
  char        *libinitargs; // additional argument string passed to lib init fxn
  unsigned     num_chips    // number of elements in chips[] array
  chip_def_t *chips;        // array of chips[]
  unsigned     num_banks;   // number of elements in ext_mem[] array
  mem_def_t  *ext_mem;      // array of ext_mem[]
} platform_definition_t;
```

 REV 5.13.07.10

Appendix C: Changes from Previous Revisions

Revision	Changes
5.13.07.09	Updated installation procedure
	Updated tools documentation links
	Added note on Eclipse
	Added chapter on E-UTILS for e-reset, e-loader, e-read, e-write, e-hw-rev
	Added chapter on the new Epiphany Programming Model
	Updated the E-LIB chapter:
	* Added description for the new workgroup config objects
	* Revised enumeration type and symbol names
	* Split register enumeration to functional groups and added _REG_
	* Add some GP regs and a couple more system regs
	* Modified the following functions:
	-- Removed e_sysreg_read(),e_sysreg_write(), e_gid(), e_gie(), e_gie_restore(), e_irq_disable(), e_irq_enable(), e_irq_raise(), e_irq_remote_raise(), e_irq_lower(), e_irq_restore(), e_mutex_destroy(), e_coreid_from_address(), e_address_from_coreid(), e_coreid_origin()
	-- Added e_reg_read(), e_reg_write(), e_irq_attach(), e_irq_global_mask(), e_irq_mask(), e_irq_set(), e_irq_clear(), e_ctimer_wait(), e_read(), e_write(), e_dma_wait(), e_dma_set_desc(), e_barrier_init(), e_barrier(), e_get_global_address()
	-- Modified e_ctimer_set(), e_ctimer_stop(), e_dma_copy(), e_dma_start(), e_mutex_init(), e_mutex_lock(), e_mutex_trylock(), e_mutex_unlock(), e_is_oncore(), e_neighbor_id()
	Updated the E-HAL chapter:
	* Revised enumeration symbols
	* Split register enumeration to functional groups and added _REG_
	* Modified the System Control section:
	-- Added e_start_group()

      REV 5.13.07.10

	-- SWI now called USER_INT
	-- e_halt() and e_resume() now implemented
	-- Modified e_set_host_verbosity(), e_set_loader_verbosity()
4.13.03.30	

       REV 5.13.07.10