CEVA®
The DSP Powerhouse

CEVA-Toolbox™

CEVA-XM4™ Compiler Reference Guide

Rev 15.1.0

Feb 2015

DSP Division

Documentation Control

History Table:

Version	Date	Description	Remarks	Author
15.0.0.Beta	18/2/2015	1st version		David M. Castiel
15.0.0.Beta2	4/3/2015	CEVA case #16679	Update to passing comma separated arguments to Assembler via Compiler driver	
15.0.0	12/4/2015	bug fixes		
15.1.0	14/7/2015	bug fixes		

Disclaimer and Proprietary Information Notice

The information contained in this document is subject to change without notice and does not represent a commitment on any part of CEVA®, Inc. CEVA®, Inc. and its subsidiaries make no warranty of any kind with regard to this material, including, but not limited to implied warranties of merchantability and fitness for a particular purpose whether arising out of law, custom, conduct or otherwise.

While the information contained herein is assumed to be accurate, CEVA®, Inc. assumes no responsibility for any errors or omissions contained herein, and assumes no liability for special, direct, indirect or consequential damage, losses, costs, charges, claims, demands, fees or expenses, of any nature or kind, which are incurred in connection with the furnishing, performance or use of this material.

CEVA®, CEVA-XC™, CEVA-XC321™, CEVA-XC323™, CEVA-Xtend™, CEVA-XC4000™, CEVA-XC4100™, CEVA-XC4200™, CEVA-XC4210™, CEVA-XC4400™, CEVA-XC4410™, CEVA-XC4500™, CEVA-TeakLite™, CEVA-TeakLite-II™, CEVA-TeakLite-III™, CEVA-TL3210™, CEVA-TL3211™, CEVA-TeakLite-4™, CEVA-TL410™, CEVA-TL411™, CEVA-TL420™, CEVA-TL421™, CEVA-Quark™, CEVA-Teak™, CEVA-X™, CEVA-X1620™, CEVA-X1622™, CEVA-X1641™, CEVA-X1643™, Xpert-TeakLite-II™, Xpert-Teak™, CEVA-XS1100A™, CEVA-XS1200™, CEVA-XS1200A™, CEVA-TLS100™, Mobile-Media™, CEVA-MM1000™, CEVA-MM2000™, CEVA-SP™, CEVA-VP™, CEVA-MM3000™, CEVA-MM3100™, CEVA-MM3101™, CEVA-XM™, CEVA-XM4™, CEVA-X2™ CEVA-Audio™, CEVA-HD-Audio™, CEVA-VoP™, CEVA-Bluetooth™, CEVA-SATA™, CEVA-SAS™, CEVA-Toolbox™, SmartNcode™ are trademarks of CEVA, Inc.

All other product names are trademarks or registered trademarks of their respective owners

Support

CEVA® makes great efforts to provide a user-friendly software and hardware development environment. Along with this, CEVA® provides comprehensive documentation, enabling users to learn and develop applications on their own. Due to the complexities involved in the development of DSP applications that may be beyond the scope of the documentation, an on-line Technical Support Service (support@ceva-dsp.com) has been established. This service includes useful tips and provides fast and efficient help, assisting users to quickly resolve development problems.

How to Get Technical Support:

FAQs: Visit our web site http://www.ceva-dsp.com or your company's protected page on the CEVA® website for the latest answers to frequently asked questions.

Application Notes: Visit our website http://www.ceva-dsp.com or your company's protected page on the CEVA website for the latest application notes.

Email: Use CEVA's central support email address support@ceva-dsp.com. Your email will be forwarded automatically to the relevant support engineers and tools developers who will provide you with the most professional support in order to help you resolve any problem.

License Keys: Please refer any License Key requests or problems to sdtkeys@ceva-dsp.com. Refer to ***SDT Installation & Licensing Scheme Guide*** for SDT license keys installation information

Email: support@ceva-dsp.com
Visit us at: www.ceva-dsp.com

List of Sales and Support Centers

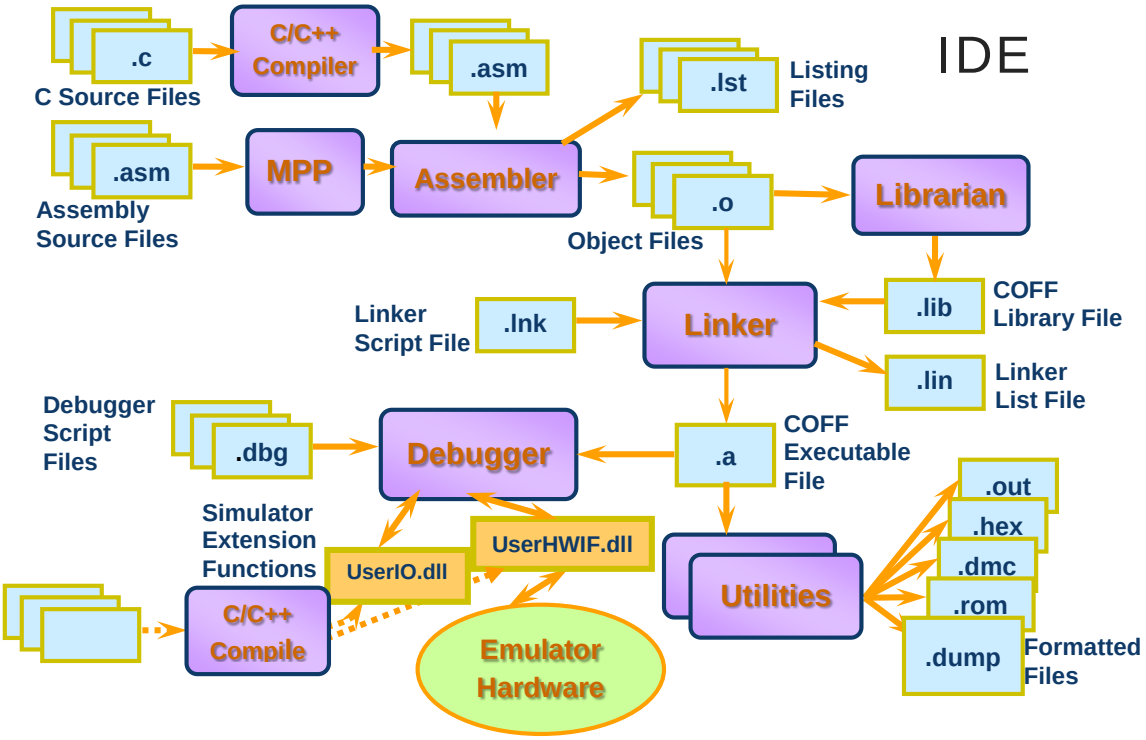
Israel	USA	Ireland	Sweden
2 Maskit Street P.O.Box 2068 Herzelia 46120 Israel **Tel:** +972 9 961 3700 **Fax:** +972 9 961 3800	1943 Landings Drive Mountain View, CA 94043 USA **Tel:** +1-650-417-7923 **Fax:** +1-650-417-7924	Segrave House 19/20 Earlsfort Terrace 3rd Floor Dublin 2 Ireland **Tel**: +353 1 237 3900 **Fax**: +353 1 237 3923	Klarabergsviadukten 70 Box 70396 107 24 Stockholm, Sweden **Tel**: +46(0)8 506 362 24 **Fax**: +46(0)8 506 362 20
China (Shanghai)	**China (Beijing)**	**China Shenzhen**	**Hong Kong**
Room 517, No. 1440 Yan An Road (C) Shanghai 200040 China **Tel:** +86-21-22236789 **Fax:** +86 21 22236800	Rm 503, Tower C, Raycom InfoTech Park No.2, Kexueyuan South Road, Haidian District Beijing 100190, China **Tel:** +86-10 5982 2285 **Fax:** +86-10 5982 2284	2702-09 Block C Tiley Central Plaza II Wenxin 4th Road, Nanshan District Shenzhen 518054 **Tel:** +86-755-86595012	Level 43, AIA Tower, 183 Electric Road, North Point Hong Kong **Tel:** +852-39751264 **:**
South Korea	**Taiwan**	**Japan**	**France**
#478, Hyundai Arion, 147, Gumgok-Dong, Bundang-Gu, Sungnam-Si, Kyunggi-Do, 463-853, Korea **Tel**: +82-31-704-4471 **Fax**:+82-31-704-4479	Room 621 No.1, Industry E, 2nd Rd Hsinchu, Science Park Hsinchu 300 Taiwan R.O.C **Tel**: +886 3 5798750 **Fax**: +886 3 5798750	3014 Shinoharacho Kasho Bldg. 4/F Kohoku-ku Yokohama, Kanagawa 222-0026 Japan **Tel:** +81 045-430-3901 **Fax:** +81 045-430-3904	RivieraWaves S.A.S 400, avenue Roumanille Les Bureaux Green Side 5, Bât 6 06410 Biot - Sophia Antipolis, France **Tel:** +33 4 83 76 06 00 **Fax:** +33 4 83 76 06 01

Table of Contents

List of Tables

1. Introduction

CEVA-Toolbox™ Software Development Tools (SDT) Diagram



1.1 Scope

This guide is the reference guide for the **CEVA-Toolbox™ C/C++ Compiler for CEVA-XM4™**. The guide describes the various command line options, main features, built-in library functions, the calling convention for linking C/C++ code with assembly code generated objects and how to use in-line assembly statements embedded in a C/C++ program. In addition, the guide describes the supported standard library functions and provides the user with programming hints and examples to improve the quality and compactness of the generated code. This guide assumes the user is already familiar with the ANSI C/C++ programming language and the architecture of the CEVA-XM4.

For CEVA-XM4, the ANSI C/C++ cross Compiler was created by porting the source code of the ORC (Open Research Compiler) project.

Unlike the Compiler that was imported and tailored to the CEVA-XM4 specific architectures from the ORC Project, the Assembler, Linker, Archiver, Utilities, Debugger, Profiler and IDE (Integrated Development Environment) used with the CEVA-XM4 were written from scratch by CEVA.

This **pdf** guide format is based on the HTML Help format. HTML Help uses the chapters and pages metaphors to represent categories and topics in the Table of Contents. Chapter icons represent categories or chapters for organizing the guide's topics. Page icons represent the individual units of information or pages. In HTML Help, both chapters and pages can display topics when clicked.

1.2 Applicable Documents

1. *CEVA-XM4 Architecture Specifications*
2. *CEVA-Toolbox™ Debugger Users Guide*
3. *CEVA-Toolbox™ C/C++ Assembler/Linker Users Guide*
4. *CEVA-Toolbox™ IDE Users Guide*
5. *CEVA-Toolbox™ Software Development Tools Installation Guide*
6. *Linking Applications Guidelines Using the CEVA-Toolbox Linker Application Note*
7. *CEVA DSPs Evaluation Development Platform (EDP) Reference Guide*
8. *The C Programming Language, 2nd ed. 1988, by B. Kernighan and D. Ritchie, Prentice Hall*
9. *Using and Porting GNU CC, Version 2.95, by Richard Stallman, FSF Inc.*
10. *Using the GNU Compiler Collection (GCC), by Richard Stallman, FSF Inc.*
11. *GNU's Bulletin, FSF Inc. 675 Massachusetts Ave, Cambridge, MA 02139 USA.*
12. *The Annotated C++ Reference Manual, by M.A Ellis and B. Stroustrup.*
13. *The ANSI/ISO C++ standard.base document.*

2. Getting Started

This chapter contains miscellaneous Getting Started sections that include the following:

- **Compiler Invocation**
- **Specifying File Names**
- **Compiling C++ Programs**
- **Linking**
- **Optimizations**

2.1 Compiler Invocation

Before proceeding with using the Compiler it is recommended to verify the successful completion of the specific **CEVA-Toolbox Software Development Tools** installation as described in ***SDT Installation & Licensing Scheme Guide***.

Following is the Compiler or the Compiler Driver command line invocation syntax used for all the cores:

<**CEVA DSP Core** specific mnemonics>cc [options] InputFile.c

Where the **CEVA-Toolbox Compiler** invocation stands for any of the notations used in Table 2-1: CEVA DSP Cores Compiler Invocation below. The table shows the names used for the Compiler Driver, Compiler's Configuration file, C Compiler and C++ Compiler for each one of the CEVA DSPs.

Table 2-1: CEVA DSP Cores Compiler Invocation

C/C++ Compiler Driver	**xm4cc**
C/C++ Preprocessor	**xm4cpp**
Compiler Driver Config File	**xm4cc.cfg**

Compiler Driver activation can involve up to four stages (in order of appearance): preprocessing, compiling, assembling and linking. The first three stages apply to an individual source file, and end by producing an object file. Linking combines all the object files (those newly compiled, and those specified as input) into an executable file. For any given input file, the file name suffix determines the type of compilation that is done (Refer to *Specifying File Names* chapter).

A number of command line options are available for flexible control over the compilation process. The list of command line options is divided into the following option categories:

- General options
- C Language Options
- C++ Language Options
- Warning Options
- Debugging Options
- Preprocessor Options
- Code Generation Options
- Optimization Options
- Specific Optimizations

Examples:

- To invoke the Preprocessor, Compiler and Assembler only, add debugging information (and disable all optimizations), enter:

  xm4cc -c -g myprog.c

- To invoke the Preprocessor, Compiler, Assembler and Linker, enable all default optimizations, enter:

  xm4cc myprog.c

- To invoke the Preprocessor, Compiler and Assembler only, make sure that only ANSI C/C++ code is compiled, print the name of each header file used, to enable all warnings and disable optimizations, enter:

  xm4cc -c -ansi -H -Wall -O0 myfile.c

- To invoke the Preprocessor, Compiler, Assembler and Linker, use a library named mylib.lib, add debugging information and use speed optimization level 3, enter:

  xm4cc -g –O3 -lmylib.lib mycode.c

- To invoke the Compiler, Assembler and Linker, for a C++ file , enter:

  xm4cc myprog.cpp

It is possible to put the Compiler command line switches in a file. The switches should be separated by new lines, for example:

<file.txt begin>

-O4

-Wa,-p

file.c

<file.txt ends>

In order to use these switches, invoke the Compiler driver with **@** followed by a filename as a command line switch:

xm4cc @tmp.txt

For full details on all the supported options, refer to the *CEVA-XM4 Compilation Options* chapter.

The following chapter deals with the meaning of each filename extension to the Compiler Driver.

2.2 Specifying File Names

Table 2-2 below details the various filenames recognized by the Compiler driver and the respective actions (taken or not taken) by the Compiler driver when filenames of that type are encountered on the command line:

Table 2-2: File Names Extensions

File Name Suffix	Extension Effect:
.c	C source code that must be preprocessed by CPP
.cpp	C++ source code that must be preprocessed by CPP
.i	Preprocessed C source core
.ii	Preprocessed C++ source code
.h	C header file (not to be compiled or linked)
.s	Assembly code that should not be preprocessed by MPP
.asm	Assembly code that must be preprocessed by MPP
.other	An object file to be fed straight into link phase

2.3 Compiling C++ Programs

C++ source files conventionally use one of the .cc, .cpp, or .cxx suffixes (file name extension). Preprocessed C++ files use the .ii suffix. The Compiler driver used for the **CEVA DSP Cores** family recognizes files with these extensions and compiles them as C++ programs. The Compiler invocation is done the same way as for compiling C programs.

C++ Level of Support

All the main features of C++ language according to ***The Annotated C++ Reference Manual*** by Margaret A. Ellis and Bjarne Stroustrup and the ANSI C++ based document are supported, including:

- ✓ Class type / objects
- ✓ Inheritance
- ✓ Function overloading
- ✓ Friend functions
- ✓ Virtual and pure virtual functions
- ✓ Polymorphism
- ✓ Templates
- ✓ Namespaces
- ✓ Standard C++ library (including STL)

The following C++ features are currently not supported:

1. Exception handling - currently not supported due to the overhead incurred in code size.
2. C++ style formatted strings (sstream.h).

2.4 Linking

There are two options to invoke the C/C++ Compilers in reference to the Linker:

The first option is to invoke the Compiler automatically followed by the Assembler and Linker.

The second option is to invoke the Assembler and Linker phases separately from compilation (using -S, -c Compiler options). When application's variables and/or functions mapping in the data or code spaces is irrelevant, it is possible to invoke the C Compiler driver followed automatically by the Assembler and the Linker using the default linking algorithm.

Example 1:

xm4cc -g myfile.c

The C/C++ Compiler compiles the source file, myfile.c, creating myfile.asm and invokes both the Assembler and Linker. The Linker script file "myfile.lnk" is automatically created by the C/C++ Compiler driver for the Linker and includes the myfile.o object. The output accepted is the executable file dsp.a that can be loaded by the Debugger.

Example 2:

xm4cc -g -lc:\libs\mylib.lib myfile.c file2.c

In this case, the Compiler compiles both source files, myfile.c and file2.c, temporarily creating "myfile.s" and "file2.s" respectively. This is followed by the Assembler invocation for each ".s" files, creating myfile.o and file2.o (and deletes ".s" files). Finally, the driver invokes the Linker while “including” the following objects: myfile.o, file2.o and following libraries: c:\libs\mylib.lib in the Linker script file. The output generated is the "dsp.a" executable file that can be loaded by the Debugger.

Example 3:

xm4cc -g -lmylib.lib lib2.lib file4.o file3.o file2.c myfile.c

In this case , the Compiler compiles both source files, "file2.c" and "myfile.c", creating (temporarily) file2.s and myfile.s respectively. This is followed by Assembler invocation for each ".s" file, creating "file2.o" and "myfile.o" (deleting the ".s" files). Finally, the driver invokes the Linker while “including” the following object files: file2.o, myfile.o, file4.o, file3.o and following libraries: mylib.lib, lib2.lib in the Linker script file.

The output generated is the executable "dsp.a" file. The libraries are prefixed by a pathname according to the ordered search in the current directory $(XM4TOOLS). Alternatively, the search in XM4TOOLS\LIBS path is used for the CEVA-XM4. If the libraries or object files contain data (data sections with simple variables, arrays, or structures) or code (code sections with functions or data sections with constant tables) needed to be mapped to defined addresses, a Linker script file has to be manually prepared and invoked by the C Compiler without the Linker (using the -c option).

Important:

1. *Refer to “Linking Application Guidelines Using CEVA-Toolbox Linker” application note for more details on how to prepare a custom Linker script file.*
2. *Refer to* “CEVA-Toolbox™ Assembler and Linker User’s Guide”, *for further information about the Linker.*

2.5 Optimizations

The C Cross Compiler includes many optimization switches. Some of these are machine independent, others are machine dependent (**CEVA DSP Core** specific).

Table 2-3 below lists some of the more important ones, including their desired effect. All of the optimizations specified below are available as defaults (-O3 option) unless otherwise noted. The set of switches that control the behavior of various compiler optimizations and code generation is described in the Compilation Options chapter.

Table 2-3 :Compiler Optimizations - General

Optimization Type	Optimization Name	Effect
Machine independent	Function inlining	Reduces execution time of short functions
	Constant folding	Saves instructions with Immediate constant operands
	Dead code/storage elimination	Dead code/storage Elimination
	Arithmetic simplification	Saves arithmetic Operations / instructions
	Jump optimization	Changes code to eliminate jumps over jumps or jumps to jumps
	Jump threading	removes code associated with unnecessary jump conditions
	Loop optimization	invariant code motion, strength reduction, loop unrolling
	Data flow analysis	identifies life ranges of variables
	Instruction combination	replaces two or more simple instructions by one complex instruction
	Register class referencing	use Rn registers for pointers, accumulators for data, etc.
	Local register allocation	allocate registers to local variables in basic blocks
	Global register allocation	allocate registers to

Optimization Type	Optimization Name	Effect
		variables used in more than one basic block
Machine dependent	Bkrep optimization	reduce execution time and program size, improve readability
	Mac optimization	reduce execution time and program size
	Auto increment addressing	reduce execution time and program size
	Instruction scheduling	Reduce the number of nops generated by pipeline hazards. Generate parallel code.

3. C/C++ Compiler Description

This chapter contains the following sections:

- **CEVA-XM4 Compilation Options**
- **CEVA-XM4 Core Specific Features**
- **Default Calling Convention**

3.1 CEVA-XM4 Compilation Options

This chapter details the more useful compilation options. The chapter is divided into the following option categories:

- **General**
- **C Language**
- **Warning**
- **Debugging**
- **Preprocessor**
- **Code Generation**
- **Optimization**
- **CEVA-XM4 Specific**
- **CEVA-XM4 Function-Specific Optimizations and Options**

3.1.1. CEVA-XM4 General Options

Table 3-1 : CEVA-XM4 General Options

Option	Description, Effect, Purpose and Usage
-emul	Builds an executable that its target is an emulator board, rather than software simulator. This switch affects the I/O library taking part in the linking process. Either **libios.lib** for simulation mode, or **libioe.lib** for emulation mode. Additionally, the default linker script file reserves data memory locations for the emulation reserved memory areas (mmio, mailbox).
-c	Compiles or assembles the source files but do not link. Without the linking stage, the output is in the form of an object file for each source file.
-S	Stop after the stage of compilation proper, do not assemble. The output, is in the form of an Assembler file for each non-Assembler input. Input files not requiring compilation are ignored.
-E	Stops after the C preprocessing stage. The output is in the form of a preprocessed source code sent to the standard output.
-o Fname	Places the output in the specified file, regardless of the output type.
-x c	The next source files are C programs.
-x c++	The source files are C++ programs.
-v	Prints (to stderr) the commands executed to run all compilation stages including all version numbers.
-Wa,opt1,opt2..	Passes on **Assembler** invocation switch options for the assembling phase. Options are separated by commas without spaces. For passing a list of arguments to an Assembler option, each comma should be preceded with a backslash. For example: • -Wa,-p\,-d\,DEF1\,-d\,DEF2,-saveTempFiles • Equivalent to Assembler flags: -p,-d,DEF1,-d,DEF2 -saveTempFiles
-Wl, opt1,opt2..	Passes on **Linker** invocation switch options for the linking phase. Options are separated by commas without spaces.
-mfile-io	Activates the standard C stream I/O implementation allowing to perform file operations on the host's file system through the Debugger. This switch defines the macro FILE_IO=1, and adds **fileio.lib** to the libraries used in the Linker script file.
-save-temps -keep	Instructs the Compiler to save all the temporary files that are generated in the compilation process in the directory where the Compiler is invoked unless otherwise specified (using -o etc').
-mvc-error-format	Under this option, error printing format is changed to conform with Microsoft Visual Studio error message format, in order to enable compilation error browsing when using MSVS environment as an IDE.
-mfilename-in-section-name	Adds the name of a file into the generated assembly file as a prefix of the following sections: Code section : .text Data sections : .data(.ndata), .bss, const_data

Option	Description, Effect, Purpose and Usage
-mfunc-prof-level-<X>	Adds a unique profiling label to each function's start and finish where X is an integer denoting a predefined group of functions. The command line option can be used to change the profiling level of the functions in the compiled file. Default profiling level is 3. The profiling level can be used for filtering the profiling results in the Debugger level and to display only specific functions that are marked with specific profiling level. For more details refer to the Debugger manual.
-mmsg-full-path	Adds files' full path in error and warning messages.
-mquiet	Informs the compiler not to output compiler messages, other than errors and warnings.
-muse-old-cpp	Invoke the GCC 2.9.5 based preprocessor, rather than the default GCC 4.2.0 based preprocessor, for backward compatibility. The GCC 4.2.0 preprocessor is invoked by default for C programs. The GCC 2.9.5 preprocessor is invoked by default for C++ programs, and when –MM is specified (for creating dependencies). Apply -muse-old-cpp for enabling the following deprecated C preprocessor constructs: 1. #ifdef [CONSTANT] 2. Usage of ## (token concatenation) where the result might form an invalid preprocessor token

3.1.2. CEVA-XM4 C Language Options

Table 3-2 : CEVA-XM4 C Language Options

Option	Description, Effect, Purpose and Usage
-ansi	Supports all ANSI standard C programs. This turns off certain features of the Compiler that are incompatible and predefined macros. The alternate keywords *__asm__*, *__extension__*, *__inline__* and *__typeof__* continue to work despite -ansi. You would not want to use them in an ANSI C program, of course, but it is useful to put them in header files that might be included in compilations done with –ansi.
-fno-asm	Do not recognize *asm*, *inline* or *typeof* as keywords. -ansi implies -fno-asm. You can use *__asm__*, *__inline__* and *__typeof__* instead.
-fno-builtin	Do not recognize built-in functions (*) that do not begin with two leading underscores. The Compiler normally generates special code to handle built-in functions more efficiently (e.g. using bkrep instructions) but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library.
-ftraditional	Attempts to support some aspects of traditional C compilers. (extern declarations take effect globally, including implicit function definitions, typeof, inline, signed, const, volatile are not recognized). Cannot be used if you include header files that rely on ANSI C features.
-fsyntax-only	Checks for syntax errors, then stop.

(*) Built-in functions (Partial list): abort, abs, alloca, cos, exit, fabs, __exp, labs, memcmp, memcpy, sin, sqrt, strcmp, strcpy, strlen.

For further information refer to the Run Time Library Functions - Details chapter.

3.1.3. CEVA-XM4 Warning Options

In general, warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error. You can request many specific warnings with options beginning **-W**, for example, **-Wimplicit** to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning **-Wno** to turn off warnings; for example, **-Wno-**implicit. This guide lists only one of the two forms, whichever is not the default.

Table 3-3 : CEVA-XM4 Warning Options

Option	Description
-pedantic	Issues all the warnings demanded by strict ANSI standard C. Reject all programs that use forbidden extensions.
-Wimplicit	Warns whenever a function or parameter is implicitly declared.
-Werror-implicit-function-declaration	Results in an error whenever a function is used before being declared. The form -Wno-error-implicit-function-declaration is not supported.
-Wreturn-type	Warns whenever a function is defined with a return-type that defaults to int. Also warn about any return statement with no return-value in a function whose return-type is not void.
-Wunused-parameter	Warns whenever a function parameter is unused aside from its declaration. Refer to Notes 1 & 2
-Wunused-variable	Warns whenever a local variable is unused aside from its declaration. Refer to Notes 1 & 2
-Wunused-function	Warns whenever a static function is declared but not defined or a non-inline static function is unused. Refer to Notes 1 & 2
-Wunused-label	Warn whenever a label is declared but not used. Refer to Notes 1 & 2
-Wunused-value	Warn whenever a statement computes a result that is explicitly not used. Refer to Notes 1 & 2
-Wunused	All the above -Wunused options combined.
-Wall	All of the above -W options combined.
-Wbad-function-cast	Warns whenever a function call is cast to a non-matching type.

Option	Description
	For example, warn if int malloc() is cast to anything.
-Wconversion	Warns whenever a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or sign of a fixed point argument except when the same as the default promotion. Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment x = -1. if x is unsigned. But do not warn about explicit casts like (unsigned) -1.
-Wstrict-prototypes	Warn if a function is declared or defined without specifying the argument types.
-Wmissing-declarations	Warn if a global function is defined without a previous declaration. This warning is issued even if the definition provides a prototype. The purpose is to detect global functions that are not declared in header files.
-Werror	Make all warnings into errors.

Notes:

1. *These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing.*
2. *These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared volatile, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.*
3. *Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.*

3.1.4. CEVA-XM4 Debugging Options

In order to facilitate debugging using the CEVA-Toolbox debugger, debugging info should be produced using the debugging options detailed below. Debugging capabilities are best utilized and most reliable when applied with low optimization levels (as explained in section 3.1.7 CEVA-XM4 Optimization Options).

Note that producing debugging information (using –gX options) prevents the compiler from removing static functions from the final COFF file, even if the said functions are unreferenced. As such, static functions which are inlined (even if lacking any additional reference) are also included in the COFF file. In order to prevent this behavior, usage of debugging information generation options must be avoided.
[Non-static functions are never removed by the compiler, as they might be referenced from external modules. Removal of such functions can be done by utilizing the linker option '-removeUnRefFunc' (see CEVA-Toolbox Assembler & Linker Users Guide).]

Table 3-4 : CEVA-XM4 Debugging Options

Option	Description
-g/0/1/2	Produces debugging information in COFF format for the CEVA-Toolbox Debugger. If an optimization level is not specified (-O switch is not used), and –g0/1/2 is specified, optimizations are disabled by default. Currently there is no difference between –g, -g0, -g1 and –g2.
-g3	Similar to **-g**, generates also extra instructions that are needed for enabling **Function's Call Stack Window** Debugger's support (3).
-mrelative-path	Compiler generates relative path references to the source files unless absolute path is explicitly used. Default is absolute path.

3.1.5. CEVA-XM4 Preprocessor Options

Table 3-5 : CEVA-XM4 Preprocessor Options

Option	Description
-E	Acivates the C preprocessor only. Processes all the C/C++ source files specified and output the results to standard output or the specified output file. **Note: This option is very important to apply for reporting bugs. The preprocessed output file should be sent to CEVA as an attachment to the bug report.**
-H	Prints the name of each header file used.
-D*macro*	Defines macro *macro* with the number 1 as its definition.
-D*macro=defn*	Defines macro *macro* as *defn*.
-U*macro*	Undefines macro *macro*.
-M [-MG]	Tells the preprocessor to output a rule suitable for make describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the files '**#include'd** in it. This rule may be a single line or may be continued with '\' and newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program. **-MG** instructs to treat missing header files as generated files and assume they live in the same directory as the source file. It must be specified in addition to **-M**.
-MM [-MG]	Similarly **-M** but the output notes only the user header files included with `#include "file"'. System header files included with `#include <file>' are omitted.
-MD	Like **-M** but the dependency information is written to files with names made by replacing `.o' with `.d' at the end of the output file names. This is in addition to compiling the file as specified - **-MD** does not inhibit ordinary compilation the way `-M' does.
-MMD	Like **-MD** except mention only user header files, not system header files.

Option	Description
-I<include-path>	Appends directory <include-path> to the list of directories searched for include files. Single path can be defined with a single -I option. Multiple -I options can be defined in order to append several paths. **Note:** It is possible to define manually an environment variable in the name of **DSPINC** for appending include directories to the list of directories searched for include files. Multiple paths, separated by a semicolon (;), can be inserted into the **DSPINC** environment variable. The Compiler will search for include files in the following order: **1.** Directories indicated by the -I<Dir> compilation options **2.** Directory containing the source file **3.** DSPINC if defined **4.** C include directory (under the tools directory, that is, <TOOLSDIR>\include) **5.** DSP include directory (under the dsplibs directory, that is, <TOOLSDIR>\dsplibs\include)

3.1.6. CEVA-XM4 Code Generation Options

Table 3-6 : CEVA-XM4 Code Generation Options

Option	Description
-mtext-sec-<section definition>	Changes the code section definition to <section definition>. Section definition is as follows: .CSECT,<section_name> .CODE,<section_name> .DSECT,<section_name> .DATA,<section_name> .data .text .bss The default Code section for functions is **.text**. **Example**: -mtext-sec-.CSECT,my_code_section **Note** that a comma (,) is used instead of space in these switches.
-mdata-sec-<section definition>	Change the data section definition to <section definition>. Section definition is as follows: .CSECT,<section_name> .CODE,<section_name> .DSECT,<section_name> .DATA,<section_name> .data .text .bss The default data section for initialized variables is .data. The default data section for variables is .ndata. Example: -mdata-sec-.DSECT,my_data_section Note – see –mtext-sec <section name>.
-mbss-sec-<section definition>	Changes the uninitialized data section definition to <section definition>. Section definition is as follows: .CSECT,<section_name> .CODE,<section_name> .DSECT,<section_name> .DATA,<section_name> .data .text .bss The default is .bss. **Example:**

Option	Description
	-mbss-sec-.DSECT,my_bss_section Note that a comma (,) is used instead of space in this switch.
-mrdata-sec-<section definition>	Change read only data section used for locating constant data variables (default is .**DSECT const_data**) **Note:** See –mtext-sec <section name>.
-ffunction-sections	Each function in source file compiled with this option will be allocated into a separate section. Each function keeps a link to its meta-section (original section name), thus it is not mandatory to explicitly map all the sections in the linking stage. **For example**: C code that contains the function "main" compiled using "-ffunction-sections" will generate a section called "main" with a link to its meta section, that is, ".text$main_section" (origname$funcname_section). At the linking stage, if "main" is explicitly mapped (e.g. main at 0x1234), the Linker will map it as requested, otherwise, "main" will be mapped next to the meta-section ".text".
-fdata-sections	Each data variable in source file compiled with this option will be allocated into a separate section. Each data variable keeps a link to its meta-section (original section name), thus it is not mandatory to explicitly map all the sections in the linking stage (see “-ffunction-sections” example). The name of the section is comprised of the name of the meta-section followed by the $ sign, followed by the variable name followed by “_sect”. **For example:** The variable ‘var’ which belongs to the .data section receives the following section name: **.data$var_sect**
-mjumptable-sections	When this switch is used, the switch-case statement jump tables of each function are assigned to a unique child section of .DSECT const_data. Since the new sections generated are child sections of .DSECT const_data, it is not mandatory to explicitly map them in the linking stage. Without “-mjumptable-sections” switch, the jump tables of all functions are mapped to the read-only data section (default is “.DSECT const_data”). Example: The function “foo” contains switch-case statements for which jump tables are created. Upon compilation with “-mjumptable-sections” switch, a section named “foo_jump_tables_section” will be generated with a link to its meta-section: “.DSECT const_data$ foo_jump_tables_section”. All of the jump tables created for foo’s switch-case statements will be assigned to this new section. At the linking stage, if “foo_jump_tables_section” is explicitly mapped (e.g. foo_jump_tables_section at 0x1234), the linker will map it as requested. Otherwise, “foo_jump_tables_section” will be mapped next to the meta-section “.DSECT const_data”.

Option	Description
	Note: Do not use "-mjumptable-sections" switch together with "-mrdata-sec" switch (i.e., do not change the read-only data section's name).
-mrebuild-switch-file	Deletes the existing '.switch' file and builds a new one. For more details refer to the 'Switch File Usage' section.
-muse-existing-switch-file	Compiles the source using the existing '.switch' file (instead of creating a new one) even when a new one should be created (according to changes in source/time stamps, compiler or other tools, compilation options etc.). This reduces compilation time significantly but does not necessarily generate optimal code and is hazardous because Compiler behavior is undefined if new functions are added to the source. For more details refer to the 'Switch File Usage' section.
-moptimizer-file-<object_name>.o.swt	This switch enables the user to apply function-specific options to the different functions in the built object (see CEVA-XM4 Function-Specific Optimizations and Options). The specified .swt file, which resides in the configuration directory, includes details of attempted option combinations and can specify options for each function separately.
-INLINE:=[on|off]	Forcibly turn on or off stand-alone inline processing; When both are seen in the command line "=off" is processed and "=on" is overridden with a warning. This switch can be used to turn on/off inlining in all optimization levels regardless of the defaults for that optimization level.
-INLINE:none	At call sites, do not attempt to inline routines not specified with must option. If "all" has been specified, and "none" is then specified, "none" is ignored with a warning.
-INLINE:all	At call sites, attempt to inline all routines not specified with never option. If "none" has been specified, and "all" is then specified, "all" is ignored with a warning.
-INLINE:must= routine_name<,routine_name>	Attempt to inline the associated routines at call sites, but do not inline if varargs or similar complications prevent it.
-INLINE:never= routine_name<,routine_name>	Do not inline the associated routines at call sites.
-INLINE:static=[on|off]	Enable inlining of static functions (default off)
-fshort-enums	This switch instructs the compiler to use the smallest fitting integer to hold enums. By default enums are stored as type 'int'.

3.1.7. CEVA-XM4 Optimization Options

The table below describes the most important optimization options that the C Cross Compilers supports. There are general optimization options (-O flag with level number) and specific machine dependent options that can be enabled or disabled (-f… or –fno… flags). Those mentioned in the table are those that are not the default.

Table 3-7 : CEVA-XM4 Optimizations Options

Option	Description
-O0	(*) Does not optimize. This is the default mode when -g is used, so that debugging information is not distorted and is always fully reliable.
-O1	(*) Optimizes the code by reducing the code size and execution time. Compilation takes longer and consumes more memory Resources. The Compiler tries to reduce the cost of compilation and make debugging produce the expected results by keeping the C statements independent. The CEVA-XM4 Compiler schedules instructions in order to parallel them.
-O2	(*) Optimizes even more without involving space-speed trade-offs. Compared to -O1 above, both the compilation time and the performance of the generated code are increased. CEVA-XM4 Compiler tries to unroll loops, which may result in larger code size, and implements heuristics procedures in order to achieve better parallelism.
-O3	(*) Optimizes yet more, performs better register allocation Optimizations. Compilation takes longer to complete. CEVA-XM4 Compiler aggressively performs loops unrolling, which may results in even larger code size. This is the default mode for the CEVA-XM4 Compiler, unless the switch is specified in the command line if -g is not used.
-O4	Performs multiple compilations with different switches in the background in order to find the best combination of switches per function. This optimization requires more time and memory.
-Os0	Does not optimize; code size optimization.
-Os1	Limited loop unrolling.
-Os2 (-Os)	Uses the same optimizations as the -Os1 optimization level with the addition of the following optimizations: Disables unrolling of loops with large bodies, Limits parallelism (to use single 16 bits instructions instead of paralleling them into a single 64 bit packet), The switch -mprolog-epilog-func-min-<number> is used and set with a value of 4.
-Os3	Uses the same optimizations as the -Os2 optimization level with the addition of the following optimizations: Disables unrolling of loops completely, even less parallelism.
-Os4	Performs multiple compilations with different switches in the background in order to find the best combination of switches per function. This optimization requires more time and memory.

(*) In general, the more advanced optimization level applied, the less debugging capabilities are available. Debugging an application that is compiled with -O1 to –O4 optimization levels (and -g), is problematic. This is due to the aggressive optimization methods applied by the Compiler on the generated code to the extent that it is hard to synchronize properly between source lines and variables to the generated code.

Following are few examples and their symptoms:

- Some code lines might be optimized out. In the Debugger level they are skipped when source stepping is performed on the line before them.
- Optimization involves instructions scheduling; meaning some of the executable code generated may be in reverse order compared to the source lines order. The above are reflected in the Debugger level when source stepping is jumpy and seems to not follow the logical source flow.
- Automatic (local) Variables might be eliminated. In the Debugger level, these variables cannot be displayed in the Watch/Locals windows.
- Performing consecutive operations on variables that are located in data memory (as opposed to registers) do not take effect after each operation but only after all the operations are completed.

  In the Debugger level, such variable is observed as updated only after all the operations are completed.

  e.g:
  a++ // ➔ move the variable into a register and perform the increment
  a++ // ➔ perform an additional increment and move back to memory
  In the example above, variable a's value are not updated in the Watch window until ending the second increment since only then it is written back to the memory.

3.1.8. CEVA-XM4 Specific Optimizations

CEVA-XM4 enables setting optimizations and options at file resolution as well as at function-specific resolution, enabling the user to determine different compilation settings for different functions within the same file. As a result, projects can now be optimized for overall performance using a 'mix' of different compilation and optimization options, as opposed to optimization specifically oriented towards either code-size OR cycle-count. This feature provides the foundation for the Build Optimizer IDE feature (see IDE User Guide, Build Optimizer chapter), but can also be very useful in standard compiler use. Through use of a .swt file, which resides in the configuration directory (%CONFIG%), different options can be specified for each function in the built object. The compiler options which can be applied at the function-specific level are detailed in the table below:

Table 3-8 : CEVA-XM4 Specific Optimizations

Option	Effect
-fno-pointers-optimization	The Compiler tries to use a single pointer (with post modifications) in order to access sequential locations in an array (only inside loops). Pointer optimization is enabled by default from -O2 (optimization level) and higher, and can be **disabled** by using **-fno-pointers-optimization** command line option.
-OPT:unroll_times_max = <number>	When a loop is not fully unrolled, limit the unrolling factor to <number>. Defaults are: 4 when not using -Os switch 1 when using -Os switch
-CG:unroll_fully= <on|off>	Enable full unrolling of loops. Defaults are: 'on' when not using -Os switch 'off' when using -Os switch
-mprolog-epilog-func-min-<number>	Specifies the minimal number of push/pop instructions that should trigger calls to the store/restore of functions that store/load all call saved registers. Defaults are: 4 when using -Os switch "infinite" number when not using the -Os switch.
-IPFEC:force_if_conv =<on|off>	Forces "if-conversion" optimization (generation of predicated code regardless of the Compiler estimated gain).
-OPT:alias=typed	The Compiler assumes by default that pointers to different basic types are NOT pointing to aliased objects (also ANSI defaults). This switch is 'on' by default. -OPT:alias=no_typed option could be used to force a strict approach. Note that relaxing memory aliasing by using this switch, can boost the performance as it allows the Compiler to improve its' scheduling capabilities.
-OPT:alias=no_typed	The Compiler assumes by default that pointers to different basic types are NOT pointing to aliased objects (also ANSI defaults). This switch is to be used in order to force a strict approach by the Compiler. Note that relaxing memory aliasing by using this switch, can boost the performance as it allows the Compiler to improve its' scheduling capabilities.

Option	Effect
-OPT:alias=strongly_typed	The Compiler assumes that pointers to different types (basic types and non-basic types) are NOT pointing to aliased objects. Note that relaxing memory aliasing by using this switch, can boost the performance as it allows the Compiler to improve its' scheduling capabilities.
-OPT:alias=restrict	Compiler assumes that distinct pointer variables are NOT pointing to aliased objects. Note that relaxing memory aliasing by using this switch, can boost the performance as it allows the Compiler to improve its' scheduling capabilities.
-mno-speculative-memop	Avoid generating load operations that might read from uninitialized data.
-CG\:gcm_aggressive_ds=[on|off]	Performs aggressive delay slots filling of branch-type instructions (such as brr, call, ret, etc.), and is enabled by default (=on). This optimization may increase compilation time for very large functions, and could be disabled by applying -CG:gcm_aggressive_ds=off (note that moderate delay slots filling mechanisms are still enabled in such case, therefore limiting the impact over performance).
-CG: restrict_loop_swp =[on|off]	Takes the number of available registers into consideration during loop Software-Pipelining. When used, it may reduce register pressure, while limiting loop rotation. This option is disabled by default (=off).
-mstack-mem-block-[0-3]	Notifies the Compiler regarding which memory block the stack section is mapped to. When used, load/store operations to/from the stack are marked as using the given block, and therefore will not be scheduled in parallel with other memory operations from the same block, in order to avoid block conflicts.

3.2 CEVA-XM4 Core Specific Features

This chapter includes the following **CEVA DSP Cores** specific features:

- **Data Types**
- **C/C++ Language Environment Assumptions**

- **Register Types and Register Usage**
- **CEVA-Toolbox Libraries**
- **Floating Point Library Emulation Functions**
- **Arithmetic,16/32 Bits Integer, Emulation Library**
- **Switch File Usage**
- **Multiple Section Allocation Support (using malloc)**

3.2.1. Data Types

The C language has four basic data types: **char**, **int**, **float** and **double**.

The **char** (character) type is 8 bits wide in CEVA-XM4. The **int** type is used for integers; in CEVA-XM4 **int** is defined as a 32 bits type. The **float** and **double** types express real (floating point) values. In the CEVA-XM4 Compiler, the IEEE-754 Single Precision Floating Point standard is used for representing both **float** and **double** types in 32 bits (i.e. there is no difference between the two).

Using two out of the four "type qualifiers" of the C language, **short**, **long**, **signed** and **unsigned**, one can decrease the size of an integer to 16 bits (short), leave the size of an integer as 32 bits (long) or extend the range of non-negative integer values (unsigned).

The **long long** data type can be used in order to represent a 64-bit wide integer.

3.2.1.1 CEVA-XM4 Vector Processing Unit (VPU) Data Types

In addition to the data types detailed above, there are 4 additional data types unique to the CEVA-XM4 VPU, used to describe the vector resources of this core.

Native 256-bit vectors	
char32	uchar32
short16	ushort16
int8	uint8
float8	
128-bit vectors	
char16	uchar16
short8	ushort8
512-bit vectors	
short32	ushort32
int16	uint16

3.2.2. C/C++ Language Environment Assumptions

As listed below, the C/C++ Compiler generated code and the C/C++ libraries make some assumptions about the DSP target core environment.

1. Users must assume worst case latencies in connection points between C and assembly code. For example, in the delay slots of ret or call instructions of an assembly function that return/call to a C function.
2. Saturation mechanisms assumed to be disabled.

3. Block Repeat nesting-level behavior:

   At all times (except in interrupts) upon function entries only, the Compiler assumes that current bkrep nesting level does not exceed hardware nesting level of two. As a result, a function is allowed to have up to two additional hardware nested loops (the CEVA-XM4 provides four hardware nested bkrep loops) without generating the *bkst/bkrest* instructions. If a function generates more than two hardware bkreps, then the Compiler starts generating *bkst/bkrest* instructions for each additional nesting level.

   For example, when the Compiler generates four nested bkrep loops, there will be two software nested loops: two *bkst* instructions at function entry and two *bkrest* at function exit. This is in addition to the two hardware nested loops. When a loop involves a function call, *bkst* is issued for all succeeding bkrep nesting levels with maximum of two bkrep loops that are generated around the function call (this way, the called function can still maintain the assumption that bkrep nesting level has not exceeded two). Since interrupts may be triggered while the bkrep nesting level is four, the generated code for interrupt routines will work under the assumption that there is no bkrep nesting level available.

   For example, a function call from an Interrupt Service Routine (ISR) having a single bkrep loop will be surrounded by 3 *bkst/bkrest* (two for the called function and one for the ISR bkrep).

4. The C/C++ Compiler assumes that _dsp_asm statements embedded in C code contain at least one instruction each.

5. The CEVA-XM4 C/C++ Compiler assumes that all sections are aligned to 4. The user must take this into consideration when manually mapping sections (see Linker Script File section in the Linker Script File chapter of the Assembler & Linker User Guide), in order to avoid issues caused by incorrect alignment given by the user (such as erroneous generated code under the assumption that the user can align sections to any size).

3.2.3. Register Types and Usage

The C/C++ Cross Compiler divides the CEVA DSPs registers into three major types:

A. Fixed registers that are not accessible to the internal register allocation.

B. Call used (scratch) registers that need not be saved across function calls.

C. Call saved registers, which should be saved across function calls.

Note: *Refer to Mixing Assembly and C Routines chapter and the relevant sub sections:*

⇒ *CEVA-XM4 Register Usage Convention*

A. Call-Used Registers

Call-Used registers are registers of the **CEVA DSP Cores** that the C/C++ Compiler uses in the generated assembly code as scratch registers, and does not expect them to retain their original values across a function call. As a result, the called function – whether it is an assembly routine or a C function – does not need to save/restore the original value of those registers. When calling an assembly routine from C, it is recommended to use as many "call-used" registers in the assembly routine as possible. This strategy will help reduce the number of save/restore operations in the assembly routine. However, note that if your assembly routine calls a C function, those registers might be changed by the C function.

B. Call-Saved Registers

Call-saved registers are also registers of the **CEVA DSP Cores** that the C/C++ Compiler uses in the generated assembly code. As opposed to call-used registers, the Compiler assumes that these registers retain their values across function calls. Any called function must, as a result, save on entry and restore on exit the original value of any of those registers used in the called function. The only exception to this rule is when specifying a call-saved register in the user defined calling convention. The Compiler-generated code will handle the storage/restoration of the call-saved registers that appear in the user defined parameter list.

Note: Registers that are **Call-saved** and are also **fixed** cannot be added to the parameter

list and must be handled specifically by the called assembly routine. Any called function must, as a result, save and restore the original value of those registers if they are used in the called function.

C. Fixed Registers

Fixed registers are those registers that are never allocated by the C/C++ Compiler's register allocation phase, but might be used in the generated assembly code. i.e., it is possible that some registers, or certain bits within registers are fixed, but at the same time call-used or call-saved.

C.1 Registers that are both Fixed and Call-saved

Certain registers, or bits within registers, are assumed by the C/C++ Compiler to retain certain pre-specified values throughout the application. These registers/bits control the operating environment of the DSP (Refer to the *C/C++ Language Environment Assumptions* for more information).

If a called assembly routine or an in-line assembly instruction manually changes these environment bits/registers, the ensuing C generated assembly code will most likely be corrupted. It is the programmer's responsibility to restore any changes of the environment in the assembly routine.

C.2 Registers that are both Fixed and Call-used

Registers that are **fixed** and also **call-used** are those registers/bits that are **not** used by the Compiler in the generated assembly code, but whose values may be affected by other register values during a function call. The Compiler assumes that these register/bits values can be changed across function calls, and does not rely on the programmer to restore their values. As such, the programmer is free to clobber these registers/bits in any called assembly routine.

The tables in the following chapters specify the register usage by the C/C++ Compiler for the various cores. Note that any registers listed under **call saved** are explicitly saved and restored by the Compiler only if used within a function. If the user wants to link C/C++ Compiler generated object files with Assembler generated object files, he needs to be aware of the Compiler's register usage to correctly save **"live"** registers or freely (efficiently) use (overwrite) **"dead"** registers.

Practically, this means that if C/C++ program calls a function manually written in assembly, it is the responsibility of the assembly routine to save and restore all those registers used by the routine body and marked as **call saved** in the SmartCore relevant tables in the Mixing Assembly and C routines.

Warnings: ***Important:*** *Certain optimizations may cause the Compiler to allocate different variables to different dedicated registers although these variables are not concurrently "**alive**" (within the same function). This may result in unpredictable values handled by the Debugger and assigned to these variables. Therefore, when using the Debugger (i.e. invoking the Compiler with **-g** debugging mode option), no optimizations by default are performed. It is recommended not to activate any explicit optimizations when using the **-g** option, unless the drawbacks (some of the displayed data is unreliable) are taken into account.*

3.2.4. CEVA-Toolbox Libraries

The CEVA-Toolbox C/C++ Cross Compilers have two sets of emulation functions. The first set consists of functions for floating point arithmetic operations (including conversion routines) and the second set consists of functions for signed and unsigned 16 and 32 bits integer arithmetic operations (which are not directly supported by simple DSP instructions). Both are included in the supplied XM4LIB.LIB library.

Notes: *Libraries are not linked unless necessary, that is, libraries are used only when there is a reference that cannot be resolved from the user objects. In that case, the Linker links only the required object file from the corresponding library that is needed for successful linking. When the Linker decides to include a library's object, the object is included as a whole with all its sections/functions (other unneeded objects from the library are not included). Since the Compiler's libraries include one function per object file, it ensures that the minimal necessary code is inserted. For that reason, it is advised to create user libraries in the same manner.*

3.2.5. Floating Point Library Emulation Functions

Table 3-9 :Floating Point Library Emulation Functions

Function name	Description	Inputs	Outputs
_fldiv	Floating Point **division**	r0, r1 (r0 / r1)	r0

3.2.6. Arithmetic,16/32 Bits Integer, Emulation Library

Table 3-10 :Arithmetic,16/32 Bit Integer, Emulation Library Functions

Function Name	Description	Inputs	Outputs
_mulsi3	signed 32 bits by signed 32 bits multiplication	r0, r1	a0
_umulsi3	unsigned 32 bits by unsigned 32 bits multiplication	r0,r1	r0
_divsi3	signed 32 bits by signed 32 bits division	r0,r1 (r0/r1)	r1
_undivsi3	unsigned 32 bits by unsigned 32 bits division	r0, r1 (r0 / r1)	r1
_modsi3	signed 32 bits by signed 32 bits modulus	r0, r1 (r0 % r1)	r0
_unmodsi3	unsigned 32 bits by unsigned 32 bits modulus	r0, r1 (r0 % r1)	r0
_divhi3	signed 32 bits by signed 32 bits division	r0, r1 (r0 / r1)	r1
_udivhi3	unsigned 32 bits by unsigned 32 bits division	r0, r1 (r0 / r1)	r1
_modhi3	signed 32 bits by signed 32 bits modulus	r0, r1 (r0 % r1)	r0
_umodhi3	unsigned 32 bits by unsigned 32 bits modulus	r0, r1 (r0 % r1)	r0

3.2.7. Switch File Usage

When using the -O4/ -Os4 optimization levels, the Compiler internally attempts various switches (methods) in order to find the best combination for each function. All compilations of -O4/-Os4 generate a file with the same name as the compiled file with the suffix '.switch'. This file contains data saved and used by the Compiler about the optimal switches state according to the Compiler's internal scoring function. (editing these file is possible since they are text files, but as the format is very strict it is not recommended.)

Once the Compiler generates the '.switch' file it will keep using that switch file until the source is changed (time stamp comparison, as done in makefiles) and therefore, in follow up compilations, the Compiler will run only the selected best method (the best combination) for each function. The user can also force the Compiler to regenerate or not regenerate the switch file by using certain command line options (For more details refer to *-mrebuild-switch-file* and *-muse-existing-switch-file* in the CEVA-XM4 Code Generation Options section).

3.2.8. SWT File Usage

The IDE enables users to specify compilation options per different functions in the same source file.

When used, the IDE will generate an XML file with the same name as the built object file and the suffix '.swt', which lists all the functions in the source file and their chosen compilation options.

The compiler will then use the settings configured in the '.swt' file in order to compile each function with a separate set of options (for more details refer to *-moptimizer-file* in the CEVA-XM4 Code Generation Options section).

For a list of compilation options that may be applied per function, refer to the CEVA-XM4 Function-Specific Optimizations and Options section.

3.2.9. Multiple Section Allocation Support (using malloc)

The Compiler supports allocation of memory in multiple sections using the standart allocation functions (malloc, calloc, ...). The allocation section can be switched in run-time by calling the new function - init_malloc(). This function is called in the default startup code (crt0.c) in order to initialize the default allocation section (__MALLOC_SECT).

Usage:

To switch between malloc various sections, the user has to use the init_malloc() function, and all other operations (allocation, reallocation, etc') are done using the well-known allocation functions (malloc, calloc, etc').

For more details refer to “init_malloc” function chapter.

3.3 Default Calling Convention

The C default function calling convention defines the conventions for parameters passing to a function, getting a return value from a function, and the register usage strategy of each function.

This chapter includes the following sections:

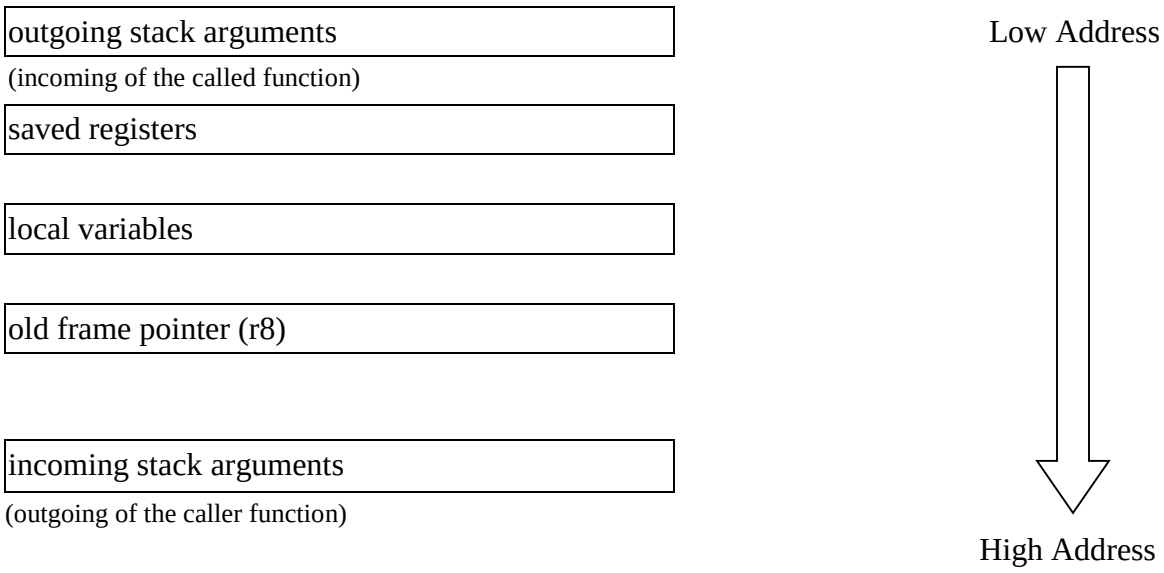
- **Function frame Structure**
- **Stack and Automatic Variables**
- **Argument Passing**
- **Function Return Value**

3.3.1. Function Frame Structure

In general, the C Compiler uses the following frame structure on the software stack for every function that is called. The Debugger locates the values of all the local (automatic) variables and the function's arguments using this structure.

The C runtime stack grows from high to low addresses (reading the arguments left to right). The **sp** register is used as stack pointer. The **r8** register is used as frame pointer. The stack pointer is initialized in the Compiler generated file **crt0.c** (Refer to the chapter *crt0.c - Startup Code* for details on changing the stack's location and size).

Stack address space is described in the following diagram:



3.3.2. Stack and Automatic Variables

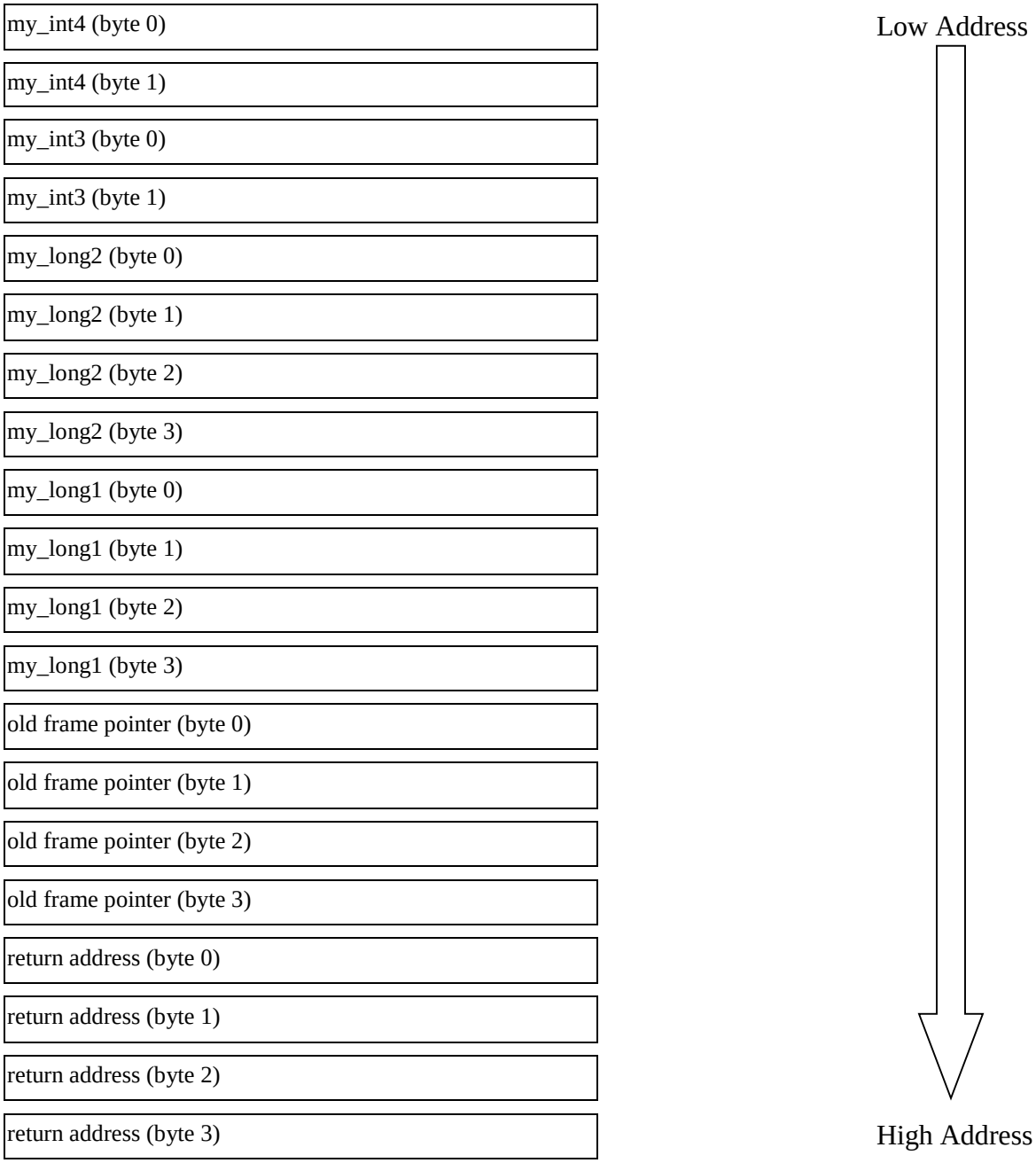
Local variables are sequentially stored in the stack frame from high to low in the order they appear in the source code.

Example:

```
short MyFunction( void )
{
    long my_long1, my_long2 ;
    short my_int3, my_int4 ;
}
```

At run time, assuming all local variables are not allocated to registers, or when the Compiler is invoked without any optimizations (i.e. with **-O0** or with **-g**), the stack frame for this function appears as follows (Little endian, byte addressable):
(the example uses the short data type since int has the same size as long)

- Byte 0 represents the LSB (bits 0-7)
- The return address will be located in the beginning (highest address) of the function's stack frame if the Compiler is invoked with -g3 (as shown in the illustration below). Otherwise it will be stored *after* the local variables, and only if the current function contains a function call



3.3.3. Argument Passing

The C Compiler passes arguments in registers and/or in the runtime stack (reading them from left to right). For functions with fixed number of arguments, the Compiler tries first to allocate the accumulators or pointer registers for passing them according to the prototype of the called function:

- Integer, long, short or char variables are passed through r0-r7

If there are not enough registers to pass all parameters, the Compiler allocates the stack for the remaining arguments. Structures are passed on the stack, and the remaining arguments are allocated to the accumulators/pointers if possible, as explained above.

Arguments of type long long (64-bits integer) are passed in two consecutive registers, the low 32-bits in the first register (rN) and the high 32-bits in the second (rN+1).

The Vector Processing Unit (VPU) types are passed in vector registers v0-v7.

For functions with a variable number of arguments, all arguments are passed on the stack not using the accumulators/pointers.

Stack pointer adjustment after arguments have been passed on the stack:

It is the caller's responsibility to free the stack space used for outgoing arguments before returning.When a short-type argument (16-bit) is passed on the stack, one padding word is used.

Warning: *Functions with variable number of arguments (varargs) must be declared with proper prototype, otherwise an argument mismatch between caller and callee may occur. In addition, these functions pass the arguments from right to left (as exception to the rule from left to right).*

Example 1:

```
long a,b;
int c,d;
main()
{
    foo(a,b);
    foo1(c,d);
}
```

Variable a is passed in r0 accumulator and variable b is passed in r1 accumulator

Variable c is passed in r0 accumulator and variable d is passed in the r1 accumulator.

Example 2:

```
long a,b;
int c,d;
float e;
long f;

main()
{
    foo (a,c,b,d);
    fft (a,b,c,d,e,f);
}
```

In the call to function foo:

a is passed in **r0, c** in **r1, b** is passed in **r2, d** in **r3.**

In the call to function fft:

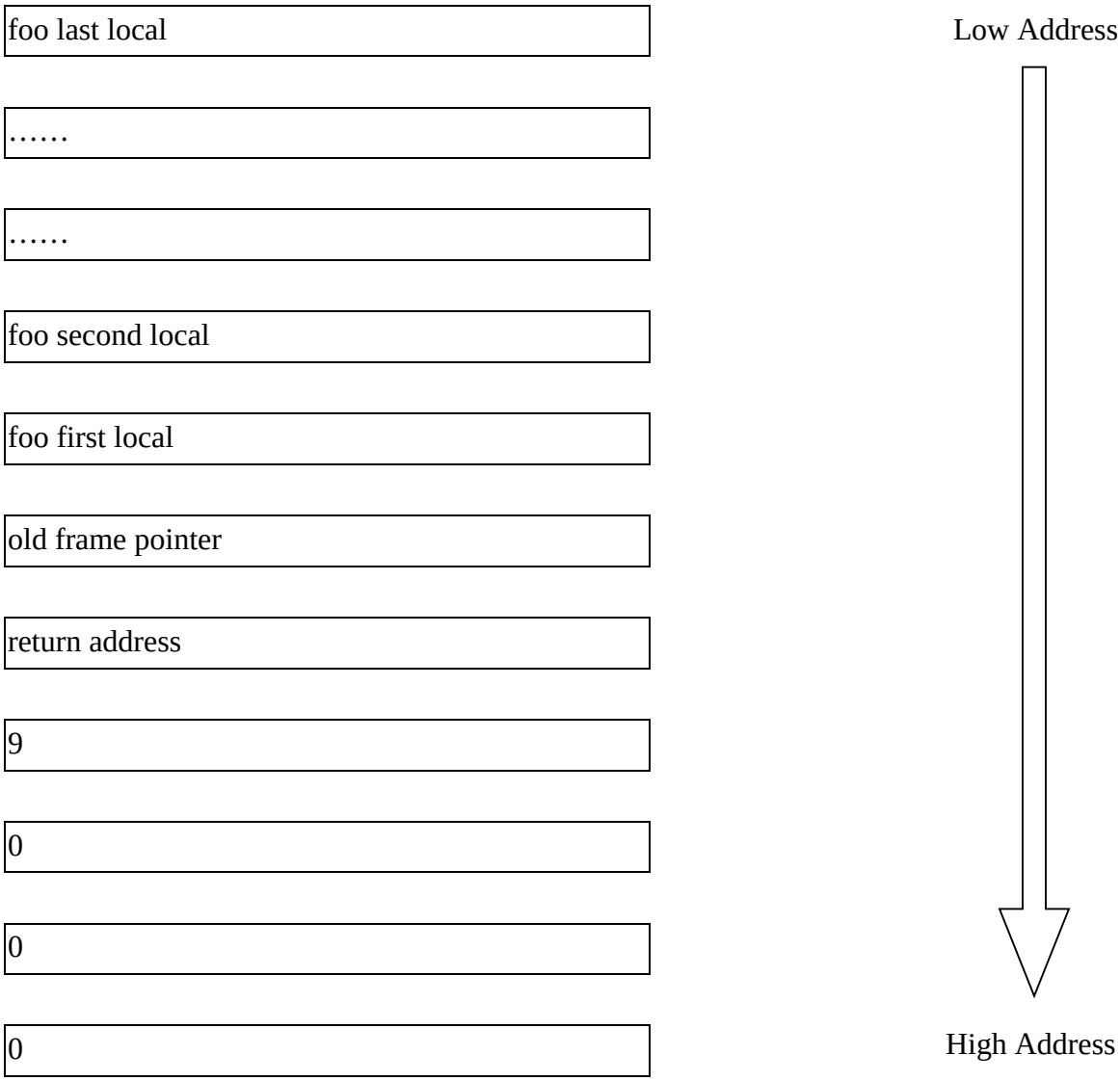
a is passed in **r0, b** in **r1, c** is passed in **r2, d** in **r3, e** in **r4, f** in **r5.**

For the CEVA-XM4 (Little endian): no parameter is passed on the stack

For example:

```
Void goo()
{
        foo(1,2,3,4,5,6,7,8,9)
}
```

The function passes immediate arguments 1 to 8 in r0 to r7, while pushing to the stack a double word (size of int) of the value 9. Since the CEVA-XM4 architectures is little endian, the upper part of the frame would contain:



Example 3:

```
struct longint
{
    long a;
    int b;
} ab;

int c;
main()
{
  foo (ab,c);
}
```

In the call to function foo, **ab** is passed on the stack and **c** is passed in the **r0** register according to the explanation above.

3.3.4. Function Return Value

Values returned by functions are by default always returned in the **r0** register. Exceptions to the above rule are the following:

(1) Return of Vector Processing Unit (VPU) vector types:

- vectors are returned via the first vector register v0.

4. C/C++ Language Extensions

Even though DSP cores have been created to resolve special applications in the world of signal processing, C/C++ language has not yet been designed to support their unique architectures; the paradox being that the requirement to use High Level language for DSP applications resulted in creating C/C++ Language Extensions enabling the user to write DSP applications combining the power of C/C++ and **CEVA DSP Cores** architecture.

This chapter includes the following sections:

- **In-Line __asm__() Extension**
- **Function and Variable "Section" Attribute**
- **Function "Interrupt"Attribute**
- **Prologue/Epilogue Function Attribute**
- **Assembly Implemented Interrupt Function**
- **Memory Block Attribute**
- **Aligne Attribute**
- **Explicit Register Variables**
- **Vec-C Intrinsics**

4.1 In-Line __asm__() Extension

The **__asm__(...)** statement extension allows the embedding of one or more assembly instructions within any C/C++ sequence. The C/C++ Cross Compiler allows to embed **in-line** assembly instructions inside **__asm__ (...)** statements without constraints on context. In order to use that macro users should include <asm-dsp.h>.

It is important to note however, that the Compiler is unaware of the context of such statements. The Compiler simply injects the instruction(s) right in between the preceding and next C statements of the **__asm__** block. It is the user's responsibility to verify correctness and syntax of the assembly code and to assure that embedded instructions do not overwrite **"live"** (in use) registers, i.e. those holding values set by the Compiler's preceding C statements to be used by C statements later.

Example 1:

```
__asm__ ( "add r0.i, r1.i, r2.i" );
```

In this example, the user specifies everything and the Compiler is "unaware" of the instruction. Had the Compiler allocated **r2** to hold data as a result of an earlier statement compilation, the above assembly instruction would overwrite **r2** contents. Due to this possibility, the use of this extension should be implemented with much care.

A second drawback to this method is the lack of a Compiler syntax check for an assembly instruction surrounded by quotation marks.

Example 2:

```
_dsp_asm( " add r0.i, r1.i, r2.i \n mov #0x0200, r0.i" );
```

This example shows how one can generate multiple assembly instructions per _dsp_asm() statement. The same limitations as in the first example apply here. If **r2** or **r0** are already set to hold valid data (generated earlier by the Compiler), the program might fail to execute correctly due to the embedded instructions "destroying" these registers contents.

Example 3:

```
register long j1 __asm__("r0") = h ;     assign j1 to r0
register short j2 __asm__("r1") = h ;    assign j2 to r1
```

Note that syntax constraints require certain instructions to be followed by other instructions, or similarly, certain instructions cannot be followed by specific instructions. Considering these limitations is within the responsibility of the in-line assembly programmer.

Notes:

1. *When referencing C/C++ names (variables or function names) from inlined assembly instructions (or assembly files), one should note the underline prefix ('_') added to C names in the Compiler's generated code. e.g:*
   int cvar;
   _dsp_asm("mov ##_cvar, r0.ui");
2. *When referencing C++ functions from inlined assembly instructions (or assembly files), one should note the suffix added to the original function name in the Compiler's generated code. In order to enable functions overloading that differ only in it's input arguments, C++ codes the input arguments in the resultant assembly function name. This addition is added as a suffix to the original C++ level function name. In order to figure out what are the assembly level names of C++ functions, it is best to observe the Compiler's generated assembly code by activating the Compiler with the –S command line option (stops before running the Assembler).*
3. *For embedding inline assembly in C code, either _dsp_asm or __asm_ can be used (both provide the same functionality). For variable binding to hardware registers, __asm__ must be used.*

4.2 Function and Variable "Section" Attribute

Normally, by default, the Compiler places the code generated in the **.text** (aliases to **.CSECT**) section. However, due to application requirements, a need for additional sections may arise such as the placement of particular functions in specific sections. Section attribute should be appended to the function's declaration line (not to the function's definition line) and specifies that a function **"lives"** in a specific section. The following example demonstrates the syntax required for the function section attribute:

```
void foobar (void) __attribute__ ((section (".CSECT mmm")));
void foobar (void)
{...}
```

In the above code example, the Compiler places the function foobar in the **.CSECT** mmm section. **Note** that the specification of a section name is done in the external declaration of the function.

Multiple attributes can be specified in a declaration by separating them by commas within the double parentheses:

```
extern int dataobj __attribute__ ((section (".DSECT ooo"), inpage));
```

Or, by immediately following an attribute declaration with another attribute declaration or other language extension. For example, to associate the assembly label *yoyo* to an integer C variable *var1* and allocate it to a section named *MyCode*, use:

```
int var1 __asm__("yoyo") __attribute__ ((section (".DSECT MyCode")));
```

4.3 Function "Interrupt" Attribute

Interrupt Handlers can be declared by C statements. This is done in a way similar to other function attributes declarations being appended to function declaration line. The difference is the placement of the **interrupt** keyword within the attribute declaration. For example,

```
void int0 (void) __attribute__ ((interrupt));
```

defines function *int0* to be compiled as an interrupt handler for *software* or *maskable* interrupts. The code generated saves all used registers upon interrupt handler invocation and recover them by applying the **reti** instruction (compared to **ret** normally used in a regular function return). **Note** that such a function (defined as an interrupt handler) should not be called from another function.

In order to implement an interrupt handler for *non-maskable* interrupts, the following attribute should be used:

```
void nmi_int0 (void) __attribute__ ((interrupt_nmi));
```

Interrupt handler routines defined by the interrupt_nmi attribute use the **retn** instruction rather than **reti**.

Setup of the interrupt vectors is still under the responsibility of the application programmer. Refer to "crt0.c - Startup Code" section of the crt0.c description, for details.

4.4 Assembly Implemented Interrupt Function

In addition to the requirements detailed in Function "Interrupt" Attribute topic, an assembly interrupt function should be implemented as follows. In order to save all used registers upon interrupt handler invocation and recover them when interrupt ends, add the following functionalities:

At the beginning of an interrupt handler function (for a software or maskable interrupt):

- Push retreg into the stack
- Call the library function **_dsp_ceva_call_saved_store_interrupt** , using **call** instruction.
  This routine saves all used registers.

At the end of an interrupt handler function:

- Branch to the library function **_dsp_ceva_call_saved_restore_interrupt** using **branch** instruction.

  This routine restores all used registers, pops the retreg, and returns from the interrupt back to the interrupted function by **reti** instruction.

For a non-maskable interrupt handler function, use the following store/restore functions: **_dsp_ceva_call_saved_store_nmi** and **_dsp_ceva_call_saved_restore_nmi**.

For example:

```
;; interrupt function start
LS0.push  retreg.ui
||   PC.call {t} INCODE __dsp_ceva_call_saved_store_interrupt
;; interrupt function body…
PCU.br {ds4,t} INCODE __dsp_ceva_call_saved_restore_interrupt
Some instruction1
Some instruction2
Some instruction3
Some instruction4
;; interrupt function end
```

Note: the delay slots in the example are optional.
Note: The implementation of _dsp_ceva_call_saved_store_interrupt can be found at the compiler libraries directory (%TOOLS%\libs\) in the file: crt0_user\crt0_interrupt_prologue.c.

The implementation of _dsp_ceva_call_saved_restore_interrupt can be found at the compiler libraries directory (%TOOLS%\libs\) in the file: crt0_user\crt0_interrupt_epilogue.asm.

In addition, an empty file named crt0_user\crt0_interrupt_epilogue.c can be found in the same location for compatibility purpose.

4.5 Prologue/Epilogue Function Attribute

Syntax:

<ret_val> func_name (args) __attribute__((prologue("func1"),epilogue("func2")));

This attribute instructs the Compiler to invoke selective (user-defined) prologue and epilogue routines for storing and restoring resources used by the declared function.
The routines whose names are provided by the attribute will replace the default routines used by the compiler:

- If the declared function is an interrupt handler (defined by the interrupt attributes), the provided routines will replace
  _dsp_ceva_call_saved_<store|restore>_interrupt or
  _dsp_ceva_call_saved_<store|restore>_nmi
- Otherwise, the provided routines will replace the default store/restore routines used in high code size optimization levels for backing up call saved registers

4.6 Memory Block Attribute

Syntax:

<type> *<variable_name> __attribute__ ((mem_block (N)));

Where N is the block number.

This attribute assists the programmer in avoiding bank conflicts.
A bank conflict occurs when both Load/Store units (LS0 and LS1) try to access the same bank in the same memory block in the same instruction packet.
The attribute allows the programmer to assign a C variable pointer to a specific memory block, thus avoiding memory accesses of two pointers to the same block in parallel.

Examples:

```
int *src1 __attribute__ ((mem_block (0)));
int *src2 __attribute__ ((mem_block (0)));
```

This indicates that the two pointers point to memory block 0.
In this case, the compiler will not perform memory accesses to the above two pointers in

parallel.

```
int *src1 __attribute__ ((mem_block (0)));
int *src2 __attribute__ ((mem_block (1)));
```

This indicates that the two pointers point to different memory blocks.
In this case, the compiler assumes that it is safe to access the memory pointed to by the above two pointers in parallel.

Important note:
By default the compiler assumes that pointers point to different memory blocks.

4.7 Aligned Attribute

A memory block is aligned N if its base address is a multiple of N. The Compiler's conventions for data alignment depend on the architecture and are driven by its spec demands. The aligned attribute enables communication with the Compiler regarding alignments of program's data.

In the CEVA-XM4 Compiler, the default alignment for all data variables is 4 bytes, except for integral types, which are aligned according to the type width (char = 1 byte, short = 2 bytes, int = 4 bytes). Declaring the alignment (in bytes) of a memory block pointed by a function argument:

Syntax:

<type> *<argument_name> __attribute__ ((aligned (N)))

When a function receives a pointer to a memory block as an argument, this block is assumed to be aligned to the size of the pointed data type. If this block is guaranteed to have a wider alignment, than using the aligned attribute on the pointer argument changes the assumed alignment and enables the Compiler to perform code optimizations that are otherwise not allowed.

Example:

```
void foo(short *p __attribute__ ((aligned (4))))
{
}
```

Here the Compiler is informed that although p is a pointer to a 1-word (2 bytes) data type, double-word (4 bytes) accesses may be used.

Forcing alignment (in bytes) of global data or struct/union data members:

Syntax:

<type> <name> __attribute__ ((aligned (N)));

Using the aligned attribute on a global data definition instructs the Compiler to precede the corresponding label in the assembly output with the assembler directive .ALIGN N. Using this attribute on a struct/union data member definition instructs the Compiler to add padding to this struct/union layout so this member's offset from its base address is aligned as specified.

Example:

```
typedef struct
{
        char c;
        short a[10] __attribute__((aligned (4)));
}S;

S s __attribute__((aligned (4)));
```

Here the global s is placed in an address that is a multiple of 4, with the layout:

Words offset	data
0	c
1	-- padding --
2	-- padding --
3	-- padding --
4	a[0]
5	a[1]
6	a[2]
7	a[3]
8	a[4]
9	a[5]
10	a[6]
11	a[7]
12	a[8]
13	a[9]
14	-- padding –
15	-- padding –

Notes:

1. Padding is added to make the size of S also a multiple of 4, in order to maintain this alignment in arrays of such structs.
2. The aligned attribute is not supported on local variables.

4.8 Explicit Register Variables (Register Binding)

The C/C++ Compiler enables linking hardware registers contents to Register Variables. It is possible to define a Register Variable pointing to the contents of a specific hardware register. Register Variables are divided into **Global** Register Variables and **Local** Register Variables.

Global Register Variables Reserve Registers' contents throughout the application (using the **extern** key word in other files than the file where the variable is defined). This may be useful in applications having some global variables that are frequently accessed.

Local Register Variables reserve registers contents in the scope of the function in which they are defined.

These local variables are sometimes convenient for use with the in-line **__asm__** feature, when one wants to assign an Assembler instruction output directly into a specified hardware register.

It is impossible to link a fixed hardware register to a register variable since the Compiler cannot generate instructions using fixed registers. The appropriate way to read/write a fixed register is by inlining assembly instructions.

4.8.1. Defining Global Register Variables

The following syntax is used to define a **global** Register Variable:

```
register int *p __asm__ ("r3");
```

In the above statement, **p** is declared to be a **global** Register Variable type pointing to an integer saved in **r3** register. It is recommended to select a register that is normally saved and restored by function calls, so that library routines do not clobber it.

Linking **global** Register Variables to specific hardware registers reserve these registers and they are not allocated by the Compiler for any other purpose. Therefore these registers are not saved and restored by relevant functions.

Instructions defining these registers are never deleted even if they would appear to be **"dead"** (useless). On the other hand, references to **global** Register Variables can be deleted, moved or modified.

A function that can alter the value of a **global** Register Variable cannot be safely called by a function that does not use this variable. The reason for that is because the called function can clobber the contents of the variable the caller expects to use on return. Therefore, the called function, used as the entry point into the part of the application that uses the **global** register variable, must explicitly save and restore the value belonged to its caller. All **global** Register Variable declarations must precede all function definitions. Declaring **global** Register Variables after function definitions can be too late to prevent the usage of those registers for other purposes in these functions. **Global** Register Variables may not be assigned by initial values because an executable file has no means to supply initial values to registers.

4.8.2. Defining Local Register Variables

A local Register Variable can be defined and linked to a specified register contents like the following:

```
int kuku ()
{
   register int       x   __asm__ ("r0");
   register int *     p   __asm__ ("r1");
   register int8    v0  __asm__ ("v0");
}
```

In the above sequence, **x** is declared to be a **local** Register Variable type pointing to an integer saved in the **r0** accumulator. The selected **"bound"** register must match the register variable type (as shown above), and it is recommended to select a register that is normally saved and restored by function calls, so that library routines used do not clobber it.

Defining such a register variable reserves the register being used by the Compiler for the entire function in which it is defined.

The difference between **local** and **global** Register Variables is that a **local** variable definition is done within a function (similar to the difference between regular local and global C variables).

Using **local** register variables is especially useful in in-line assembly statements, and in functions that utilize a large amount of variables that are used in the same scope or execution flow.

Note that apart from the desired functionality, register binding blocks beneficial Compiler optimizations and so its usage should be reduced to a minimum.

WARNING: *Excessive usage of this capability may leave the Compiler with too little available registers to use in certain functions.*
*In particular, binding **all** registers of a certain register file leaves no registers for compiler generated temporaries, and is therefore not recommanded in low optimization levels (such as O0/O1).*

4.9 Vec-C Intrinsics

4.9.1. Introduction

The CEVA-XM4™ DSP Core contains the Vector Processing Unit (VPU), which can simultaneously perform computations on several inputs stored together in vectors. Vector operations are performed on such vectors of inputs and outputs, enabling concurrent processing. Use of Vec-C intrinsics facilitates the VPU's full potential.

The Vec-C intrinsics are overloaded functions which are mapped to built-in instrinsic functions, each of which is mapped to one single assembly instruction. The Vec-C intrinsic format closely resembles the specification instruction format.

For example:.

The following 'C' code will be mapped after compilation into a single 'vmpy instruction:

```
{ …
  short16 vin0, vin1;
  int16 vres;
  vres = vmpy(vin0,vin1);
…}
```

Additional instruction switches such as 'psh', 'sat', etc. are added as the first operand to the intrinsic function e.g. 'vmpy(sat,…)'.

4.9.2. Vec-C Types

In order to provide access to the specialized registers of the CEVA-XM4, dedicated C types are defined. The following Vec-C types are available for use with Vec-C intrinsics:

Native 256-bit vectors	
char32	uchar32
short16	ushort16
int8	uint8
float8	
128-bit vectors	
char16	uchar16
short8	ushort8
512-bit vectors	
short32	ushort32
int16	uint16

Operations using 128-bit vectors take just as many cycles as operations on 256-bit vectors - The only thing saved by using these vectors is memory. Their primary purpose is allowing truncating a higher precision vector into a lower precision vector where necessary.

512-bit vectors are represented by 2 hardware 256-bit registers, and in some cases may require a pair of operations to support a single ANSI-C operation. Their purposes is saving the result of multiplications where the required result precision is double of the operand precision. Copy back into a 256-bit vector when the added precision is no longer needed.

4.9.3. Include Files

- **vec-c.h –** This file should be included in order to use Vec-C intrinsics. It defines all Vec-C intrinsics, each mapped to a single instruction.

4.9.4. List of VEC-C Intrinsics

For a complete list of Vec-C Intrinsics for CEVA-XM4, please refer to: help\CEVA-XM4\CEVA-XM4_Arch_Spec_Instructions\index.html.
Each ISA is linked to the corresponded intrinsic prototype.

5. Pragma Directives

5.1 Introduction

The Compiler can receive additional information from the programmer in the form of pragma directives. This enables the user to 'hint' to the Compiler that certain optimizations should or shouldn't be performed, or to help with the Compiler heuristic information that the Compiler collects, for example how many times a specific loop iterates.

5.2 Supported Pragmas

#pragma dsp_ceva_unroll =<X>

This hints the compiler that a specific loop should be unrolled X times, in order to achieve optimal performance. If no pragma is used, the Compiler attempts to unroll the loop according to heuristic information calculated during compilation.

Usage:

```
for (i=0; i<n; i++)
#pragma dsp_ceva_unroll =8
{
// loop body
}
```

#pragma dsp_ceva_trip_count=<X>

This indicates that the average number of iterations of a specific loop is X. The Compiler uses this information to better calculate heuristic information required for optimizations.

Usage:

for (i=0; i<n; i++)

#pragma dsp_ceva_trip_count=256

{

// loop body

}

Important note: If the value specified by X is greater than 2^16 (65536), this indicates to the Compiler that a bkrep loop cannot be generated. Loops that perform more than 2^16 iterations, where the number of iterations is not a compile-time constant, must use this pragma to disable bkrep generation.

#pragma dsp_ceva_trip_count_min=<X>

This indicates that the number of iterations of a specific loop is at least X. The Compiler uses this information to remove the "guarding if" of the loop. A "guarding if" is a compare and branch sequence, used to skip over the loop body if the number of iterations to be performed is 0. In the case of loop unrolling the Compiler needs to know that the minimum iterations number is equal or greater than the unrolling factor in order to remove the "guarding if".

Usage:

Example 1:

for (i=0; i<n; i++)

#pragma dsp_ceva_trip_count_min=1

{

// loop body

}

Example 2:

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

#pragma dsp_ceva_trip_count_min=4

#pragma dsp_ceva_unroll =4

{

// loop body

}
```

#pragma dsp_ceva_trip_count_factor=<X>

This indicates that the number of iterations of a specific loop is a factor of X. The Compiler uses this information to remove the "remainder loop" of the loop, the "remainder loop" is used when loop unrolling is done and the loop factor is unknown. The compiler needs to create a second loop for the remainding loop iterations. In the case of loop unrolling the loop factor must be equal to (or a factor of) the unrolling in order to remove the "remainder loop".

Usage:

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

#pragma dsp_ceva_trip_count_factor=4

#pragma dsp_ceva_unroll=4

{

// loop body

}
```

#pragma MUST_ITERATE(min,max,multiple)

This pragma provides the Compiler with loop iteration properties to allow the optimizer to choose the best loop control strategy. It is useful for improving loop efficiency when the loop iteration count (also known as trip count) is unknown at compile-time.
Note that it is *not* required to provide all three properties, for example:

#pragma MUST_ITERATE(1,,) indicates that loop is executed at least once.
#pragma MUST_ITERATE(4,,4) indicates that loop trip count is a multiple of 4, with minimum of 4 iteration (notice that maximum count is not provided).
For more details regarding optimization implications refer to *dsp_ceva_trip_count_min*, *dsp_ceva_trip_count* and *dsp_ceva_trip_count_factor* pragma directives in this section.

Usage:

```
#pragma MUST_ITERATE(4,100,4) // trip count is 4 ≤ N ≤ 100, in multiples of 4
for (i=0; i<n; i++)
{

// loop body

}
```

6. Runtime Support

This chapter provides guidance for writing C/C++ applications running on **CEVA DSP Cores.** In addition to compiling the C/C++ source files, linking the resulting object files (possibly with assembly and/or library objects) it is required to select the "start-up" routine to be linked with the rest of the application's objects. The chapter "**CTR0.C - Startup Code"** describes this start-up code in detail. Standard C library functions are described in the *Run Time Library Functions - Details* chapter and those used should be linked with the rest of the application objects. **"CRTN.C - Constructors Destructors"** chapter includes the supported functions for "C++", and provides syntax details and the parameters needed when using library functions.

This chapter includes the following sections:

- **crt0.c - Startup Code**
- **crtn.c - Constructors and Destructors**
- **Default Linker Script File, Sections and Libraries**
- **I/O Support**
  - **I/O support in Simulation**
  - **Standard C stream I/O implementation**
  - **I/O utility programs**
- **Run Time Library Functions**

6.1 crt0.c - Startup Code

This chapter describes the initial code supplied and executed in all compiled and linked applications. This is provided by a special file, named **crt0.o**, which is located in the Compiler libraries **(%TOOLS%\libs**) directory and a set of user editable function, which are located in the **crt0_user** directory under it. The user can use these defaults functions by simply link the LIBC.LIB (done in default) or edit any function and add the newly created object to the objects list in the linker script file.

This file contains standard initial code. Additionally, if required, there is a possibility to prepare and select different initialization files, named **crt1.c** or **crt2.c,** by using the **-crt1** or **-crt2** command line options respectively that in turn force the usage of **crt1.o** or **crt2.o** on the Compiler instead of **crt0.o** (Refer to the **"General Options"** chapter).

The startup code is composed of the following elements and operations:

- The interrupt handlers table – located in special section (**.CSECT inttbl**) which must be mapped first in the code memory.
- Default empty interrupts (implemented in **crt0_user\crt0_interrupt_functions.c**)
- User hook1 (at start) – this hook is called at the very beginning of the start up code in order to enable the user to make changes before any of the real start up code occurs (implemented in **crt0_user\crt0_hook_post_start.c**).
- Definition of data sections for software stack (**__STACK_SECT**), heap (**__MALLOC_SECT**) and IO ports for low level stdin/stdout simulation (**__INPORT_SECT** and **__OUTPORT_SECT**).
- Initalization of the stack register (sp)
- Initalization of configuration registers – done by calling the initConfigRegs function (implemented in **crt0_user\crt0_init_config_regs.c**). The function initalizes the status and mode registers, page bits, bit shifter and more.
- Initalization of data sections – done by calling the initDataSections function (implemented in **crt0_user\crt0_init_data_sections.c**). By default, the function initalizes only the Compiler defaults data sections (.bss, .data and const_data) and the user must modify it in order to initialize any other specific data sections.

- Initalization of file IO for high level simulation – done by calling the initFileIo which is located in the fileio.lib (dumy version of the function is also located in the libc.lib)
- Initalzation of global objects by calling its constructors (in C++) – done by calling the ctor_sect.0 (generated in compilation time for global objects)
- User hook2 (before main) – this hook is called just before the main function in order to enable the user to make some last minute changes before main and after all initalization were made (implemented in **crt0_user\crt0_hook_pre_main.c**).
- Definiton of ARG_SECT and initalization of program arguments. The section ARG_SECT is used for the purpose of calling the main function with argc and argv arguments. Registers r0 and r1 are loaded with argc and argv values respectively according to the C/C++ program arguments that were set through the Debugger's I/O menu. For further information, refer to the relevant section in CEVA-Toolbox Debugger User's Guide
- Call to main function
- Delete of global objects by calling their destructors (in C++) – done by calling the dtor_sect.0 (generated in compilation time for global objects)
- Call to exit function

The source file **crt0.c** can be found in the same library as the default crt0.o file (The release ***libs*** directory). The user can modify the existing **crt0.c** source file, compile it, and replace the existing crt0.o with the newly created one.

Important:

Care should be taken when mixing user's assembly routines with Compiler generated code. This must be done by insuring the recovering of the cores' specific flags when "control" is given back to the Compiler-generated code (Refer to the C/C++ Language Environment Assumptions for more information). Failure to recover the above bits state to its original values may result in failures in library routines and Compiler-generated code.

By invoking the Linker automatically as part of the compilation process, the Compiler automatically creates the Linker script file, containing crt0.o as the first file in the objects

file list. The Linker should include the code of crt0.c in the executable file.

crt0.c file parameters can be modified in some ways:

- Edit and adjust any of the **crt0_user** sources by modifying their functions using "C" or assembly code, compiling and linking it properly.
- various sections (such as **__STACK_SECT, __MALLOC_SECT, __MMIO_SECT, __MAILBOX_SECT**) can be manipulated in order to initialize their starting address and size. It is not recommended to deviate from sections names **__STACK_SECT, __MALLOC_SECT (**The Debugger automatically maps the addresses of the **__STACK_SECT** as "stack" while loading coff and thus enables detection of stack overflow, and **__MALLOC_SECT** is being referenced by the **malloc** library function).

Note that heap overflows are reported at run time.

6.2 CRTN.C — Constructors and Destructors

For C++ applications, the Compiler uses 2 default sections, **ctor_sect** and **dtor_sect**, for constructor/destructor function calls.

- **crt0.o** calls the start of the constructors/destructors section. This is done in order to execute the static/global variables Constructors prior to calling main(), and to execute the static/global variables Destructors following main()'s execution.
- **Crtn.o** uses the 2 sections **crtn_ctor_sect** and **crtn_dtor_sect** in order to add the **"ret"** instruction to the end of the Constructors/Destructors sections respectively.

The skeleton Linker's script file in the **Compiler configuration file** places **crtn.o** as the last file to link, in order to locate **crtn_ctor_sect** after all **ctor_sect** constructor calls and **crtn_dtor_sect** after all **dtor_sect** destructor calls.

Note that the default crt0.c and Linker's script file require that **crtn_ctor_sect** and **crtn_dtor_sect** be located immediately next to **ctor_sect** and **dtor_sect** respectively. Thus, unless modifying Linker script file and crt0.c, the default crtn.o file is required for non C++ applications as well.

6.3 Default Linker Script File, Sections and Libraries

When the Linker is invoked by the Compiler driver (invocation generates an executable rather than an object/assembly file), a default Linker script file is generated for the Linker by the Compiler driver.

Important:

For new projects, it is advised to start from the default Linker script file and add objects, libraries and section mapping as necessary (note that unused libraries do not cost extra code size as they are identified as unreferenced symbols by the Linker and would not be linked). Starting from the default Linker script file is important since some objects and libraries sequence order is important as well as the mapping of some of the special sections.

Default Linker script file can be achieved in two options:

- The skeleton of that file can be found embedded in **.cfg** file located in the **CEVA-Toolbox** Software Development Tools release directory.
- The more direct way is to run the Compiler on a simple C/C++ (dummy) program with the **same** command line options the real program is compiled with, and then using the generated Linker's listing file (.LIN) to create the default script file. **Note** that different compilation options (e.g. –emul, -mfile-io) may lead to different Linker script files (generated by the Compiler), thus, it is important to use these options when creating a default Linker's script file through the Compiler – **there is no ONE default Linker script file**.

Following is an annotated Linker script file:

(Note that using non-default compilation switches may result in linkage of different libraries/section. For further details, refer to the Compilation Options chapter).

objects:
; Start up code, must always be the first object file
    %TOOLSPATH%\libs/crt0.o
; User's object file
    demo.o
;C++ constructor and destructor end sections, must always be the last object file.
    %TOOLSPATH%\libs/crtn.o
libraries:
; Contains standard library functions such as printf, strcmp etc.
    %TOOLSPATH%\libs/libc.lib
; I/O library and _exit symbol
    %TOOLSPATH%\libs/libios.lib
    %TOOLSPATH%\libs/libioe.lib
; Floating point and Arithmetic functions such as _fldiv , _mulhi3 etc.
    %TOOLSPATH%\libs/<core>lib.lib

code:
; The interrupt handlers table – declared in crt0.c
    inttbl

; Default code section name
    .text lo

; C++ Constructor and Destructor sections (crt0.c/crtn.c)
    ctor_sect
    crtn_ctor_sect
    dtor_sect
    crtn_dtor_sect

data:

; Default data section for initialized variables
    .data lo

; Default data section for uninitialized variables

```
        .bss lo

; Data section for variables that must not be initialized in the startup code
        .no_init lo

; Default data section for const variables
        const_data lo noload

; Section for storing program arguments (argc/argv) set from the debugger
        ARG_SECT lo
        ARG_SECT_END next noload

; Section for heap area (malloc/free)
        __MALLOC_SECT lo noload size 0x4000

; Section reserved for stack
        __STACK_SECT lo noload size 0x200

; Section for old I/O port implementation
        __OUTPORT_SECT lo noload
        __INPORT_SECT lo noload
```

The Compiler Driver uses the proper path names (for crt0.o, crtn.o and libraries) corresponding to the environment variables set up during the installation process.

6.4 I/O Support

Input/Output support for the **CEVA DSP Cores** includes the following chapters:

- **I/O support in Simulation**
- **Standard C stream I/O implementation (Simulation & Emulation)**
- **I/O utility programs**

6.4.1. I/O Support in Simulation

When using one of the simulation's I/O functions, it is required to perform the following steps:

1. Linking the application with **libios.lib -** this is done automatically by the Compiler's driver.
2. In the debugging session, a file should be associated (logically connected) with the appropriate port assigned to the **__OUTPORT_SECT** or **__INPORT_SECT** sections. The following C sample program and Debugger script file (both included with the Compiler files in the Tools **examples\simul\io** directory) demonstrates how this is done:

Example:
Following is the demo.c source file:

```
#include <stdio.h>

void main (int argc, char *argv[])
{
   printf ("hello world\n");
}
```

Following is the compilation line:

```
xm4cc -g3 demo.c
```

Following is a suggested Debugger script needed to load and run this demo program:

```
load coff dsp.a
; all mapping will be done automatically using the information
; from the loaded coff file. For more details on automatic memory mapping see
; the Debugger's Memory Mapping chapter.
go
;When this program is executed under the CEVA DSP Cores simulator, the ascii
;codes corresponding to "hello world\n" will be output to stdout
```

Important:

Make sure there is a correct match between the Linker script file and the Debugger script file in terms of section and address mappings and port connections. The Debugger script file must always be adjusted according to the Linker script file who in turn must be correlated with the code written in crt0.c.

6.4.2. Standard C Stream I/O Implementation

The standard C stream I/O support enables various file operations on the host's file system through the Debugger.

In order to activate stream I/O, programs must be compiled using the **-mfile-io** Compiler command line option .This option defines the macro FILE_IO=1, and adds **fileio.lib** and **libioe.lib** to the libraries used in the Linker script file (I/O emulation library is linked here even in Simulation mode as it contains crucial File-I/O components).

Example:

xm4cc file.c –mfile-io

Supported functions are:

- **void** clearerr (FILE *pStream);
- **int** fclose (FILE *pStream);
- **int** fcloseall (void);
- **int** feof (FILE *pStream);
- **int** ferror (FILE *pStream);
- **int** fflush (FILE *pStream);
- **int** fgetc (FILE *pStream);
- **int** fgetchar (void);
- **int** fgetpos (FILE *pStream, fpos_t *pPos);
- **char** *fgets (char *pString, int n, FILE *pStream);
- **int** fileno (FILE *pStream);
- **int** flushall (void);
- **FILE** *fopen (const char *pFileName, const char *pMode);
- **Int** fprintf (FILE *pStream, const char *pFormat, ...);
- **int** fputc (int c, FILE *pStream);
- **int** fputchar (int c);
- **int** fputs (const char *pString, FILE *pStream);
- **size_t** fread (void *pBuf, size_t size, size_t count, FILE *pStream);
- **FILE** *freopen (const char *pPath, const char *pMode, FILE *pStream);

- **int** fscanf (FILE *pStream, const char *pFormat, ...);
- **int** fseek (FILE *pStream, long offset, int origin);
- **int** fsetpos (FILE *pStream, const fpos_t *pPos);
- **long** ftell (FILE *pStream);
- **size_t** fwrite (const void *pBuf, size_t size, size_t count, FILE *pStream);
- **int** getc (FILE *p*Stream*);
- **int** getw (FILE **stream*);
- **int** putc (int c, FILE *p*Stream*);
- **int** putw (int *binint*, FILE **stream*);
- **void** rewind (FILE **stream*);
- **int** ungetc (int *c*, FILE **stream*);

The following sections provide an elaborate description of each routine.

6.4.2.2 Clearerr

Clear the end-of-file and error indicators for a stream

Syntax:

```
#include <stdio.h>
void clearerr( FILE *fp );
```

Description:

The *clearerr()* function clears the end-of-file and error indicators for the stream pointed to by *fp*. These indicators are cleared only when the file is opened, or by an explicit call to the *clearerr()* or *rewind()* functions.

Example:

```
#include <stdio.h>

void main()
 {
  FILE *fp;
  int c;

  c = 'J';
  fp = fopen( "file", "w" );
  if( fp != NULL ) {
   fputc( c, fp );
   if( ferror( fp ) ) {  /* if error        */
       clearerr( fp );   /* clear the error */
       fputc( c, fp );   /* and retry it    */
    }
  }
 }
```

6.4.2.3 fclose

Close a file

Syntax:

```
#include <stdio.h>
int fclose( FILE *fp );
```

Dscription:

The *fclose()* function closes the file *fp*. If there is any unwritten buffered data for the file, it is written out before the file is closed. Any unread buffered data is discarded. If the associated buffer was automatically allocated, it is deallocated.

returns 0 for success, non-zero when An error occurred.

Example:

```
#include <stdio.h>

void main()
 {
   FILE *fp;

   fp = fopen( "stdio.h", "r" );
   if( fp != NULL ) {
     fclose( fp );
   }
 }
```

6.4.2.4 fcloseall

close all open stream files, except stdin, stdout and stderr

Syntax:

```
#include <stdio.h>
int fcloseall( void );
```

Description:

The *fcloseall()* function closes all open stream files, except *stdin*, *stdout* and *stderr*. This includes streams created (and not yet closed) by *fdopen()*, *fopen()* and *freopen()*.

The *fcloseall()* function returns the number of streams that were closed, if no errors were encountered.

When an error occurs, EOF is returned.

Example:

```
#include <stdio.h>

void main()
  {
    printf( "The number of files closed is %d\n",
        fcloseall() );
  }
```

6.4.2.5 feof

Test the end-of-file indicator

Syntax:

```
#include <stdio.h>
int feof( FILE *fp );
```

Description:

The *feof()* function tests the end-of-file indicator for the stream pointed to by *fp*. Because this indicator is set when an input operation attempts to read past the end of the file, the *feof()* function detects the end of the file only after an attempt is made to read beyond the end of the file. Thus, if a file contains 10 lines, the *feof()* won't detect the end of the file after the tenth line is read; it will detect the end of the file once the program attempts to read more data.

The feof() function returns 0 if the end-of-file indicator is not set for *fp*, or non-zero if the end-of-file indicator is set.

Example:

```
#include <stdio.h>

void main()
  {
    FILE *fp;
    char buffer[100];

    fp = fopen( "file", "r" );
    fgets( buffer, sizeof( buffer ), fp );
    while( ! feof( fp ) ) {
      process_record( buffer );
      fgets( buffer, sizeof( buffer ), fp );
    }
    fclose( fp );
  }

void process_record( char *buf )
  {
    printf( "%s\n", buf );
  }
```

6.4.2.6 ferror

Test the error indicator for a stream

Syntax:

```
#include <stdio.h>
int ferror( FILE *fp );
```

Description:

The *ferror()* function tests the error indicator for the stream pointed to by *fp*.

Return value is 0 if the error indicator is not set, non-zero if the error indicator is set.

Example:

```
#include <stdio.h>
void main()
 {
   FILE *fp;
   int c;

   fp = fopen( "file", "r" );
   if( fp != NULL ) {
     c = fgetc( fp );
     if( ferror( fp ) ) {
       printf( "Error reading file\n" );
     }
   }
   fclose( fp );
 }
```

6.4.2.7 fflush

Flush the input or output buffer for a file

Syntax:

```
#include <stdio.h>
int fflush( FILE *fp );
```

Description:

If the file *fp* is open for output or update, the *fflush()* function causes any unwritten data to be written to the file. If the file *fp* is open for input or update, the *fflush()* function undoes the effect of any preceding ungetc operation on the stream. If the value of *fp* is NULL, then all files that are open will be flushed.

Return value is 0 if Successed, (-1) when n error occurred.

Example:

```
#include <stdio.h>

void main()
  {
    fflush( stdout );
  }
```

6.4.2.8 fgetc

Get the next character from a file

Syntax:

```
#include <stdio.h>
int fgetc( FILE *fp );
```

Description:

The *fgetc()* function gets the next character from the file designated by *fp*. The character is signed.

The *fgetc()* function returns the next character from the input stream pointed to by *fp*. If the stream is at end-of-file, the end-of-file indicator is set, and *fgetc()* returns EOF. If a read error occurs, the error indicator is set, and *fgetc()* returns EOF.

Example:

```
#include <stdio.h>

void main()
 {
  FILE *fp;
  int c;

  fp = fopen( "file", "r" );
  if( fp != NULL ) {
   while( (c = fgetc( fp )) != EOF )
     fputc( c, stdout );
   fclose( fp );
  }
 }
```

6.4.2.9 fgetchar

Get a character from stdin

Syntax:

```
#include <stdio.h>
int fgetchar( void );
```

Description:

The *fgetchar()* function is equivalent to *fgetc()* with the argument *stdin*.

The *fgetchar()* function returns the next character from the input stream pointed to by *stdin*.

If the stream is at end-of-file, the end-of-file indicator is set, and *fgetchar()* returns EOF.

If a read error occurs, the error indicator is set, and *fgetchar()* returns EOF.

Example:

```
#include <stdio.h>

void main()
 {
   FILE *fp;
   int c;

   fp = freopen( "file", "r", stdin );
   if( fp != NULL ) {
     while( (c = fgetchar()) != EOF )
       fputchar(c);
     fclose( fp );
   }
 }
```

6.4.2.10 fgetpos

Stores the current position of a file

Syntax:

```
#include <stdio.h>
int fgetpos( FILE *fp, fpos_t *pos );
```

Description:

The *fgetpos()* function stores the current position of the file *fp* in the object pointed to by *pos*. The value stored can be used by the *fsetpos()* function for repositioning the file to its position at the time of the call to the *fgetpos()* function.

Return value is 0 if the error indicator is not set, non-zero if the error indicator is set.

Example:

```
#include <stdio.h>

void main()
 {
   FILE *fp;
   fpos_t position;
   char buffer[80];

   fp = fopen( "file", "r" );
   if( fp != NULL ) {
     fgetpos( fp, &position ); /* get position     */
     fgets( buffer, 80, fp );  /* read record       */
     fsetpos( fp, &position ); /* set position     */
     fgets( buffer, 80, fp );  /* read same record */
     fclose( fp );
   }
 }
```

6.4.2.11 fgets

Get a string of characters from a file

Syntax:

```
#include <stdio.h>
char *fgets( char *buf, size_t n, FILE *fp );
```

Description:

The *fgets()* function gets a string of characters from the file designated by *fp*, and stores them in the array pointed to by *buf*. The *fgets()* function stops reading characters when:

- end-of-file is reached
- a newline character is read
- *n-1* characters have been read

whichever comes first. The new-line character is not discarded. A null character is placed immediately after the last character read into the array.

Note: A common programming error is to assume the presence of a new-line character in every string that is read into the array. A new-line character will not be present when more than *n*-1 characters occur before the new-line. Also, a new-line character might not appear as the last character in a file, just before end-of-file.

The *gets()* function is similar to *fgets()*, except that it operates with *stdin*, it has no size argument, and it replaces a newline character with the null character.

The *fgets()* function returns *buf* if successful.

NULL is returned if end-of-file is encountered, or a read error occurs.

Example:

```
#include <stdio.h>
void main()
  {
    FILE *fp;
    char buffer[80];

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      while( fgets( buffer, 80, fp ) != NULL )
        fputs( buffer, stdout );
      fclose( fp );
    }
  }
```

6.4.2.12 fileno

Returns the number of the file descriptor for a stream

Syntax:

```
#include <stdio.h>
int fileno( FILE *stream );
```

Description:

The *fileno()* function returns the number of the file descriptor for the file designated by *stream*. If an error occurs, a value of -1 is returned.

Example:

```
#include <stdio.h>

void main()
  {
    FILE *stream;

    stream = fopen( "file", "r" );
    printf( "File number is %d\n", fileno( stream ) );
    fclose( stream );
  }
```

produces output similar to the following:

```
File number is 7
```

6.4.2.13 flushall

Clears all input buffers and write all output buffers

Syntax:

```
#include <stdio.h>
int flushall( void );
```

Description:

The *flushall()* function clears all buffers associated with input streams, and writes any buffers associated with output streams. A subsequent read operation on an input file causes new data to be read from the associated file or device.

Calling the *flushall()* function is equivalent to calling the *fflush()* for all open stream files.

The *flushall()* function returns the number of open streams.

Example:

```
#include <stdio.h>

void main()
  {
    printf( "The number of open files is %d\n",
        flushall() );
  }
```

produces output similar to the following:

The number of open files is 4

6.4.2.14 fopen

Opens a file

Syntax:

```
#include <stdio.h>
FILE *fopen( const char *filename,
        const char *mode );
```

Description:

The ***fopen()*** function opens the file whose name is the string pointed to by *filename*, and associates a stream with it. The argument *mode* points to a string beginning with one of the following sequences:

- **r** – Open file for reading
- **w** - Create file for writing, or truncate to zero length
- **a** - Append: open text file or create for writing at end-of-file
- **r+** - Open file for update (reading and/or writing); use default file translation
- **w+** - Create file for update, or truncate to zero length; use default file translation
- **a+** - Append; open file or create for update, writing at end-of-file; use default file translation

The letter **b** may be added to any of the above sequences in the second or third position to indicate that the file is (or must be) a binary file DSPcores make a distinction between text and binary files, due to the fact that DSPcore word size is 16 bits. Opening a file in textual mode (deafult) always writes/reads Bytes, while in binary mode operation will be done in Words.

- Opening a file with read mode (r as the first character in the *mode* argument) fails if the file does not exist or it cannot be read.
- Opening a file with append mode (a as the first character in the *mode* argument) causes all subsequent writes to the file to be forced to the current end-of-file, regardless of previous calls to the fseek function.
- When a file is opened with update mode (+ as the second or third character of the *mode* argument), both input and output may be performed on the associated stream.

When a stream is opened in update mode, both reading and writing may be performed.

However, writing may not be followed by reading without an intervening call to the *fflush()* function, or to a file-positioning function (*fseek(), fsetpos(), rewind()*). Similarly, reading may not be followed by writing without an intervening call to a file-positioning function, unless the read resulted in end-of-file.

The *fopen()* function returns a pointer to the object controlling the stream. This pointer must be passed as a parameter to subsequent functions for performing operations on the file.

If the open operation fails, *fopen()* returns NULL.

Example:

```
#include <stdio.h>

void main()
 {
  FILE *fp;

  fp = fopen( "report.dat", "r" );
  if( fp != NULL ) {
    /* rest of code goes here */
    fclose( fp );
  }
 }
```

6.4.2.15 fprintf

Writes output to a file

Syntax:

```
#include <stdio.h>
int fprintf( FILE *fp, const char *format, ... );
```

Description:

The *fprintf()* function writes output to the file pointed to by *fp*, under control of the argument *format*. The *format* string is described under the description of the *printf()* function.

The *fprintf()* function returns the number of characters written, or a negative value if an output error occurred.

Example:

```
#include <stdio.h>

char *weekday = { "Saturday" };
char *month = { "April" };

void main()
  {
    fprintf( stdout, "%s, %s %d, %d\n",
      weekday, month, 18, 1987 );
  }
```

produces the output:

```
Saturday, April 18, 1987
```

6.4.2.16 fputc

Write a character to an output stream

Syntax:

```
#include <stdio.h>
int fputc( int c, FILE *fp );
```

Description:

The *fputc()* function writes the character specified by the argument *c* to the output stream designated by *fp*.

The *fputc()* function returns the character written. If a write error occurs, the error indicator is set, and *fputc()* returns EOF.

Example:

```
#include <stdio.h>

void main()
 {
   FILE *fp;
   int c;

   fp = fopen( "file", "r" );
   if( fp != NULL ) {
     while( (c = fgetc( fp )) != EOF )
       fputc( c, stdout );
     fclose( fp );
   }
 }
```

6.4.2.17 fputchar

Writes a character to stdout

Syntax:

```
#include <stdio.h>
int fputchar( int c );
```

Description:

The *fputchar()* function writes the character specified by the argument *c* to the output stream *stdout*. This function is identical to the *putchar()* function.

The function is equivalent to:

```
fputc( c, stdout );
```

The *fputchar()* function returns the character written. If a write error occurs, the error indicator is set, and *fputchar()* returns EOF When an error has occurred.

Example:

```
#include <stdio.h>

void main()
 {
   FILE *fp;
   int c;

   fp = fopen( "file", "r" );
   if( fp != NULL ) {
     c = fgetc( fp );
     while( c != EOF ) {
       fputchar( c );
       c = fgetc( fp );
     }
     fclose( fp );
   }
 }
```

6.4.2.18 fputs

Writes a character string to an output stream

Syntax:

```
#include <stdio.h>
int fputs( const char *buf, FILE *fp );
```

Description:

The *fputs()* function writes the character string pointed to by *buf* to the output stream designated by *fp*. The terminating null character is not written.

The *fputs()* function returns EOF if an error occurs; otherwise it returns a non-negative value.

Example:

```
#include <stdio.h>

void main()
 {
  FILE *fp_in, *fp_out;
  char buffer[80];

  fp_in = fopen( "file", "r" );
  fp_out = fopen( "outfile", "w" );
  if( fp_in != NULL && fp_out != NULL) {
   while( fgets( buffer, 80, fp_in ) != NULL )
    fputs( buffer, fp_out );
   fclose( fp_in );
   fclose( fp_out );
  }
 }
```

6.4.2.19 fread

Reads elements of a given size from a file

Syntax:

```
#include <stdio.h>
size_t fread( void *buf,
        size_t elsize,
        size_t nelem,
        FILE *fp );
```

Description:

The *fread()* function reads *nelem* elements of *elsize* bytes each from the file specified by *fp* into the buffer specified by *buf*.

The *fread()* function returns the number of complete elements successfully read. This value may be less than the requested number of elements.

The *feof()* and *ferror()* functions can be used to determine whether the end of the file was encountered or if an input/output error has occurred.

Example:

The following example reads a simple student record containing binary data. The student record is described by the struct student_data declaration.

```
#include <stdio.h>

struct student_data {
   int  student_id;
   unsigned char marks[10];
};

size_t read_data( FILE *fp, struct student_data *p )
  {
   return( fread( p, sizeof(*p), 1, fp ) );
  }

void main()
  {
   FILE *fp;
   struct student_data std;
   int i;
```

```
  fp = fopen( "file", "r" );
  if( fp != NULL ) {
    while( read_data( fp, &std) != 0 ) {
      printf( "id=%d ", std.student_id );
      for( i = 0; i < 10; i++ )
        printf( "%3d ", std.marks[ i ] );
      printf( "\n" );
    }
    fclose( fp );
  }
}
```

6.4.2.20 freopen

Reopens a stream

Syntax:

```
#include <stdio.h>
FILE *freopen( const char *filename,
        const char *mode,
        FILE *fp );
```

Description:

The stream located by the fp pointer is closed. The *freopen()* function opens the file whose name is the string pointed to by *filename*, and associates a stream with it. The stream information is placed in the structure located by the *fp* pointer.

The argument *mode* is described in the description of the fopen function.

The *freopen()* function returns a pointer to the object controlling the stream. This pointer must be passed as a parameter to subsequent functions for performing operations on the file. If the open operation fails, *freopen()* returns NULL.

Example:

```
#include <stdio.h>

void main()
 {
  FILE *fp;
  int c;

  fp = freopen( "file", "r", stdin );
  if( fp != NULL ) {
    while( (c = fgetchar()) != EOF )
      fputchar(c);
    fclose( fp );
  }
 }
```

6.4.2.21 fscanf

Scans input from a file

Syntax:

```
#include <stdio.h>
int fscanf( FILE *fp, const char *format, ... );
```

Description:

The *fscanf()* function scans input from the file designated by *fp*, under control of the argument *format*. Following the format string is a list of addresses to receive values. The *format* string is described under the description of the *scanf()* function.

The *fscanf()* function returns EOF when the scanning is terminated by reaching the end of the input stream. Otherwise, it returns the number of input arguments for which values were successfully scanned and stored.

Example:

To scan a date in the form “Saturday April 18 1987”:

```
#include <stdio.h>

void main()
  {
    int day, year;
    char weekday[10], month[10];
    FILE *in_data;

    in_data = fopen( "file", "r" );
    if( in_data != NULL ) {
      fscanf( in_data, "%s %s %d %d",
          weekday, month, &day, &year );
      printf( "Weekday=%s Month=%s Day=%d Year=%d\n",
          weekday, month, day, year );
      fclose( in_data );
    }
  }
```

6.4.2.22 fseek

Changes the read/write position of a file

Syntax:

```
#include <stdio.h>
int fseek( FILE *fp, long int offset, int where );
```

Description:

The *fseek()* function changes the read/write position of the file specified by *fp*. This position defines the character that will be read or written on the next I/O operation on the file. The argument *fp* is a file pointer returned by *fopen()* or *freopen()*. The argument *offset* is the position to seek to, relative to one of three positions specified by the argument *where*. Allowable values for *where* are:

SEEK_SET

The new file position is computed relative to the start of the file. The value of *offset* must not be negative.

SEEK_CUR

The new file position is computed relative to the current file position. The value of *offset* may be positive, negative or zero.

SEEK_END

The new file position is computed relative to the end of the file.

The *fseek()* function clears the end-of-file indicator, and undoes any effects of the *ungetc()* function on the same file.

The *ftell()* function can be used to obtain the current position in the file before changing it. The position can be restored by using the value returned by *ftell()* in a subsequent call to *fseek()* with the *where* parameter set to SEEK_SET.

The function returns 0 if succeeded; otherwise it returns a non-zero value.

Example:

The size of a file can be determined by the following example, which saves and restores the current position of the file:

```
#include <stdio.h>

long int filesize( FILE *fp )
  {
    long int save_pos, size_of_file;

    save_pos = ftell( fp );
    fseek( fp, 0L, SEEK_END );
    size_of_file = ftell( fp );
    fseek( fp, save_pos, SEEK_SET );
    return( size_of_file );
  }

void main()
  {
    FILE *fp;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      printf( "File size=%ld\n", filesize( fp ) );
      fclose( fp );
    }
  }
```

6.4.2.23 fsetpos

Sets the current file position

Syntax:

```
#include <stdio.h>
int fsetpos( FILE *fp, fpos_t *pos );
```

Description:

The *fsetpos()* function positions the file *fp* according to the value of the object pointed to by *pos*, which must be a value returned by an earlier call to the *fgetpos()* function on the same file.

The function returns 0 if succeeds; otherwise it returns a non-zero value.

Example:

```
#include <stdio.h>

void main()
 {
   FILE *fp;
   fpos_t position;
   char buffer[80];

   fp = fopen( "file", "r" );
   if( fp != NULL ) {
     fgetpos( fp, &position );   /* get position     */
     fgets( buffer, 80, fp );    /* read record      */
     fsetpos( fp, & position );  /* set position     */
     fgets( buffer, 80, fp );    /* read same record */
     fclose( fp );
   }
 }
```

6.4.2.24 ftell

Returns the current read/write position of a file

Syntax:

```
#include <stdio.h>
long int ftell( FILE *fp );
```

Description:

The *ftell()* function returns the current read/write position of the file specified by *fp*. This position defines the character that will be read or written by the next I/O operation on the file. The value returned by *ftell()* can be used in a subsequent call to *fseek()* to set the file to the same position.

The *ftell()* function returns the current read/write position of the file specified by *fp*. When an error is detected, -1L is returned.

Example:

```
#include <stdio.h>

long int filesize( FILE *fp )
  {
    long int save_pos, size_of_file;

    save_pos = ftell( fp );
    fseek( fp, 0L, SEEK_END );
    size_of_file = ftell( fp );
    fseek( fp, save_pos, SEEK_SET );
    return( size_of_file );
  }

void main()
  {
    FILE *fp;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      printf( "File size=%ld\n", filesize( fp ) );
      fclose( fp );
    }
  }
```

6.4.2.25 fwrite

Writes a number of elements into a file

Syntax:

```
#include <stdio.h>
size_t fwrite( const void *buf,
               size_t elsize,
               size_t nelem,
               FILE *fp );
```

Description:

The *fwrite()* function writes *nelem* elements of *elsize* bytes each to the file specified by *fp*. The *fwrite()* function returns the number of complete elements successfully written. This value will be less than the requested number of elements only if a write error occurs.

Example:

```
#include <stdio.h>

struct student_data {
   int  student_id;
   unsigned char marks[10];
};

void main()
  {
    FILE *fp;
    struct student_data std;
    int i;

    fp = fopen( "file", "w" );
    if( fp != NULL ) {
      std.student_id = 1001;
      for( i = 0; i < 10; i++ )
        std.marks[ i ] = (unsigned char) (85 + i);

      /* write student record with marks */
      i = fwrite( &std, sizeof(std), 1, fp );
      printf( "%d record written\n", i );
      fclose( fp );
    }
  }
```

6.4.2.26 getc

Gets the next character from a file.

Syntax:

#include <stdio.h>
int getc(FILE * *stream* **);**

Description:
The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Example:

```
/* Use getc and fputc to read and write characters
 * from a stream.
 */

#include <stdio.h>

void main()
  {
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      while( (c = getc( fp )) != EOF )
        fputc( c, stdout );
      fclose( fp );
    }
  }
```

6.4.2.27 getw

Gets an integer from a stream.

Syntax:

```
#include <stdio.h>
int getw( FILE *stream );
```

Description:

The **getw** function reads the next binary value of type **int** from the file associated with *stream* and increments the associated file pointer (if there is one) to point to the next unread character.

getw returns the integer value read. A return value of **EOF** indicates either an error or end of file. However, because the **EOF** value is also a legitimate integer value, use **feof** or **ferror** to verify an end-of-file or error condition.

Example:

```
/* GETW.C: This program uses getw to read a word
 * from a stream, then performs an error check.
 */

#include <stdio.h>
#include <stdlib.h>
void main( void )
{
  FILE *stream;
  int i;
  if( (stream = fopen( "getw.c", "rb" )) == NULL )
    printf( "Couldn't open file\n" );
  else
  {
    /* Read a word from the stream: */
    i = getw( stream );
    /* If there is an error... */
    if( ferror( stream ) )
    {
      printf( "getw failed\n" );
      clearerr( stream );
    }
    else
      printf( "First data word in file: 0x%.4x\n", i );
    fclose( stream );
  }
}
```

6.4.2.28 putc

Writes a character to an output stream

Syntax:

#include <stdio.h>
int putc(int c, **FILE ***stream* **);**

Description:

The **putc** function is equivalent to **fputc**, except that if it is implemented as a macro, it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Example:

```
/* Use getc and putc to read and write characters
 * from a stream.
 */

#include <stdio.h>

void main()
  {
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      while( (c = getc( fp )) != EOF )
        putc( c, stdout );
      fclose( fp );
    }
  }
```

6.4.2.29 putw

Writes an integer to a stream.

Syntax:

#include <stdio.h>
int putw(int *binint***, FILE ****stream* **);**

Description:

The **putw** function writes a binary value of type **int** to the current position of *stream.*

putw returns the value written. A return value of **EOF** may indicate an error. As **EOF** is also a legitimate integer value, use **ferror** to verify an error.

Example:

```
/* PUTW.C: This program uses putw to write a
 * word to a stream, then performs an error check.
 */

#include <stdio.h>
#include <stdlib.h>

void main( void )
{
  FILE *stream;
  unsigned u;
  if( (stream = fopen( "data.out", "wb" )) == NULL )
    exit( 1 );
  for( u = 0; u << 10; u++ )
  {
    putw( u + 0x2132, stdout );
    putw( u + 0x2132, stream );   /* Write word to stream. */
    if( ferror( stream ) )        /* Make error check. */
    {
      printf( "putw failed" );
      clearerr( stream );
      exit( 1 );
    }
  }
  printf( "\nWrote ten words\n" );
  fclose( stream );
}
```

6.4.2.30 rewind

Sets the file position indicator to the beginning of the file

Syntax:

```
#include <stdio.h>
void rewind( FILE *fp );
```

Description:

The *rewind()* function sets the file position indicator for the stream indicated by *fp* to the beginning of the file. It is equivalent to the following:

```
fseek( fp, 0L, SEEK_SET );
```

except that the error indicator for the stream is cleared.

Example:

```
#include <stdio.h>

static assemble_pass( int passno )
 {
  printf( "Pass %d\n", passno );
 }

void main()
 {
  FILE *fp;

  if( (fp = fopen( "program.asm", "r")) != NULL ) {
  assemble_pass( 1 );
  rewind( fp );
  assemble_pass( 2 );
  fclose( fp );
  }
 }
```

6.4.2.31 ungetc

Pushes a character back onto an input stream

Syntax:

```
#include <stdio.h>
int ungetc( int c, FILE *fp );
```

Description:

The *ungetc()* function pushes the character specified by *c* back onto the input stream pointed to by *fp*. This character will be returned by the next read on the stream. The pushed-back character is discarded if a call is made to the *fflush()* function or to a file-positioning function (*fseek()*, *fsetpos()* or *rewind()*) before the next read operation is performed.

Only one character (the most recent one) of pushback is remembered.

The *ungetc()* function clears the end-of-file indicator, unless the value of *c* is EOF.

Returns:

The character pushed back

Example:

```
#include <stdio.h>
#include <ctype.h>

void main()
  {
    FILE *fp;
    int c;
    long value;

    fp = fopen( "file", "r" );
    value = 0;
    c = fgetc( fp );
    while( isdigit(c) ) {
      value = value*10 + c - '0';
      c = fgetc( fp );
    }
    ungetc( c, fp ); /* put last character back */
    printf( "Value=%ld\n", value );
    fclose( fp );
  }
```

6.4.3. I/O Utility Programs

6.4.3.1 Dspprnt

Usage:

dspprnt <file_name> [BACK [<data_word_size>]]

BACK - Transforms text to hex numbers

Default is hex to text

data_word_size - DSPCore data word size (16/20/24)

Default is 16-bit

Valid only when used with BACK

Description:

When connecting to an output file, ASCII codes (0xXXXX - hex numbers) corresponding to the string's characters are output to the file. By using the ***dspprnt.exe*** utility, it is possible to convert the stored hex numbers in the file (representing the ASCII codes of the printed string) into readable character strings. Similarly, the ***dspprnt.exe*** utility is capable to convert back a text string to its representative hex/ASCII numbers (by using the BACK command line option, activate the ***dspprnt.exe*** for getting its usage), this can be useful for preparing text input files readable by the Debugger (as input connected files).

Examples:

1. Converting the debugger's output file into a readable form:

The file cctmp.ou is generated by the "**connect file cctmp.ou port D:F7FE output**" debugger CLI command.

The command:

dspprnt cctmp.ou

Will print the contents of this file in a readable form.

2. Converting a text file into a debugger's readalbe form:

The file cctmp.in is generated by the command:

dspprnt myfile.txt BACK 16 > cctmp.in

It is now possible to use the "**connect file cctmp.in port D:F7FF input**" debugger CLI command and utilize C functions such as getchar() and scanf().

6.5 Run Time Library Functions - Details

This chapter describes the Runtime Support Library Functions supplied with the CEVA-Toolbox C Cross Compiler. The complete functions' list divided into categories is provided first followed by a detailed syntax, description and "how to use" example for each function.

String/Memory Manipulation Functions (string.h):

memchr strcat strcspn strpbrk strtol

memcmp strchr strlen strrchr strtoul

memcpy memmove strcmp strncat strspn

mmove strcoll strncmp strstr

memset strcpy strncpy strtod

Mathematical Library Functions (math.h and stdlib.h):

Floating Point:

acos div labs pow tan

asin exp ldexp rand tanh

atan atan2 fabs ldiv sin

ceil Floor Log sinh

cos fmod log10 sqrt

cosh frexp modf srand

Fixed Point:

Q14_cos_Q12 Q14_cos_Q0

Q14_sini_Q12 Q14_sini_Q0

Q14_log2_Q0 Q7_sqrt_Q0

_isqrt

Type Handling Functions or Macros (ctype.h):

isalnum isdigit isprint isupper toupper

isalpha isgraph ispunct isxdigit

iscntrl islower isspace tolower

varargs Macros (stdarg.h):

va_end va_start va_arg

Standard I/O Functions (stdio.h):

clearerr fclose fcloseall feof ferror fflush

fgetc fgetchar fgetpos fgets fileno flushall

fopen fprintf fputc fputchar fputs fread

freopen fscanf fseek fsetpos ftell fwrite getc

getchar gets getw ifprintf iprintf isprintf

ivprintf ivsprintf printf putc putchar

puts _puts putw rewind scanf sprintf

sscanf ungetc vprintf vsprintf

Miscellaneous Functions (stdlib.h, setjmp.h):

abort atol exit malloc

abs atof free realloc

atoi calloc longjmp setjmp

Clock Sampling Functions (ceva-time.h):

start_clock pause_clock

reset_clock clock

6.5.1. abort

Syntax:

```
#include <stdlib.h>
        void abort ( void );
```

Description:

The **abort** function causes abnormal termination of a program. It calls the exit function (defined in crt0.c) with an error code which is non-zero.

Example:

```
#include <stdlib.h>
void main( void )
{
        if ( ! malloc ( 1000 ) )
         abort ( ); /* if not enough memory, abort program */
 }
```

See also: exit

6.5.2. abs

Syntax:

```
#include <stdlib.h>
 int abs ( int i );
```

Description:

The **abs** function returns the absolute value of an integer.

Example:

```
#include <stdlib.h>
        void main( void )
        {
                int x = - 4;
                int y = abs ( x ); /* y = 4 */
        }
```

See also: labs

6.5.3. acos

Syntax:

```
#include <math.h>
double acos ( double x );
```

Description:

The **acos** function returns the arc cosine value of a floating point argument. The argument must be in the range of [-1,+1], while the return argument is an angle in the range [0, π] radians.

Example:

```
#include <math.h>
void main( void )
{
        double x = 0.0;
        double y = acos ( x ) /* y = π/2 */
}
```

See also: cos

6.5.4. asin

Syntax:

```
#include <math.h>
double asin ( double x );
```

Description:

The **asin** function returns the arc sine value of a floating-point argument. The argument must be in the range of [-1,+1], while the return argument is an angle in the range [- π/2, π/2] radians.

Example:

```
#include <math.h>
void main( void )
{
        double x = 1.0;
double y = asin ( x );    /* y = π/2 */
}
```

See also: sin

6.5.5. atan

Syntax:

```
#include <math.h>
double atan ( double x );
```

Description:

The **atan** function returns the arc tangent value of a floating point argument. The argument must be in the range of [-1.0,+1.0], while the return argument is an angle in the range [- π/2, + π/2] radians.

Example:

```
#include <math.h>void main( void )
{
        double x = 1.0;        double y = atan ( x ); /* y = π/4 */
 }
```

See also:tan

6.5.6. atan2

Syntax:

```
#include <math.h>
        double atan2 ( double y, double x );
```

Description:

The **atan2** function returns the arc tangent value of a floating point argument y divided by the argument x (i.e. y/x).

Example:

```
#include <math.h>
void main( void )
{               double x = 1.0, y = 0.5;
        double y = atan2 ( y, x );
 }
```

See also:tan, atan

6.5.7. atof

Syntax:

```
#include <stdlib.h>
double atof ( char *p );
```

Description:

The **atof** function converts a character string to a floating point number and returns the value of that number. The argument is a pointer to the character string. The string must have the following format:

[space] [sign] digits [.digits] [e|E [sign] integer]

Where space can be one or more white space characters (space, horizontal tab, vertical tab, carriage return, formfeed, newline), The first character that cannot be part of the number terminating the string.

Example:

```
#include <stdlib.h>
double x;
x = atof ( "2.51e-3" ); /* x = 0.00251 */
```

See also:atoi, atol

6.5.8. atoi

Syntax:

```
#include <stdlib.h>
int atoi ( char *p );
```

Description:

The **atoi** function converts a character string to an integer number and returns the value of that number. The argument is a pointer to the character string. The string must have the following format:

[space] [sign] digits [.digits] [e|E [sign] integer]

Where space can be one or more white space characters (space, horizontal tab, vertical tab, carriage return, formfeed, newline), the first character that cannot be part of the number terminating the string.

Example:

```
#include <stdlib.h>

int x;
x = atoi ( "-219e1" ); /* x = -2190 */
```

See also: atof, atol

6.5.9. atol

Syntax:

```
#include <stdlib.h>

long atol ( char *p );
```

Description:

The **atol** function converts a character string to a long integer number and returns the value of that number. The argument is a pointer to the character string. The string must have the following format:

[space] [sign] digits [.digits] [e|E [sign] integer]

Where space can be one or more white space characters (space, horizontal tab, vertical tab, carriage return, formfeed, newline), the first character that cannot be part of the number terminating the string.

Example:

```
#include <stdlib.h>

long x
x = atol ( "-1257" ); /* x = -1257 */
```

See also: atof, atoi

6.5.10. calloc

Syntax:

```
#include <stdlib.h>

void * calloc ( size_t nwords, size_t size );
```

Description:

The **calloc** function allocates and initializes with zero, size words for each of nwords objects and returns a pointer to the space. If it cannot allocate the memory, it returns a null pointer.

The memory that calloc uses is in a special memory pool called the heap. One can setup the location of the heap in the crt0.c file. See chapter In-Line __asm__() Extension for more details.

Example:

```
#include <stdlib.h>

int * p;
    p = calloc ( 10, 1 ); /* allocate and clear 10 words */
```

See also: malloc, free, init_malloc

6.5.11. ceil

Syntax:

```
#include <math.h>

double ceil ( double x );
```

Description:

The **ceil** function returns a floating point number that represents the smallest integer that is larger or equal to x.

Example:

```
#include <math.h>

double f;
f = ceil ( 2.13204 ); /* f = 3.0 */
f = ceil ( -4.103 ); /* f = -4.0 */
```

See also: floor

6.5.12. clock

Syntax:

```
include <ceva-time.h>

static inline clock_t clock();
```

Description:

The **clock** function returns the processor time consumed by the program (in cycles).

Example:

```
#include <ceva-time.h>
#include <stdio.h>

void main( void )
 {
        clock_t c1,c2;
        c1 = clock();
        foo ();
        c2 = clock();
        printf("function foo takes %d cycles\n", c2-c1);
 }
```

See also: start_clock, pause_clock, reset_clock

6.5.13. cos

Syntax:

include <math.h>

double cos (double x);

Description:

The **cos** function returns the cosine value of a floating point argument. The argument must be expressed in radians, the result will be in the range [-1.0,1.0].

Example:

```
#include <math.h>

void main( void )
 {
        double x = 3.1415927;
        double y = cos ( x ); /* y = -1.0 */
 }
```

See also: acos, sin, Q14_cos_Q12, Q14_cosi_Q0

6.5.14. cosh

Syntax:

```
#include <math.h>

double cosh ( double x );
```

Description:

The **cosh** function returns the hyperbolic cosine value of a floating point argument. A range error occurs if the magnitude of x is too large.

Example:

```
#include <math.h>

void main( void )

{
        double x = 0.0;
        double y = cosh ( x )   /* y = 1.0 */
}
```

See also: cos, sinh

6.5.15. div

Syntax:

```
#include <stdlib.h>

div_t div( int numerator, int denomerator );
```

Description:

The **div** function computes the quotient and remainder of the division of the numerator by the denominator. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined. The function returns a structure of type div_t, comprising both the quotient and the remainder:

```
typedef struct
{
        int quot; /* quotient */
        int rem; /* remainder */
} div_t;
```

The sign of the quotient is the same as the algebraic quotient and the sign of the remainder is the same as the sign of the numerator.

Example:

```
#include <stdlib.h>

void main( void )
{
        int x = -10;
        int y = 3;
        div_t result = div( x, y )&#9;/* result.quot = -3 ; result.rem = -1 */
}
```

See also: ldiv

6.5.16. exit

Syntax:

```
#include <stdlib.h>

void exit( int status );
```

Description:

The **exit** function causes normal program termination to occur. The user can modify the exit function to perform application specific "ending" operations, by modifying the crt0.c file, where it is defined. See chapter In-Line __asm__() Extension for more details.

Example:

```
#include <stdlib.h>

void main( void )

{
        int k = 1;
        exit( k );
}
```

See also: abort

6.5.17. exp

Syntax:

```
# include <math.h>

double exp ( double x );
```

Description:

The **exp** function returns the exponential function of the argument, x. The return value is the number e (2.718281) raised to the power of x. A range error occurs if the magnitude of x is too large.

Example:

```
#include <math.h>

void main( void )
{
        double x = 2.0;
        double y = exp ( x ); /* y = 7.389056 (e**2) */
}
```

See also: frexp, ldexp, log

6.5.18. fabs

Syntax:

```
#include <math.h>

double fabs ( double x );
```

Description:

The **fabs** function returns the absolute value of a floating point number, x.

Example:

```
#include <math.h>

void main( void )

{
        double x = - 4.75;
        double y = fabs ( x ); /* y = +4.75 */
}
```

See also: abs, labs

6.5.19. floor

Syntax:

```
#include <math.h>

double floor ( double x );
```

Description:

The **floor** function returns a floating point number that represents the largest integer that is less than or equal to x.

Example:

```
#include <math.h>

double f;
f = floor ( 2.13204 ); /* f = 2.0 */
f = ceil ( -4.103 ); /* f = -5.0 */
```

See also: ceil

6.5.20. fmod

Syntax:

```
#include <math.h>

double fmod ( double x, double y );
```

Description:

The **fmod** function returns the remainder after dividing x by y an integral number of times. If y=0, the function returns 0.

Example:

```
#include <math.h>

double x = 10.0;
double y = 3.0;
double z = fmod ( x, y ); /* z = 1.0 */
```

See also: div, ldiv

6.5.21. free

Syntax:

```
#include <stdlib.h>

void free ( void * p );
```

Description:

The **free** function deallocates memory space (pointed to by p) that was previously allocated by a malloc or calloc function call. This makes that memory space available again. If unallocated space is attempted to be freed, no action is performed and no return value is given.

Example:

```
#include <stdlib.h>

char * ptr;
ptr = malloc ( 10 ); /* allocate 10 words */
free ( ptr ); /* free 10 words */
```

See also: calloc, malloc

6.5.22. frexp

Syntax:

```
#include <math.h>

double frexp ( double value, int *exp );
```

Description:

The **frexp** function breaks a floating point number into a normalized fraction and an integral power of 2. It stores the integer in the int object pointed to by exp. The frexp function returns a float number x with magnitude in the interval [1/2,1) or 0, such that value = x * 2**exp. If value is 0, both parts of the result are 0.

Example:

```
#include <math.h>

void main( void )

{
        int exp;
        double f = frexp ( 3.0, &exp ); /* f = .75 and exp = 2 */
}
```

See also: exp, ldexp, log

6.5.23. getchar

Syntax:

```
#include <stdio.h>

int getchar ( void );
```

Description:

The **getchar** function obtains the next character (if present) as an unsigned char converted to an int, from the standard input stream. For simulation purposes the standard input stream is a memory mapped I/O address in the DSP processor's data space. If this address is "connected" to a file, implicit reading from files is possible, otherwise, reading from the keyboard is performed. See the ***CEVA-Toolbox Debugger User's Guide*** of the Debugger for more details. If the stream is at end-of-file, getchar returns EOF.

Example:

```
#include <stdio.h>

void main( void )
{
        int n = getchar ( );
}
```

See also: scanf, sscanf, gets

6.5.24. gets

Syntax:

```
#include <stdio.h>

char *gets ( char *s );
```

Description:

The **gets** function reads characters from the input stream pointed to by **stdin** into the array pointed to by s until end-of-file is encountered or a new-line characted is read. The **gets** function returns s if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned.

For simulation purposes the standard input stream is a memory mapped I/O address in the DSP processor's data space. If this address is "connected" to a file, implicit reading from files is possible, otherwise, reading from the keyboard is performed. See the ***CEVA-Toolbox Debugger User's Guide*** of the Debugger for more details.

Example:

```
#include <stdio.h>

void main( void )
{
        char buf [20];
        char str[20];
        str = gets(buf);
}
```

See also: scanf, sscanf, getchar

6.5.25. isgraph

Syntax:

```
#include <ctype.h>

int isgraph ( char c );
```

Description:

The **isgraph** function or macro tests the argument character to identify it as a non-space character. The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = 'a';
        int k = isgraph ( c ) /* k is non-zero */
}
```

See also: isalpha, isdigit, isprint, isspace

6.5.26. isprint

Syntax:

```
#include <ctype.h>

int isprint ( char c );
```

Description:

The **isprint** function or macro tests the argument character to identify printable ASCII characters, including spaces (ASCII codes 32-126). The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = '&';
        int k = isprint ( c ); /* k is non-zero */
}
```

See also: isalpha, iscntrl, isdigit, ispunct, isspace

6.5.27. isprintf

Syntax:

```
#include <stdio.h>

int isprintf ( char *s, const char *format, ... );
```

Description:

The **isprintf** function is equivalent to the sprintf function, except that it does not handle floating point types, thus achieving smaller code size and reduced execution time.

Example:

```
#include <stdio.h>

void main( void )
{
        long r1 = 43;
        char buf[20];
        char s[10] = "Hello !";
        isprintf ( buf, "%s result is:%ld\n", s, r1 );
}
```

See also: printf, iprintf, vprintf, ivprintf, putchar, puts, _puts, sprintf, ivsprintf

6.5.28. ivprintf

Syntax:

```
#include <stdarg.h>
#include <stdio.h>

int ivprintf ( const char *format, va_list arg );
```

Description:

The **ivprintf** (integer vprintf) function is a smaller and faster version of vprintf. The improvement is achieved by not supporting floating point types in the printing. As a result, the floating point special characters for the format string (%f/%g/%e) are not supported by ivprintf.

Example:

```
#include <stdarg.h>
#include <stdio.h>

void error( char *function_name, char *format, … )
{
        va_list args;
        va_start ( args, format );

        /* print out name of function causing error */
        iprintf ( "Error in %s: “, function_name );

        /* print out remainder of message */
        ivprintf ( “format, args” );

        va_end ( args );
}
```

See also: printf, iprintf, vprintf, putchar, puts, _puts, sprintf, isprintf, ivsprintf

6.5.29. ivsprintf

Syntax:

```
#include <stdio.h>

int ivsprintf ( char *s, const char *format, va_list arg );
```

Description:

The **ivsprintf** (integer vsprintf) function is a smaller and faster version of vsprintf. The improvement is achieved by not supporting floating point types in the printing. As a result, the floating point special characters for the format string (%f/%g/%e) are not supported by ivsprintf.

Example:

```
#include <stdarg.h>
#include <stdio.h>

char *error( char *function_name, char *format, … )
{
        char error_message[80];
        va_list args;
        va_start ( args, format );

        /* print out name of function causing error */
        isprintf ( error_message , "Error in %s: “, function_name );

        /* print out remainder of message */
        ivsprintf ( error_message, “format, args” );
        va_end ( args );
        return error_message;
}
```

See also: printf, iprintf, vprintf, ivprintf, putchar, puts, _puts, sprintf, isprintf

6.5.30. isupper

Syntax:

```
#include <ctype.h>

int isupper ( char c );
```

Description:

The **isupper** function or macro tests the argument character to identify uppercase ASCII alphabetic characters. The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = 'U';
        int k = isupper ( c ); /* k is non-zero */
}
```

See also: isalnum, isalpha, islower, isprint

6.5.31. isxdigit

Syntax:

```
#include <ctype.h>

int isxdigit ( char c );
```

Description:

The **isxdigit** function or macro tests the argument character to identify hexadecimal digits (0-9, a-f, A-F). The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = ',';
        int k = isxdigit ( c ); /* k is non-zero */
}
```

See also: isalnum, isalpha, isdigit, isprint

6.5.32. labs

Syntax:

```
#include <stdlib.h>

long labs ( long l );
```

Description:

The **labs** function returns the absolute value of a long integer.

Example:

```
#include <stdlib.h>

void main( void )
{
        long x = - 4;
        long y = labs ( x ); /* y = 4 */
}
```

See also: abs

6.5.33. ldexp

Syntax:

```
#include <math.h>

double ldexp ( double x, int exp );
```

Description:

The **ldexp** function multiplies a floating point number by a power of 2 and returns x * 2**exp. The exponent, exp, can be a negative or positive value. A range error occurs if the magnitude of x or exp is too large.

Example:

```
#include <math.h>

void main( void )
{
        double x = 0.5;
        double y = ldexp ( x, 6 ) /* y = 32.0 */
        y = ldexp ( 3.0, -2 ) /* y = 0.75 */
  }
```

See also: exp, frexp

6.5.34. ldiv

Syntax:

```
#include <stdlib.h>

ldiv_t ldiv( long numerator, long denominator );
```

Description:

The **ldiv** function computes the quotient and remainder of the division of the numerator by the denominator. If the long division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined. The function returns a structure of type ldiv_t, comprising both the quotient and the remainder:

```
typedef struct
{
        long quot; /* quotient */
        long rem; /* remainder */

} ldiv_t;
```

The sign of the quotient is the same as the algebraic quotient and the sign of the remainder is the same as the sign of the numerator.

Example:

```
#include <stdlib.h>

void main( void )
{
        long x = -10;
        long y = 3;
        ldiv_t result = ldiv( x, y )&#9;/* result.quot = -3 ; result.rem = -1 */
}
```

See also: div

6.5.35. log

Syntax:

```
#include <math.h>

double log ( double x );
```

Description:

The **log** function returns the natural logarithm of a floating point number, x. A domain error occurs if x is negative and a range error occurs if x=0.

Example:

```
#include <math.h>

void main( void )
{
        double x = 2.718282;
        double y = log ( x ); /* y = 1.0 */
}
```

See also: exp, log10, Q14_log2_Q0

6.5.36. log10

Syntax:

```
#include <math.h>

double log ( double x );
```

Description:

The **log10** function returns the base 10 logarithm of a floating point number, x. A domain error occurs if x is negative and a range error occurs if x=0.

Example:

```
#include <math.h>

void main( void )
{
        double x = 100.0;
        double y = log10 ( x )   /* y = 2.0 */
}
```

See also: exp, log, Q14_log2_Q0

6.5.37. longjmp

Syntax:

```
#include <setjmp.h>
jmp_buf env;
int longjmp ( jmp_buf env, int val );
```

Description:

setjmp() and **longjmp** () are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. longjmp () restores the environment saved by the last call of setjmp, and then returns in such a way that execution continues as if the call of setjmp() had just returned the value val to the function that invoked setjmp(); however, if val were zero, execution would continue as if the call of setjmp() had returned one. This ensures that a return from setjmp() caused by a call to longjmp () can be distinguished from a regular return from setjmp(). The calling function must not itself have returned in the interim, otherwise longjmp () will be returning control to a possibly non-existent environment. All local stack data have values as of the time longjmp () was called. The callee-saved registers are restored to the values they had at the time that setjmp() was called.

Example:

```
jmp_buf my_env;
if ( !setjmp( my_env ) )
{
     /* this is the usual path (return from setjmp) */
    my_proc();
}
else
{
     /* error occured:
     code after return from longjmp, (stack and register environment restored) */
}
void my_proc( )
{
         /* return non-zero value to restore environment */
         if( error_condition )
         longjmp ( my_env, 2 );
}
```

See also: setjmp

6.5.38. isalnum

Syntax:

```
#include <ctype.h>

int isalnum ( char c );
```

Description:

The **isalnum** function or macro tests the argument character to identify it as an alphabetic ASCII character (isalpha) or a numeric character (isdigit). The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>
void main( void )
{
        char c = 'A';
        int k = isalnum ( c ); /* k is non-zero */
}
```

See also: isalpha, isdigit

6.5.39. isalpha

Syntax:

```
#include <ctype.h>

int isalpha ( char c );
```

Description:

The **isalpha** function or macro tests the argument character to identify it as an alphabetic ASCII character (A,B,...,Z,a,b,...,z). The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = 'b';
        int k = isalpha ( c ); /* k is non-zero */
}
```

See also: isalnum, isdigit, islower, isupper

6.5.40. isdigit

Syntax:

```
#include <ctype.h>

int isdigit ( char c );
```

Description:

The **isdigit** function or macro tests the argument character to identify it as a numeric character ('0'-'9'). The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = '3';
        int k = isdigit ( c ); /* k is non-zero */
}
```

See also: isalpha, isxdigit

6.5.41. islower

Syntax:

```
#include <ctype.h>

int islower ( char c );
```

Description:

The **islower** function or macro tests the argument character to identify it as a lowercase alphabetic ASCII character ('a'-'z'). The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = 'd';
        int k = islower ( c ); /* k is non-zero */
}
```

See also: isalnum, isalpha, isupper

6.5.42. ispunct

Syntax:

```
#include <ctype.h>

int ispunct ( char c );
```

Description:

The **ispunct** function or macro tests the argument character to identify ASCII punctuation characters. The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = ',';
        int k = ispunct ( c ); /* k is non-zero */
}
```

See also: isalnum, isalpha, iscntrl, isdigit, isprint, isspace

6.5.43. ifprintf

Syntax:

```
#include <stdio.h>

int ifprintf (FILE *fp, const char *format, ... );
```

Description:

The **ifprintf** (integer fprintf) function is a smaller and faster version of fprintf. The improvement is achieved by not supporting floating point types in the printing. As a result, the floating point special characters for the format string (%f/%g/%e) are not supported by ifprintf.

Example:

```
#include <stdio.h>

void main( void )
{
        long lvar = 44;
        ifprintf (stdout, "num=%ld\n", lvar );
}
```

See also: fprintf, ifprintf, fputc, putc, fputchar, fputs, sprintf, isprintf

6.5.44. iprintf

Syntax:

```
#include <stdio.h>

int iprintf ( const char *format, ... );
```

Description:

The **iprintf** (integer printf) function is a smaller and faster version of printf. The improvement is achieved by not supporting floating point types in the printing. As a result, the floating point special characters for the format string (%f/%g/%e) are not supported by iprintf.

Example:

```
#include <stdio.h>

void main( void )
{
        long lvar = 44;
        iprintf ("num=%ld\n", lvar );
}
```

See also: printf, vprintf, ivprintf, putchar, puts, _puts, sprintf, isprintf, ivsprintf

6.5.45. iscntrl

Syntax:

```
#include <ctype.h>

int iscntrl ( char c );
```

Description:

The **iscntrl** function or macro tests the argument character to identify it as a control character (ASCII code 0-31 or 127). The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>
void main( void )
{
        char c = 30;
        int k = iscntrl ( c ) /* k is non-zero */
}
```

See also: isalpha, isdigit, isprint, isspace

6.5.46. isspace

Syntax:

```
#include <ctype.h>

int isspace ( char c );
```

Description:

The **isspace** function or macro tests the argument character to identify the ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed and newline characters. The function returns a non-zero value if the test is true, otherwise it returns 0.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = '\n';
        int k = isspace ( c ); /* k is non-zero */
 }
```

See also: iscntrl, isdigit, isprint, ispunct

6.5.47. _isqrt

Syntax:

```
#include <math.h>

short _isqrt ( long x );
```

Description:

The **_isqrt** function returns the square root of a 32 bits signed long positive integer number, x. The value returned is a signed 16 bits integer value.

Example:

```
#include <math.h>

void main( void )
{
        long x = 49;
         short y = _isqrt ( x ); /* y = 7*/
}
```

See also: sqrt, Q7_sqrt_Q0

6.5.48. init_malloc

Syntax:

```
#include <stdlib.h>

void * init_malloc( void *malloc_start_block, int size, int restore );
```

Description:

Input parameters:

malloc_start_sect	- pointer to the begining of a memory area to be used for the following **malloc/calloc/free** calls.
size	- size of the memory area to be used.
restore	- initializes a new malloc area if set to 0, or use the old values in that memory area if set to a non zero value.

Returns:
A pointer to the start of the previously used malloc area.

The **init_malloc** function initializes the memory area to be used in the **malloc/calloc/free** functions. An application can have an unlimited different memory areas that can be used in different **malloc** calls, while the **init_malloc** can be used for altering between them.

The **init_malloc** function is called from the startup code (crt0.c) for initializing the default memory area for **malloc** which is __MALLOC_SECT. For more details on the **malloc** function see its description.

Example:

```
#include <stdlib.h>

void xmalloc_sect  __attribute__((section(".DSECT   __X_MALLOC_SECT")));
void ymalloc_sect __attribute__((section(".DSECT __Y_MALLOC_SECT")));

#define XARR_SIZE 100
#define YARR_SIZE 150

void main( void )
{
        int *xarr;
        int *yarr;
        void *rc;

        /* initialize xram malloc session */
        rc = (void *)init_malloc( &malloc_sect, XARR_SIZE * 10, 0 );
        if (! rc)
        {
                printf("First time xmalloc detected\n");
        }
        xarr = ( int *)malloc( sizeof(int) * XARR_SIZE );

        /* initialize yram malloc session */
        rc = (void *)init_malloc( &ymalloc_sect, YARR_SIZE * 10, 0 );
        if (!rc)
        {
                printf("First time ymalloc detected\n");
        }
        yarr = ( int *)malloc( sizeof(int) * YAR_SIZE );

        /*
        user code…
        */

        /* free both sessions */
        init_malloc( &ymalloc_sect, YARR_SIZE * 10, 1 );
        free(yarr);
        init_malloc( &xmalloc_sect, XARR_SIZE * 10, 1 );
        free(xarr);
}
```

See also: malloc, calloc, free

6.5.49. malloc

Syntax:

```
#include <stdlib.h>

void * malloc ( size_t size );
```

Description:

The **malloc** function allocates space in the heap area of the data memory for an object of size bytes and returns a void pointer to that space. If malloc cannot allocate the space, it returns a null pointer usually indicating that the data memory allocated for the heap memory is too small.

The size of the heap memory is controlled by a special section - __MALLOC_SECT, whose size (0x4000 by default) is specified in the Linker script file. See *Section 6.3 - Default Linker Script File, Sections and Libraries* for more details.

Starting from version 8.6 of the CEVA-Toolbox Software Development Tools, it is possible to handle multiple memory areas for malloc. This is achieved by a new library function **init_malloc** that alters the memory area to be used by the **malloc** function. See the **init_malloc** function description for more details.

Example:

```
#include <stdlib.h>
void main( void )
{
        typedef struct {
                int num;
                float real;
        } pair;

        struct pair *ptr;
        ptr = malloc ( sizeof( struct pair) ); /* allocates 3 words */
}
```

See also: calloc, free, init_malloc

6.5.50. memchr

Syntax:

```
#include <string.h>

void * memchr ( void *s, char c, size_t n );
```

Description:

The **memchr** function finds the first occurrence of the character c in the first n characters of the object that s points to. If the character is found, the function returns a pointer to the character, otherwise it returns a null pointer.

Example:

```
#include <string.h>

char *s = "The CEVA-TeakLite is a fixed point DSP";
char *p = memchr ( s, "e", 20 ); /* p points to the 3rd letter of s[ ] */
```

See also: memcmp, strchr

6.5.51. memcmp

Syntax:

```
#include <string.h>

int memcmp ( void *s1, void *s2, size_t n );
```

Description:

The **memcmp** function compares the first n characters of the object that s2 points to with the object that s1 points to. If *s1<*s2, the function returns a negative value. If *s1=*s2, the function returns zero. If *s1>*s2, the function returns a positive value.

Example:

```
#include <string.h>

char *s1 = "The CEVA-TeakLite is a fixed point DSP";

char *s2 = "The CEVA-TeakLite is a 16 bits DSP";

int i = memcmp (s1, s2, 20)   /* i > 0 */
```

See also: strncmp

6.5.52. memcpy

Syntax:

```
#include <string.h>

void * memcpy ( void *s1, void *s2, size_t n );
```

Description:

The **memcpy** function copies the first n characters of the object that s2 points to into the object that s1 points to. If the s1 and s2 objects overlap, the functions behavior is undefined. The function returns the value of s1.

Example:

```
#include <string.h>

char *s1 = "The CEVA-TeakLite is a fixed point DSP";

char *s2 = "The CEVA- TeakLite is a 16 bits DSP";

s1 = memcpy  (s1, s2, 30);
```

See also: strncpy

6.5.53. memmove

Syntax:

```
#include <string.h>

void * memmove ( void *s1, void *s2, size_t n );
```

Description:

The **memmove** function moves the first n characters of the object that s2 points to into the object that s1 points to. The function returns the value of s1.

Example:

```
#include <string.h>

char *s1 = "The CEVA- TeakLite is a fixed point DSP";

char *s2 = "The CEVA- TeakLite is a 16 bits DSP";

s1 = memmove  (s1, s2, 30);
```

See also: memcpy

6.5.54. memset

Syntax:

```
#include <string.h>

 void * memset ( void *s, int k, size_t n );
```

Description:

The **memset** function copies the value of k into the first n characters of the object that s points to. The function returns the value of s.

Example:

```
#include <string.h>

char *s1 = "The CEVA- TeakLite is a fixed point DSP";

s1 = memset (s1, ' ', 3);

printf  ( "%s", s1 ) /* gives " CEVA- TeakLite is a fixed point DSP" */
```

See also: memcpy, memmove

6.5.55. modf

Syntax:

```
#include <math.h>

double modf ( double x, double *p );
```

Description:

The **modf** function breaks a floating point value, x, into a signed integer and a signed fraction, and returns the fractional part of x and stores the integer part as a double at the object pointed to by p. Both parts get the same sign as x.

Example:

```
#include <math.h>

void main( void )
{
        double x, y, z;

        x = 2.1345;
        z = modf ( x, &y ); /*y=2.0,z=0.1345*/
}
```

See also: modf, frexp

6.5.56. pause_clock

Syntax:

include <ceva-time.h>

static inline void pause_clock();

Description:

The **pause_clock** function pauses the clock profiler counter during debugger execution. Every call to **clock** following a call to **pause_clock** will return the clock value at time of pause.

Example:

```
#include <ceva-time.h>

void main( void )
 {
        clock_t c1,c2;
        c1 = clock();
        pause_clock();
        foo ();
        c2 = clock() ; // c2 == c1
 }
```

See also: start_clock, clock, reset_clock

6.5.57. pow

Syntax:

```
#include <math.h>

double pow ( double x, double y );
```

Description:

The **pow** function returns x raised to the power of y. A domain error occurs if x=0 and y<=0, or if x is negative and y is not an integer. A range error may occur if the result is too large.

Example:

```
#include <math.h>

void main( void )
{
        double x = 2.0, y = 4.0;
        double z = pow ( x, y )    /* z = 16.0 */
}
```

See also: exp, ldexp, log, log10

6.5.58. printf

Syntax:

```
#include <stdio.h>
int printf  ( const char *format, ... );
```

Description:

The **printf** function writes output to a predefined memory mapped I/O address in DSP processor data memory space (definable in crt0.c), under control of the string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. The function ends when the end of the format string is encountered, and returns the number of characters transmitted, or a negative value if an output error occurred. By "connecting" files to this memory mapped I/O address, the output can be "collected" and saved (effectively implementing a pseudo fprintf function). The printf function is included in the library for simulation purposes only. The format string is composed of zero or more directives, ordinary characters (which are copied unchanged to the output) and conversion specifications (introduced by the character %) each of which results in fetching zero or more subsequent arguments. After the %, the following appear in sequence:

Zero or more flags that modify the meaning of the conversion specification.

An optional minimum field width. Smaller values are left padded with spaces.

An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x and X conversions, the number of digits to appear after the decimal-point character for e, E and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in the s conversion.

An optional h specifying that a following d, i, o, u, x or X conversion specifier applies to a short int or unsigned short int argument, that a following n conversion specifier applies to a pointer to a short int argument ; an optional l specifying that a following d, i, o, u, x or X conversion specifier applies to a long int or unsigned long int argument, that a following n conversion specifier applies to a pointer to a long int argument ; or an

optional L specifying that a following e, E, f, g or G conversion specifier applies to a long double argument.

A character that specifies the type of conversion to be applied:
- left-justification within the field.

+ signed conversion will always begin with a plus or minus sign.

space if the first character of a signed conversion is not a sign, a space will be prefixed to the result.
alternate form.

0 for d, i, o,u, x, X, e, E, f, g and G conversions, leading zeros are used for padding (no space padding).

d, i the int argument is converted to signed decimal, style [-]dddd, default precision is 1 (leading zeros are used if the value can be represented with fewer digits than the specified precision).

o, u, x, X the unsigned int argument is converted to unsigned octal, unsigned decimal or unsigned hexadecimal, style dddd, default precision is 1 (leading zeros are used if value can be represented with fewer digits than the specified precision).

f the double argument is converted to decimal notation, using style [-]dddd.dddd, default precision is 6, values are rounded to the appropriate number of digits.

e, E the double argument is converted, style [-]d.ddde±dd, (or E) the number of digits after the decimal point is equal to the precision, which defaults to 6, values are rounded to the appropriate number of digits and the exponent contains always at least 2 digits.

g, G the double argument is converted, style f or e (or E in case of G) with the precision specifying the number of significant digits, depending on the value converted: style e (E) will be used only if the exponent is less than -4 or greater than or equal to the precision. c the int argument is converted to an unsigned char.

s the argument shall be a pointer to an array of character type, and the characters from the array are written up to (not including) the terminating null character.

p the argument shall be a pointer to void, and the value of the pointer is converted to a sequence of printable characters.

n the argument shall be a pointer to an integer into which is written the number of characters written to the output stream so far.

% a % is written (no conversion).

Example:

```
#include <stdio.h>

void main( void )
 {
          int num = 3;
         float real = 0.43;
         char s[10] = "Hello !";
         printf ( "Num=%d %s (%.3f)\n", num, s, real );
}
```

See also: iprintf, vprintf, ivprintf, putchar, puts, _puts, sprintf, isprintf, ivsprintf

6.5.59. putchar

Syntax:

```
#include <stdio.h>

int putchar ( int c );
```

Description:

The **putchar** function writes the character specified by c (converted to an unsigned char) to the output stream (i.e. to a predefined memory mapped I/O address in DSP processor data memory space, definable in crt0.c). The putchar function returns the character written and if a write error occurs, it returns EOF.

Example:

```
#include <stdio.h>

void main( void )
 {
        int c1 = 'F';
        putchar ( c1 );
 }
```

See also: printf, iprintf, vprintf, ivprintf, puts, _puts, sprintf, isprintf, ivsprintf

6.5.60. puts

Syntax:

```
#include <stdio.h>

int puts ( const char *s );
```

Description:

The **puts** function writes the character string pointed to by s to the output stream (i.e. to a predefined memory mapped I/O address in DSP processor data memory space, definable in crt0.c) and appends a new-line character to the output. The puts function returns EOF if a write error occurs, otherwise a non-negative value.

Example:

```
#include <stdio.h>

void main( void )
{
        char s1[15] = 'Hello World !";
        puts ( s1 );
}
```

See also: printf, iprintf, vprintf, ivprintf, putchar, _puts, sprintf, isprintf, ivsprintf

6.5.61. _puts

Syntax:

```
#include <stdio.h>

int _puts ( const char *s );
```

Description:

The **_puts** function writes the character string pointed to by s to the output stream (i.e. to a predefined memory mapped I/O address in DSP processor data memory space, definable in crt0.c). The puts function returns EOF if a write error occurs, otherwise a non-negative value.

Example:

```
#include <stdio.h>

void main( void )
{
        char s1[15] = 'Hello World !";
        _puts ( s1 );
}
```

See also: printf, iprintf, vprintf, ivprintf, putchar, puts, sprintf, isprintf, ivsprintf

6.5.62. rand

Syntax:

```
#include <stdlib.h>

int rand ( void );
```

Description:

The **rand** function returns a pseudo-random integer in the range [0, RAND_MAX]. It is affected by the function srand that sets the seed for the random number generator. If rand is called before srand, rand generates the same sequence it would produce if you first called srand with a seed value of 1. If srand is called with the same seed value, rand will generate the same sequence of random numbers.

Example:

```
#include <stdlib.h>

void main( void )
{
        int num;
        srand ( 34567 )          /* initialize seed */
        num = rand ()  /* create random number */
}
```

See also: srand

6.5.63. realloc

Syntax:

```
#include <stdlib.h>

void * realloc ( void *ptr, size_t size );
```

Description:

The **realloc** function changes the size of the allocated memory pointed to by ptr, to the size specified in words by size. The contents of the memory space remain unmodified. If ptr is 0, realloc is identical to malloc; if ptr points to unallocated memory, no action takes place; if the new space cannot be allocated the original memory remains unchanged.

Example:

```
#include <stdlib.h>

int * p;

realloc ( p, 10 ); /* allocate 10 words */
```

See also: calloc, malloc, free

6.5.64. reset_clock

Syntax:

include <ceva-time.h>

static inline void reset_clock();

Description:

The **reset_clock** function resets the clock profiler counter during debugger execution (sets its value to zero).

Example:

```
#include <ceva-time.h>

void main( void )
 {
       foo ();
       reset_clock(); // clock profiler counter is set to zero
 }
```

See also: start_clock, clock, pause_clock

6.5.65. scanf

Syntax:

```
#include <stdio.h>

int scanf ( const char *format, ... );
```

Description:

The **scanf** function scans input from the file designated by **stdin** under control of the argument format. The format string is described below. Following the format string is the list of addresses of items to receive values.

Returns:

The scanf function returns EOF when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

Example:

To scan a date in the form "Saturday April 18 1987":

```
#include <stdio.h>

void main()
{
        int day, year;
        char weekday[10], month[12];
        scanf ( "%s %s %d %d", weekday, month, &day, &year );
}
```

Format Control String:

The format control string consists of zero or more format directives that specify acceptable input file data. Subsequent arguments are pointers to various types of objects that are assigned values as the format string is processed.

A format directive can be a sequence of one or more white-space characters, an ordinary character, or a conversion specifier. An ordinary character in the format string is any character, other than a white-space character or the percent character (%), that is not part of a conversion specifier. A conversion specifier is a sequence of characters in the format string that begins with a percent character (%) and is followed, in sequence, by the following:

- an optional assignment suppression indicator: the asterisk character (*)
- an optional decimal integer that specifies the maximum field width to be scanned for the conversion
- an optional pointer-type specification: one of "N" or "F"
- an optional type length specification: one of "h", "l" or "L"
- a character that specifies the type of conversion to be performed: i.e., one of the characters cdefgionpsux[

As each format directive in the format string is processed, the directive may successfully complete, fail because of a lack of input data, or fail because of a matching error as defined by the particular directive. If end-of-file is encountered on the input data before any characters that match the current directive has been processed (other than leading white-space where permitted), the directive fails for lack of data. If an end-of-file occurs after a matching character has been processed, the directive is completed (unless a matching error occurs), and the function returns without processing the next directive. If a directive fails because of an input character mismatch, the character is left unread in the input stream. Trailing white-space characters, including new-line characters, are not read unless matched by a directive. When a format directive fails, or the end of the format string is encountered, the scanning is completed and the function returns.

When one or more white-space characters (space " ", horizontal tab "\t", vertical tab "\v", form feed "\f", carriage return "\r", new line or linefeed "\n") occur in the format string, input data up to the first non-white-space character is read, or until no more data remains. If no white-space characters are found in the input data, the scanning is complete and the function returns.

An ordinary character in the format string is expected to match the same character in the input stream.

A conversion specifier in the format string is processed as follows:

- For conversion types other than "[", "c" and "n", leading white-space characters are skipped
- For conversion types other than "n", all input characters, up to any specified maximum field length, that can be matched by the conversion type are read and converted to the appropriate type of value; the character immediately following the last character to be matched is left unread; if no characters are matched, the format directive fails.
- Unless the assignment suppression indicator ("*") was specified, the result of the conversion is assigned to the object pointed to by the next unused argument (if assignment suppression was specified, no argument is skipped); the arguments must correspond in number, type and order to the conversion specifiers in the format string

A type length specifier affects the conversion as follows:

- "h" causes a "d", "i", "o", "u" or "x" (integer) conversion to assign the converted value to an object of type short int or unsigned short int
- "h" causes an "f" conversion to assign a fixed-point number to an object of type long consisting of a 16-bit integer part and a 16-bit fractional part.
- "h" causes an "n" (read length assignment) operation to assign the number of characters that have been read to an object of type unsigned short int
- "l" causes a "d", "i", "o", "u" or "x" (integer) conversion to assign the converted value to an object of type long int or unsigned long int
- "l" causes an "n" (read length assignment) operation to assign the number of characters that have been read to an object of type unsigned long int
- "l" causes an "e", "f" or "g" (floating-point) conversion to assign the converted value to an object of type double
- "L" causes an "e", "f" or "g" (floating-point) conversion to assign the converted value to an object of type long double

The valid conversion type specifiers are:

c

Any sequence of characters in the input stream of the length specified by the field width, or a single character if no field width is specified, is matched. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence, without a terminating null character ('\0'). For a single character assignment, a pointer to a single object of type char is sufficient.

d

A decimal integer, consisting of an optional sign, followed by one or more decimal digits, is matched. The argument is assumed to point to an object of type int.

e, f, g

A floating-point number, consisting of an optional sign ("+" or "-"), followed by one or more decimal digits, optionally containing a decimal-point character, followed by an optional exponent of the form "e" or "E", an optional sign and one or more decimal digits, is matched. The exponent, if present, specifies the power of ten by which the decimal fraction is multiplied. The argument is assumed to point to an object of type float.

i

An optional sign, followed by an octal, decimal or hexadecimal constant is matched. An octal constant consists of "0" and zero or more octal digits. A decimal constant consists of a non-zero decimal digit and zero or more decimal digits. A hexadecimal constant consists of the characters "0x" or "0X" followed by one or more (upper- or lowercase) hexadecimal digits. The argument is assumed to point to an object of type int.

n

No input data is processed. Instead, the number of characters that have already been read is assigned to the object of type unsigned int that is pointed to by the argument. The number of items that have been scanned and assigned (the return value) is not affected by the "n" conversion type specifier.

o
An octal integer, consisting of an optional sign, followed by one or more (zero or non-zero) octal digits, is matched. The argument is assumed to point to an object of type int.

p
A hexadecimal integer, as described for "x" conversions below, is matched. The converted value is further converted to a value of type void* and then assigned to the object pointed to by the argument.

s
A sequence of non-white-space characters is matched. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating null character, which is added by the conversion operation.

u
An unsigned decimal integer, consisting of one or more decimal digits, is matched. The argument is assumed to point to an object of type unsigned int.

x
A hexadecimal integer, consisting of an optional sign, followed by an optional prefix "0x" or "0X", followed by one or more (upper- or lowercase) hexadecimal digits, is matched. The argument is assumed to point to an object of type int.

[c1c2...]
The longest, non-empty sequence of characters, consisting of any of the characters c1, c2, ... called the scanset, in any order, is matched. c1 cannot be the caret character ('^'). If c1 is "]", that character is considered to be part of the scanset and a second "]" is required to end the format directive. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating null character, which is added by the conversion operation.

[^c1c2...]

The longest, non-empty sequence of characters, consisting of any characters other than the characters between the "^" and "]", is matched. As with the preceding conversion, if c1 is "]", it is considered to be part of the scanset and a second "]" ends the format directive. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating null character, which is added by the conversion operation.

For example, the specification %[^\n] will match an entire input line up to but not including the newline character.

A conversion type specifier of "%" is treated as a single ordinary character that matches a single "%" character in the input data. A conversion type specifier other than those listed above causes scanning to terminate the function to return.

The line:

```
scanf ( "%s%*f%3hx%d", name, &hexnum, &decnum )
```

with input:

```
some_string 34.555e-3 abc1234
```

will copy "some_string" into the array name, skip 34.555e-3, assign 0xabc to hexnum and 1234 to decnum. The return value will be 3.

The program:

```
#include <stdio.h>

void main()
  {
    char string1[80], string2[80];

    scanf ( "%[abcdefghijklmnopqrstuvwxyz"
          "ABCDEFGHIJKLMNOPQRSTUVWZ ]%*2s%[^\n]",
          string1, string2 );
    printf ( "%s\n%s\n", string1, string2 );
  }
```

with input:

```
They may look alike, but they do not perform alike.
```

will assign

"They may look alike"

to string1, skip the comma (the "%*2s" will match only the comma; the following

blank terminates that field), and assign
" but they do not perform alike."
to string2.

printf, iprintf, vprintf, ivprintf, putchar, puts, sprintf, isprintf, ivsprintf

6.5.66. setjump

Syntax:

```
#include <setjmp.h>
jmp_buf env;
int setjmp (jmp_buf env);
```

Description:

setjmp() and **longjmp** () are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. The setjmp() routine saves its stack environment in its env parameter for later use by longjmp (). A normal call to setjmp() returns zero. setjmp() also saves the register environment (all the callee saved registers, see chapter Optimizations). If a longjmp() call will be made, the routine which called setjmp() should not return until after the longjmp () has returned control (see example).

Example:

```
jmp_buf my_env;
...
if ( !setjmp( my_env ) )
 {
        /* this is the usual path (return from setjmp) */
        ...
        my_proc();
 }
else
 {
        /* error occured:
        code after return from longjmp, (stack and register environment restored)
*/
 }
void my_proc( )
 {
        if( error_condition )
        longjmp ( my_env, 2 ); /* return non-zero value to restore environment */
        ...
 }
```

See also: longjmp

6.5.67. sin

Syntax:

```
#include <math.h>

double sin ( double x );
```

Description:

The **sin** function returns the sine value of a floating point argument. The argument must be expressed in radians, the result will be in the range [-1.0,1.0].

Example:

```
#include <math.h>

void main( void )
{
        double x = 0.5 * 3.1415927;
        double y = sin ( x );/* y = 1.0 */
}
```

See also: asin, Q14_sin_Q12, Q14_sini_Q0

6.5.68. sinh

Syntax:

```
#include <math.h>

double sinh ( double x );
```

Description:

The **sinh** function returns the hyperbolic sine value of a floating point argument. A range error occurs if the magnitude of the argument is too large.

Example:

```
#include <math.h>

void main( void )

{
        double x = 0.0;
        double y = sinh ( x ); /* y = 0.0 */
}
```

See also: cosh, sin

6.5.69. sprintf

Syntax:

```
#include <stdio.h>

int sprintf ( char *s, const char *format, ... );
```

Description:

The **sprintf** function is equivalent to the printf function, except that it writes the output to an array pointed to by s rather than the standard output stream. A null character is written at the end of the characters written to the array. See the description of the printf function for details on the format string and how it controls the output. The sprintf function returns the number of characters written to the array, not counting the terminating null character.

Example:

```
#include <stdio.h>

void main( void )
{
        float r1 = 0.43;
        char buf[30];
        char s[10] = "Hello !";
        sprintf ( buf, "%s result is:%.3f\n", s, r1 );
}
```

See also: printf, iprintf, vprintf, ivprintf, putchar, puts, _puts, isprintf, ivsprintf

6.5.70. sscanf

Syntax:

```
#include <stdio.h>

int sscanf ( const char *s, const char *format, ... );
```

Description:

The **sscanf** function is equivalent to the fscanf function, except that it scans the input from an array pointed to by s rather than the file designated by **stdin.**

Reaching the end of the string is equivalent to encountering end-of-file for the fscanf function. See the description of the scanf function for details on the format string and how it controls the output. The **sscanf** function returns the value of the macro EOF if an input failure occurs. Otherwise, the **sscanf** function returnsthe number of the input items assigned.

Example:

To scan a date in the form "Saturday April 18 1987":

```
#include <stdio.h>

char *date = "Saturday April 18 1987";
void main( void )
{
        int day, year;
        char weekday[10], month[12] ;

        sscanf ( date, "%s %s %d %d", weekday, month, &day, &year );
}
```

See also: scanf, getchar, gets

6.5.71. vprintf

Syntax:

```
#include <stdarg.h>
#include <stdio.h>

int vprintf ( const char *format, va_list arg );
```

Description:

The **vprintf** function is equivalent to printf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro. The **vprintf** function does not invoke the va_end macro.

Example:

```
#include <stdarg.h>
#include <stdio.h>

void error( char *function_name, char *format, … )
{
        va_list args;
        va_start ( args, format );

        /* print out name of function causing error */
        printf ( "Error in %s: “, function_name );

        /* print out remainder of message */
        vprintf ( “format, args” );

        va_end ( args );
}
```

See also: printf, iprintf, vsprintf, putchar, puts, _puts, sprintf, isprintf, ivsprintf

6.5.72. vsprintf

Syntax:

```
#include <stdarg.h>
#include <stdio.h>

int vsprintf ( char *s, const char *format, va_list arg );
```

Description:

The **vsprintf** function is equivalent to sprintf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro. The **vsprintf** function does not invoke the va_end macro.

Example:

```
#include <stdarg.h>
#include <stdio.h>

char *error( char *function_name, char *format, … )
{
        va_list args;
        va_start ( args, format );
        char error_message[80];

        /* print out name of function causing error */
        sprintf ( error_message, "Error in %s: “, function_name );

        /* print out remainder of message */
        vsprintf ( error_message, “format, args” );

        va_end ( args );
}
```

See also: printf, iprintf, vprintf, putchar, puts, _puts, sprintf, isprintf, ivsprintf

6.5.73. sqrt

Syntax:

```
#include <math.h>

double sqrt ( double x );
```

Description:

The **sqrt** function returns the non-negative square root value of a floating point argument.

A domain error occurs, if the argument is negative.

Example:

```
#include <math.h>

void main( void )
 {
        double x = 9.0;
        double y = sqrt ( x ); /* y = 3.0 */
 }
```

See also: exp, log, pow, Q7_sqrt_Q0

6.5.74. srand

Syntax:

```
#include <stdlib.h>

void srand ( unsigned int seed );
```

Description:

The **srand** function sets the seed for the random number generator, so that subsequent calls to the rand function produce a new sequence of random numbers. The srand function does not return a value.

Example:

```
#include <stdlib.h>


void main( void )
{
        int num;
        srand ( 34567 ); /* initialize seed */
        num = rand(); /* create random number */
}
```

See also: rand

6.5.75. start_clock

Syntax:

include <ceva-time.h>

static inline void start_clock();

Description:

The **start_clock** function enables updates of the clock profiler counter during debugger execution (exits pause state of the counter).

The Compiler starts the clock counter by default on startup (see **initConfigRegs** in section crt0.c - Startup Code).

Example:

```
#include <ceva-time.h>

void main( void )
 {
        clock_t c1,c2;
        c1 = clock();
        pause_clock();
        foo ();
        start_clock() ;
        c2 = clock(); // execution cycles of foo are excluded from measurements
 }
```

See also: reset_clock, clock, pause_clock

6.5.76. strcat

Syntax:

```
#include <string.h>

char * strcat ( char *s1, const char *s2 );
```

Description:

The **strcat** function appends a copy of s2 (including the terminating null character) to the end of s1. The initial character of s2 overwrites the null character that originally terminated s1. The strcat function returns the value of s1.

Example:

```
#include <string.h>

void main( void )
 {
        char a[10] = "He loves ";
        char b[5] = "her";
        char *c = strcat ( a, b ); /* c[] = "He loves her" */
 }
```

See also: strcpy, strncat

6.5.77. strchr

Syntax:

#include <string.h>

char * strchr (const char *s, int c);

Description:

The **strchr** function finds the first occurrence of c in s. If strchr finds the character, it returns a pointer to the character, otherwise it returns a null pointer.

Example:

```
#include <string.h>

void main( void )
{
        char a[20] = "Hello World ";
        char *c = strchr ( a, 'W' ); /* c points to "World" */
}
```

See also: strcmp, strcspn, strrchr, strspn, strstr

6.5.78. strcmp

Syntax:

```
#include <string.h>

int strcmp ( const char *s1, const char *s2 );
```

Description:

The **strcmp** function compares s2 with s1. The function returns a negative value if *s1 < *s2, zero if *s1 = *s2, and a positive value if *s1 > *s2.

Example:

```
#include <string.h>

void main( void )
{
        char a[10] = "hello";
        char b[10] = "there";
        int k = strcmp ( a, b ); /* k<0 */
}
```

See also: strcoll, strncmp, strspn

6.5.79. strcoll

Syntax:

```
#include <string.h>

int strcoll ( const char *s1, const char *s2 );
```

Description:

The **strcoll** function is equivalent to the strcmp function above.

Example:

```
#include <string.h>

void main( void )
{
        char a[10] = "Hello";
        char b[10] = "Hello";
        int k = strcoll ( a, b )   /* k=0 */
}
```

See also: strcmp, strncmp, strspn

6.5.80. strcpy

Syntax:

```
#include <string.h>

char * strcpy ( char *s1, const char *s2 );
```

Description:

The **strcpy** function copies s2 (including the terminating null character) into s1. If the two

strings overlap, the behavior is undefined. The function returns a pointer to s1.

Example:

```
#include <string.h>
void main( void )
{
  char a[10] = "hello";
  char b[10] = "there";
  char *c = strcpy ( a, b ); /* c[] = "there" */
}
```

See also: strcat, strncpy

6.5.81. strcspn

Syntax:

```
#include <string.h>

size_t strcspn ( const char *s1, const char *s2 );
```

Description:

The **strcspn** function returns the length of the initial segment of s1 which is made up entirely of characters that are not in s2. If the first character in s1 is in s2, the function returns 0.

Example:

```
#include <string.h>

void main( void )
{
        char a[10] = "Hi There";
        char b[10] = "abcde";
        size_t i = strcspn ( a, b ); /* i=5 */
}
```

See also: strrchr, strspn, strstr

6.5.82. strlen

Syntax:

```
#include <string.h>

size_t strlen ( const char *s );
```

Description:

The **strlen** function returns the length of the string pointed to by s (not including the terminating null character).

Example:

```
#include <string.h>
void main( void )
{
        char a[10] = "hello";
        size_t i = strlen ( a ); /* i = 5 */
}
```

See also: strspn

6.5.83. strncat

Syntax:

```
#include <string.h>

char * strncat ( char *s1, const char *s2, size_t n );
```

Description:

The **strncat** function appends up to n characters of s2 (including a terminating null character) to the end of s1. The first character of s2 overwrites the null character that originally terminated s1. the function returns the value of s1.

Example:

```
#include <string.h>

void main( void )
{
        char a[10] = "Hello ";
        char b[10] = "World !!!";
        char *c = strncat ( a, b, 5); /* c[]="Hello World" */
 }
```

See also: strcat

6.5.84. strncmp

Syntax:

```
#include <string.h>

int strncmp ( const char *s1, const char *s2, size_t n );
```

Description:

The **strncmp** function compares up to n characters of s2 with s1. The function returns a negative value if *s1 < *s2, zero if *s1 = *s2, and a positive value if *s1 > *s2.

Example:

```
#include <string.h>

char a[10] = "hello";

char b[10] = "there";

int k = strncmp ( a, b, 3 ); /* k<0 */
```

See also: strcoll, strcmp

6.5.85. strncpy

Syntax:

```
#include <string.h>

char * strncpy ( char *s1, const char *s2, size_t n );
```

Description:

The **strncpy** function copies up to n characters from s2 into s1. If characters from overlapping strings are copied, the behavior is undefined. If s2 is n characters long or longer, the null character that terminates s2 is not copied. If s2 is shorter than n characters, strncpy appends null characters to s1 so that s1 contains n characters. The function returns a pointer to s1.

Example:

```
#include <string.h>

char a[10] = "hello";

char b[10] = "there";

char *c = strncpy ( a, b, 2 );/* c[] = "thllo" */
```

See also: strcat, strncpy

6.5.86. strpbrk

Syntax:

```
#include <string.h>

char * strpbrk ( const char *s1, const char *s2 );
```

Description:

The **strpbrk** function locates the first occurrence in s1 of any character in s2. If strpbrk finds a matching character, it returns a pointer to that character; otherwise the function returns a null pointer.

Example:

```
#include <string.h>

void main( void )
{
        char a[15] = "hello world";
        char b[10] = "dawn";
        char *c = strpbrk ( a, b ); /* c points to "w" in "world" */
}
```

See also: strchr, strcmp

6.5.87. strrchr

Syntax:

```
#include <string.h>

char * strrchr ( const char *s, int c );
```

Description:

The **strrchr** function finds the last occurrence of c in s. If strrchr finds the character, it returns a pointer to the character, otherwise it returns a null pointer.

Example:

```
#include <string.h>

void main( void )
{
        char a[20] = "Hello World ";
        char *c = strrchr ( a, 'o' ); /* c points to "o" in "World" */
}
```

See also: strchr, strcmp, strcspn, strstr

6.5.88. strspn

Syntax:

```
#include <string.h>

size_t strspn ( const char *s1, const char *s2 );
```

Description:

The **strspn** function returns the length of the maximum initial segment of s1 which consists entirely of characters in s2. If the first character in s1 is in s2, the function returns 0.

Example:

```
#include <string.h>

void main( void )
{
        char a[10] = "abcdhfghi";
        char b[10] = "abcdef";
        size_t i = strspn ( a, b )   /* i=4 */
}
```

See also: strrchr, strcspn, strstr

6.5.89. strstr

Syntax:

```
#include <string.h>

char * strstr ( const char *s1, const char *s2 );
```

Description:

The **strstr** function finds the first occurrence of s2 in s1 (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string, otherwise it returns a null pointer. If s2 points to a string with zero length, the function returns s2.

Example:

```
#include <string.h>

char a[20] = "Hello World";

char b[10] = "Wo";

char *c = strstr ( a, b ); /* c points to "World" */
```

See also: strchr, strcmp, strrchr, strspn

6.5.90. strtod

Syntax:

```
#include <stdlib.h>

double strtod ( const char *nptr, char **endptr );
```

Description:

The **strtod** function converts a character string, pointed to by nptr, to a floating point number and returns the value of that number. The argument endptr is a pointer to a pointer that points to the first character after the converted string. The string must have the following format:

[space] [sign] digits [.digits] [e|E [sign] integer]

where space can be one or more white space characters (space, horizontal tab, vertical tab, carriage return, formfeed, newline). If the original string is empty or does not have the correct format, the function returns 0. If the converted string is too large, the function returns 1e500, if too small it returns 0.

Example:

```
#include <stdlib.h>

double x;

x = strtod ( " -12.57e-1, NULL" ); /* x = -1.257 */
```

See also: atof

6.5.91. strtol

Syntax:

```
#include <stdlib.h>

long int strtol ( const char *nptr, char **endptr, int base );
```

Description:

The **strtol** function converts a character string, pointed to by **nptr**, to a long integer and returns the value of that number. The argument endptr is a pointer to a pointer that points to the first character after the converted string. The string must have the following format:

[space] [sign] digits [.digits] [e|E [sign] integer]

where space can be one or more white space characters (space, horizontal tab, vertical tab, carriage return, formfeed, newline). If the original string is empty or does not have the correct format, the function returns 0.

Example:

```
#include <stdlib.h>

void main( void )
{
        long x;
        x = strtol ( "1.257e4", NULL, 10 ); /* x = 12570*/
}
```

See also: atol

6.5.92. strtoul

Syntax:

```
#include <stdlib.h>

unsigned long int strtoul ( const char *nptr, char **endptr, int base );
```

Description:

The **strtoul** function converts a character string, pointed to by **nptr**, to an unsigned long integer and returns the value of that number. The argument **endptr** is a pointer to a pointer that points to the first character after the converted string. The argument **base** is the number base used. The string must have the following format:

[space] [sign] digits [.digits] [e|E [sign] integer]

where space can be one or more white space characters (space, horizontal tab, vertical tab, carriage return, formfeed, newline). If the original string is empty or does not have the correct format, the function returns 0.

Example:

```
#include <stdlib.h>

void main( void )
{
        unsigned long x;
        x = strtoul ( " 1.257e4, NULL, 10" ); /* x = 12570 */
}
```

See also: atol

6.5.93. tan

Syntax:

```
#include <math.h>

double tan ( double x );
```

Description:

The **tan** function returns the tangent of a floating point argument. The argument must be expressed in radians. A too large argument produces a non significant result.

Example:

```
#include <math.h>

void main( void )
{
        double x = 0.25 * 3.1415927;
        double y = tan ( x ); /* y = 1.0 */
}
```

See also: atan, cos, sin

6.5.94. tanh

Syntax:

```
#include <math.h>

double tanh ( double x );
```

Description:

The **tanh** function returns the hyperbolic tangent value of a floating point argument.

Example:

```
#include <math.h>

void main( void )
{
        double x = 0.0;
        double y = tanh ( x ); /* y = 0.0 */
}
```

See also: cosh, sinh, tan

6.5.95. tolower

Syntax:

```
#include <ctype.h>

int tolower( char c );
```

Description:

The **tolower** function or macro converts (and returns) the argument character to lower case if it is upper case, otherwise it returns the argument unchanged.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = 'H';
        int k = tolower ( c ); /* k = 'h' */
}
```

See also: isalpha, toupper

6.5.96. toupper

Syntax:

```
#include <ctype.h>

int toupper ( char c );
```

Description:

The **toupper** function or macro converts (and returns) the argument character to upper case if it is lower case, otherwise it returns the argument unchanged.

Example:

```
#include <ctype.h>

void main( void )
{
        char c = 'p';
        int k = toupper ( c ); /* k = 'P' */
}
```

See also: isalpha, tolower

6.5.97. va_arg

Syntax:

```
#include <stdarg.h>

type va_arg ( va_list ap, type );
```

Description:

The **va_arg** macro expands to an expression that has the type and value of the next argument in a call to a variable argument function. Each time va_arg is invoked, ap is modified so that the values of successive arguments are returned. The type parameter is a type name, corresponding to the type of the current argument in the list. The first invocation of the function va_arg must be made after a call to the va_start macro. The first invocation of the va_arg macro after that of the va_start macro, returns the value of the argument after that specified by parmN. Successive invocations return the values of the remaining arguments in succession.

Example:

```
#include <stdarg.h>

int printf ( char *format, ...)
{
        va_list ap;
        va_start ( ap, format );
        ...
        i = va_arg ( ap, int );
        s = va_arg ( ap, char *);
        ...
        va_end ( ap );
}
```

See also: va_start, va_end

6.5.98. va_end

Syntax:

```
#include <stdarg.h>

void va_end ( va_list ap );
```

Description:

The **va_end** macro resets the stack environment after va_start abd va_arg are used, enabling a normal return from a function whose variable argument list was referred to by va_start that initialized the va_list ap. The function returns no value.

Example:

See the example of va_arg above.

See also: va_arg, va_start

6.5.99. va_start

Syntax:

```
#include <stdarg.h>

void va_end ( va_list ap, paramN );
```

Description:

The **va_start** macro must be invoked before any access to the unnamed arguments. The va_start macro initializes a list of variable arguments, **ap**, for subsequent use by va_arg and va_end. The parameter paramN is the rightmost parameter in the fixed declared list (the one just before the ,...). The macro returns no value.

Example:

see the example of va_arg above.

See also: va_arg, va_end

7. Mixing Assembly and C Routines

As described in the following chapter, it is possible to optimize the generated code of the Compiler by in-lining assembly instructions in the C code using the **__asm__** C language extension macro. Furthermore, the C application is able to call exiting assembly routines and it is also possible to call C functions from assembly routines. In all cases of mixing C and assembly code, it is vital to comply with the Compiler's assumptions and its function calling convention.

This chapter includes the following sections:

- **Complying with Compiler assumptions and calling conventions when implementing assembly code**
- **Calling C/C++ Functions From Assembly Routines**
- **CEVA-XM4 Register Usage Convention**

7.1 Complying with Compiler Assumptions and Calling Conventions when Implementing Assembly Code

When implementing assembly code which is to be called from C code, it is vital to follow the list of Compiler assumptions as well as its function calling convention and stack layout. Non compliant assembly code is likely to cause run-time malfunctions as it likely to disrupt the execution of Compiler generated code.

Although not mandatory, it is highly recommended to follow these guidelines also when writing assembly code, which is not supposed to be called from C code. This is important in order to make the code C callable for future usage and it also makes programmers more used to comply with these guidelines and decreases the chances of writing non compliant assembly code, when compliance is mandatory.

7.2 Calling C/C++ Functions from Assembly Routines

Calling C functions from assembly routines must be done using the default calling convention as described in this chapter. In addition, the proper status bits settings are required for proper execution of C code (i.e. Saturation disabled, Post-Shift disabled, etc. (Refer to the *C/C++ Language Environment Assumptions* for more information):

Example:

```
.CSECT CODE4
;
.EXTERN _c_func2
.PUBLIC yoyo4
yoyo4:
;
; calls c_func2
; prototype: extern int c_func2(int *, int, long, int, int)
;
SC0.mov r3, r0  ;;      ; 1st parameter in r0
SC0.mov #1, r1  ;;      ; 2nd parameter in r1
SC0.mov #2, r2  ;;      ; 3rd long parameter in r2
SC0.mov #3, r3  ;;      ; 4th parameter in r3
SC0.mov #4, r4  ;;      ; 5th parameter in r4
PCU.call {t}  INCODE _c_func2  ;;   ; Compiler naming convention requires a
"_"
```

7.3 CEVA-XM4 Register Usage Convention

Note: ***For more information about call-used, call-saved and fixed registers refer to Register Types and Usage.***

Note: ***CEVA-XM4****'s registers that are not mentioned in the following table are not used or accessed by the Compiler generated code (except for inline assembly instructions).CEVA-XM4 VPU vector-type registers are detailed in a separate Table 7-2.*

Table 7-1 :CEVA-XM4 Register Usage Convention

	Register Name	Fixed	Call used (Scratch)	Call Saved
1.	r0	no	yes	no
2.	r1	no	yes	no
3.	r2	no	yes	no
4.	r3	no	yes	no
5.	r4	no	yes	no
6.	r5	no	yes	no
7.	r6	no	yes	no
8.	r7	no	yes	no
9.	r8	yes	no	yes
10.	r9	no	no	yes
11.	r10	no	no	yes
12.	r11	no	no	yes
13.	r12	no	no	yes
14.	r13	no	no	yes
15.	r14	no	no	yes
16.	r15	no	no	yes
17.	r16	no	no	yes
18.	r17	no	no	yes
19.	r18	no	no	yes
20.	r19	no	no	yes
21.	r20	no	no	yes
22.	r21	no	no	yes
23.	r22	no	no	yes
24.	r23	no	no	yes

	Register Name	Fixed	Call used (Scratch)	Call Saved
25.	r24	no	yes	no
26.	r25	no	yes	no
27.	r26	no	yes	no
28.	r27	no	yes	no
29.	r28	no	yes	no
30.	r29	no	yes	no
31.	r30	no	yes	no
32.	r31	no	yes	no
33.	modu0	no	yes	no
34.	modu1	no	yes	no
35.	modu2	no	yes	no
36.	modu3	no	yes	no
37.	s0	no	no	yes
38.	s1	no	no	yes
39.	s2	no	no	yes
40.	s3	no	no	yes
41.	s4	no	no	yes
42.	s5	no	no	yes
43.	s6	no	no	yes
44.	s7	no	no	yes
45.	vpr0	no	yes	no
46.	vpr1	no	yes	no
47.	vpr3	no	yes	no
48.	vpr3	no	yes	no
49.	vpr4	no	yes	no
50.	vpr5	no	yes	no
51.	vpr6	no	yes	no
52.	bknest0	yes	no	no
53.	bknest1	yes	no	no
54.	bknest2	yes	no	no
55.	retreg	no	no	yes
56.	retregi	yes	no	no

	Register Name	Fixed	Call used (Scratch)	Call Saved
57.	retregn	yes	no	no
58.	retregb	yes	no	no
59.	LP	yes	no	no
60.	BCx	yes	no	no
61.	modA	yes	no	no
62.	modB	yes	no	no
63.	modC	yes	no	no
64.	modD	yes	no	no
65.	modE	no	no	yes
66.	modF	yes	no	no
67.	lc0	no	yes	no
68.	lc1	no	yes	no
69.	lc2	no	yes	no
70.	lc3	no	yes	no
71.	lcstep0	no	yes	no
72.	lcstep1	no	yes	no
73.	pr0	no	no	yes
74.	pr1	no	no	yes
75.	pr2	no	no	yes
76.	pr3	no	no	yes
77.	pr4	no	no	yes
78.	pr5	no	no	yes
79.	pr6	no	no	yes
80.	pr7	no	no	yes
81.	pr8	yes	no	yes
82.	pr9	no	no	yes
83.	pr10	no	no	yes
84.	pr11	no	no	yes
85.	pr12	no	no	yes
86.	pr13	no	no	yes
87.	pr14	no	no	yes
88.				

	Register Name	Fixed	Call used (Scratch)	Call Saved
89.				

Table 7-2: CEVA-XM4 VPU Register Usage Convention

	Register Name	Fixed	Call used (Scratch)	Call Saved
1.	v0	no	yes	no
2.	v1	no	yes	no
3.	v2	no	yes	no
4.	v3	no	yes	no
5.	v4	no	yes	no
6.	v5	no	yes	no
7.	v7	no	yes	no
8.	v8	no	yes	no
9.	v9	no	yes	no
10.	v10	no	yes	no
11.	v11	no	yes	no
12.	v12	no	yes	no
13.	v13	no	yes	no
14.	v14	no	yes	no
15.	v15	no	yes	no
16.	v16	no	yes	no
17.	v17	no	yes	no
18.	v18	no	yes	no
19.	v19	no	yes	no
20.	v20	no	yes	no
21.	v21	no	yes	no
22.	v22	no	yes	no
23.	v23	no	yes	no
24.	v24	no	yes	no
25.	v25	no	yes	no
26.	v26	no	yes	no
27.	v27	no	yes	no
28.	v28	no	yes	no
29.	v29	no	yes	no
30.	v30	no	yes	no
31.	v31	no	yes	no

	Register Name	Fixed	Call used (Scratch)	Call Saved
32.	v32	no	yes	no
33.	v33	no	yes	no
34.	v34	no	yes	no
35.	v35	no	yes	no
36.	v36	no	yes	no
37.	v37	no	yes	no
38.	v38	no	yes	no
39.	v39	no	yes	No

8. Programming Hints

This chapter includes various programming guidelines along with some examples demonstrating the usage of C code resulting in compact, efficient cores assembly code.

The following chapters are covered:

- **Variable "Live" Scope Usage Minimization**
- **Non-Address Taken Local Variables**
- **Array Index Variable Type**
- **ETSI/ITU Basic-ops (Intrinsics)**
- **Keep it Samll and Simple**
- **Mix of -O3 / -O3 –Os**
- **Variables Type Usage for Multiplication**
- **Minimize Loops Contents**
- **Set Well Known/Simple Limits to Loops – 1**
- **Set Well Known/Simple Limits to Loops – 2**
- **Overcome Memory Aliasing**
- **Pointers and Local Variables Usage Minimization**
- **Efficient Loops**
- **Smart Usage of Pointers**

8.1 Variable "Live" Scope Usage Minimization

Try to minimize the "live" variables scope.

This can be achieved by using C blocks. It may help the Compiler in the task of allocating registers especially around small loops or other critical performance sections.

Example 1:

```
int i;
long acc;
acc = 0;
for (i = 0; i < 10; i++)
        acc += ...
acc = 0;
for (i = 0; i < 10; i++)
        acc += ...
```

If acc is not assigned to a register in the 1st program, (example 1) try the 2nd program (example 2).

Example 2:

```
int i;
long acc;
{
  long acc1 = 0;
  for (i = 0; i < 10; i++)
  acc1 += ...
  acc = acc1;
}
{
  long acc1 = 0;
  for (i = 0; i < 10; i++)
    acc1 += ...
   acc = acc1;
 }
```

8.3 Non-Address Taken Local Variables

Use local, not static variables, whose addresses are not referred (&variable) as much as possible. In addition, avoid using global, static variables, or address-taken variables for computation intensive code sequences.

The reason behind is to let the Compiler assign local variables to registers as much as possible. The Compiler does not allocate non-automatic (static)/address-taken ("C" sense) variables into registers. If an address of an automatic local variable must be taken then it can be avoided by introducing another automatic variable.

Example:

Instead of:

```
int i;
for (i=0; i<10; i++) ...
p(&i);
for (i=0; i<10; i++) ...
```

write:

```
int i, j;
for (i=0; i<10; i++) ...
j = i;
p(&j);
i = j;
for (i=0; i<10; i++) ...
```

This way the Compiler will be able to allocate the variable **i** into a register.

8.4 Array Index Variable Type

Apply ‘int’ type instead of ‘short’ type to variables that are used as array indexes (especially in loops)

- Loop counters commonly used as indexes into an arrays in the loop body
- As address computation is done in 32 bits – 16 bits index variables require extra instructions for each index computation in order to perform sign extension and to ensure 16 bits overflow when incrementing the loop counter

Example:

Instead of:

```
short i;

for ( i = 0;  i < 100; i++ )
        array[i] = 2;
```

write:

```
int i;

for ( i = 0;  i < 100; i++ )
        array[i] = 2;
```

8.5 Keep It Small and Simple

Large functions might cause the register allocation phase to spill variables onto the stack. Keeping the functions small and simple will reduce the likelihood of spilling variables.

8.6 Mix of -OX / -OsX Optimization Levels

Apply the most appropriate optimization strategy to each function/file to achieve optimal cycle count and code size.

(can be done at the file level only - functions can be separated into to different files)

- Use -O3, -O4 for critical code (kernels) in order to get best performance
- Use -O3 -Os for non critical code in order to get compact code

8.7 Variables Type Usage for Multiplication

32-bit multiplication is not supported natively by the Compiler (using a single instruction) and requires a sequence of instructions or even a library function call. When multiplying a 16-bit variable with another 16-bit variable type casting should be used (i.e. short)

8.8 Minimize Loops Content

Try to minimize the code size inside the loop. Code that does not depend on the loop index can and should be placed out of the loop. Memory accesses can be done outside of the loop and be assigned to local variables in order to save ld/st instructions.

Example:

Instead of:

```
for ( i=0; i < size1; i++ )
{
   arr1[i] = 0;
   arr2[i] = 0;

   for ( j=0;  j < size2 ; j++ )
   {
      arr1[i] = arr1[i] + arr3[j] + arr4[i+j] ;
      arr2[i] = arr2[i] + arr3[j] + arr4[i+j] ;
   }

   ... // use of arr1[i] and arr2[i]
}
```

Write:

```
int a, b;

for ( i=0; I < size1; i++ )
{
   a = arr1[i] = 0;
   b = arr2[i] = 0;

   for ( j=0; j < size2 ; j++)
   {
      a = a + arr3[j] + arr4[i+j];
      b = b + arr3[j] + arr4[i+j];
   }

   ... // use of a and b
   arr1[i] = a;
   arr2[i] = b;
}
```

8.9 Set Well Known/Simple Limits to Loops – 1

This helps the Compiler in generating “bkrep” instructions.

Example:

Instead of:

```
while ((*p != 0) && (i < 200))
{
        …
        i++;
}
```

write:

```
while (i < 200)
{
        if (*p == 0)
            break;
        …
        i++;
}
```

8.10 Set Well Known/Simple Limits to Loops – 2

If 'foo' has no dependency in the loop code, it's better to call 'foo' before the loop in order to:

- Help the Compiler in applying a "bkrep"
- Save a call to the function in each iteration

Example:
Instead of:

```
for ( n = 0; n < foo(); n++ )
{
…
}
```

write:

```
iter = foo();

for ( n = 0; n < iter; n++ )
{
…
}
```

8.11 Overcome Memory Aliasing

In some cases, the user cannot minimize the memory operations in the loop since the pointers are variant in the loop. Memory aliasing reduces the Compiler's capability for better scheduling inside the loop. Use memory aliasing switches in order to relax aliasing and allow better instruction scheduling.

Example:

```
void foo (int *p, int *q)
{
    int i;
    for ( i=0; i < 100; i++ )
    {
        *p++ = *q++;
    }
 }
```

Without aliasing switches:

```
        PCU.bkrep #24
        {
                LS0.ld  (r3.ui).i+#4, r1.i
                nop
                LS0.st r1.i, (r7.ui).i+#4
                LS0.ld (r3.ui).i+#4, r2.i
                nop
                LS0.st   r2.i, (r7.ui).i+#4
                LS0.ld   (r3.i).ui+#4, r1.i
                nop
                LS0.st {dw}  r1.i, (r7.ui).i+#4
                LS0.ld {dw}  (r3.ui).i+#4, r0.i
                nop
                LS0.st {dw}  r0.i, (r7.ui).i+#4
        }
```

12 cycles

With aliasing switches:

```
…
PCU.bkrep #23
        {
                LS0.ld (r3.ui).i+#4, r3.i
        ||      LS1.st r2.i, (r7.ui).i+#4
                LS0.ld (r3.ui).i+#4, r0.i
        ||      LS1.st r1.i, (r7.ui).i+#4
                LS0.ld (r3.ui).i+#4, r2.i
        ||      LS1.st r3.i, (r7.ui).i+#4
                LS0.ld (r3.ui).i+#4, r1.i
        ||      LS1.st r0.i, (r7.ui).i+#4
        }
…
```

4 cycles

* in this example –OPT:alias=restrict switch was used, please refer to section 3.1.8 for more alias options.

8.12 Pointers and Local Variables Usage Minimization

Try to simplify the compiler's task in register allocation by keeping as few as possible pointers and local variables within any particular scope.

In the following example, the Compiler is likely to execute better register allocation by using one pointer (p) instead of 2 pointers (p, q), for performing the same job.

Instead of:

```
long i;
char *p, *q;
...
for (i = 0; i < 10; i++)
      *p = 0;
...
for (i = 0; i < 10; i++)
      *q = 0;
...
for (i = 0; i < 10; i++)
      *p = 0;
```

write:

```
long i;
char *p;
...
for (i = 0; i < 10; i++)
      *p = 0;
...
for (i = 0; i < 10; i++)
      *p = 0;
...
for (i = 0; i < 10; i++)
       *p = 0;
```

8.13 Efficient Loops

Try to use

do {...} while(..)
loops

instead of
for(..){...}

or
while(..) {...}
loops.

The reason is, that for the last 2 constructs, the C language requires the Compiler to generate code to check whether the loop body should be entered at all. This is not relevant for loops in which the number of iterations is known in advance, since in that case, the Compiler optimizes out the extra compare & branch.

Example:

```
int i;
extern int j;
for (i = 0; i < j; i++)
        *p++ = 0;
```

requires the Compiler to generate the following assembly code:

```
 ...
 cmp #0x0,r0          ; check if the loop should be entered
 brr L1, le
L0:
 ...                  ; loop body
 cmp r0,r1
 brr L0, lt
L1:
```

Rewriting the C code in the following way would cause the removal of the 1st compare & branch:

```
i = 0;
```

```
do {
...
} while (i++ < j)
```

The following assembly code generated now looks like:

```
L0:
 ... ; loop body
 cmp r0,r1
 brr L0, lt
```

8.14 Smart Usage Of Pointers

Try to substitute array indices and structure elements access with pointers. The usage of pointers (especially inside inner loops) makes it easier for the C/C++ Compiler to perform optimizations on the Code and will yield a more efficient register allocation and usage of post-modification in the output assembly code.

For example, in order to access efficiently the elements within the defined structure:

```
struct summator_struct
{
        short *InParam ;
        short *input1,*input2,*output ;
} ;
```

Write:

/** This function is longer in C but utilizes pointer usage instead of structure elements access, which makes it easier for the Compiler to perform optimizations such as generating post modifications

* */

```
short summator_run2(struct summator_struct* sum)
{
 short **p;
 short *in1,*in2,*out ;
 p=(short**)sum +1;
 in1=*(p++) ;
 in2=*(p++) ;
 out=*p ;
 *(out++)=*(in1++)+*(in2++) ;
 *out=*in1+*in2;
}
```

Instead of writing:

/*This function is short in C but generate a longer assembly code :*/

```
short summator_run1(struct summator_struct* sum)
{
        *(sum->output)=*(sum->input1)+*(sum->input2) ;
        *(sum->output+1)=*(sum->input1+1)+*(sum->input2+1) ;
}
```

9. Appendix

The following chapters are included:

- **IEEE-754-1990 Floating Point Format**
- **Glossary**
- **License**

9.1 IEEE-754-1990 Floating Point Format

The CEVA-Toolbox Software Development Tools is conformed to IEEE 754-1990 standard for binary floating point arithmetic. This standard is commonly referred to as "IEEE floating point". The standard utilizes the 32-bit IEEE floating point format as follows:

N = 1.F* 2^(E-127)

N = floating point number

F = fractional part in Binary notation

E = exponent in bias 127 representation

In 32 bits IEEE format, 1 bit is allocated as the sign bit, the next 8 bits are allocated as the exponent field, and the last 23 bits are the fractional parts of the normalized number.

Sign Exponent Fraction

0 00000000 00000000000000000000000

Bit 31 [30 -- 23] [22 -- 0]

A sign bit of 0 indicates a positive number, and a 1 is negative. The exponent field is represented by "excess 127 notation". The 23 fraction bits actually represent 24 bits of precision, as a leading 1 in front of the decimal point is implied.

There are some exceptions:

- E = 255; F = 0 ➔ +/- infinity
- E = 255; F != 0 ➔ Not a number. Overflow, error.....
- E = 0; F = 0 ➔ 0

- E = 0; F != 0 ➔ denormalized, tiny number, smaller than smallest allowed.

With exponent field 00000000 and 11111111 now reserved, the range is restricted to 2 ** -126 to 2 ** 127.

IEEE floating point format example:

Suppose that we want to convert 9 97/128 into IEEE 32 bits format.

Following is a suggested solution:

1. Convert to base 2:
   1001.1100001
2. Shift number to the form of 1.FFFFFF * 2**E:
   1.0011100001 * 2**3
3. Add 127 (excess 127 code) to exponent field, convert to binary:
   3+127 = 130 = 10000010
4. Determine the sign bit. If a negative number, set to 1. Otherwise set to 0.
5. Now put the numbers together, using only the fractional part of the number represented by step 2 above (remove the "1." preceding the fractional part):
   0 10000010 00111000010000000000000
   in hex representation, this is
   0100 0001 0001 1100 0010 0000 0000 0000
   or
   411C2000 in Hex format
6. Finally, the standard asks for the low-order byte first:
   00201C41

9.2 Glossary

- **Application Profiler** – Can be invoked while in the Debugger's Simulation Mode to perform statistical analysis on memory usage, program flow, real time consumption, instruction usage and more. The Profiler is capable of identifying bottlenecks. This aids code optimization.
- **ASDSP - Application Specific DSP**.
- **ASSYST ™ Simulator** – Highlights include:
  - Associated with UserIO Dll by allowing convenient simulation of external hardware interface (i.e. Hardware Coders, Glue logic etc.) and the full **system-on-chip**.
  - Enables the simulation of user-defined registers and memory-mapped I/O ports.
  - Easy modeling of an external in C level language.
- **API – Application Program Interface**.
- **CAS – Cycle Accurate Simulator**, a simulator that simulates the CEVA DSPs pipeline, instructions and pins in cycle accuracy. CAS can simulate cycle steps and update the core interface pins after each cycle.
- **CHM - Compressed Help Manuals,** used for the SDT on line Help. The file that is created whenever a Microsoft HTML Help project is compiled. The .CHM file includes all of the files in the project, including HTML topics, images, multimedia files, context-sensitive help topics, map files and .ALI files (for aliases). The .CHM file is updated whenever the project is compiled.
- **CLI - Command Language Interpreter** (Supported by CEVA-Toolbox Debugger). Can be manually activated from either the Debugger's upper screen Edit Line box, the Command window, GUI controls or via the Debugger's script (.dbg) file.
- **COFF - Common Object File Format**. Supported by the CEVA-Toolbox Assembler and Linker.
- **C/C++ Compiler** - a tool that translate for a high level language – C/C++ - into the

target CEVA DSPs assembly language in a way that produces efficient code and yet preserves the functionality described in the C/C++ source.

Highlights Includes:

- GNU FSF based Compiler.
- DSP-specific high-level language extensions.
- Special optimization for DSP operations and parallel instructions.
- Convenient interface to handwritten assembly codes.
- Full set of libraries, including IEEE floating point emulation.
- Special fast floating format emulation (10X faster than standard IEEE).

- **Debugger** – A tool that is used for locating bugs in the developed application sources by high level of visualization of the CEVA DSPs and application internals.

  CEVA-Toolbox Debugger Highlights Includes:

  - Graphic Interface – MFC based (multiple views, popup menus, data tips etc.).
  - Symbolic, dual mode debugging (C/C++, Assembly and Mixed mode).
  - Integrated simulator .
  - Extendible simulator (ASSYST™).
  - Customizable Hardware Interface.
  - Integrated TCL Interpreter.
  - Extensive I/O support in Simulation and Emulation modes.
  - Counted and conditional breakpoints.
  - Paging support
  - Trace support
  - Interrupts Simulation.
  - Graphic data display.
  - Integrated, powerful Application Profiler.
  - Hardware accelerator Board for real-time SW development.
  - MATLAB connection

- **Debugger Modes** – Include:
  - **Emulation Mode** – An operation mode in which the CEVA-Toolbox Debugger communicates with the hardware emulator through a dedicated user- customizable UserHWIF DLL (hardware interface). This DLL can be modified in order to adapt the Debugger to work with different emulator hardware or protocols supplied by CEVA.
  - **Simulation Mode** – The default CEVA-Toolbox Debugger operation mode in which each program instruction (asm) is simulated, one at a time (ISS – Instruction Set Simulator) or in cycles steps level while simulating the pipeline and core interface pins (CAS – Cycle Accurate Simulator). **UserIO** DLL is used for simulation extension of the Core's external hardware and Glue Logic.
- **DLL** – **Dynamic Linked Library,** Windows term for a self-contained application module that provides services of support for a set of exported routines and symbols. DLLs provide the ability to link libraries of routines dynamically to an application. Three user's customizable DLLs are provided with CEVA-Toolbox tools: **UserIO**, **UserHWIF** and **UserDBG**.
  - **UserIO DLL** – Provides CEVA-Toolbox Debugger's **Simulator** Extension. Enables the simulation of Core's external Glue Logic and peripherals by implementing internal C modeled functions.
  - **UserHWIF DLL** – Provides the interface between the Debugger and the external hardware (CEVA DSPs specific Development Kit) in **Emulation** mode. The default DLL provided supports the company's development kit. The user can modify the DLL supplied if Dev kit changes or alternatively replace the provided DLL with new functions to fit his specific hardware.
  - **UserDBG DLL** – Supporting Multi-core debugging environment, providing communication and synchronization between CEVA-Toolbox Debugger and other debuggers. In addition, enables the extension of Debugger's CLI macros and Tool-Bar customization. Active both in **Emulation** and **Simulation** modes.
- **FSF - Free Software Foundation**, dedicated to eliminating restrictions on copying,

redistribution, understanding, and modification of computer programs. This is done by promoting the development and use of free software in all areas of computing-but most particularly, by helping to develop the GNU operating system. Refer to **http://www.gnu.org/home.html** for details.

- **GNU** - Gnu's Not Unix, is the name for the complete Unix-compatible software system which written by **Richard Stallman** so he can give it away free to everyone who can use it. The GNU Project was launched in 1984 to develop a complete free Unix-like operating system--the GNU system. Variants of the GNU system, which use the kernel Linux, are now widely used; though these systems are often referred to as ``Linux'', they are more accurately called **GNU/Linux** systems. Refer to **http://www.gnu.org/home.html** for details.
- **GUI** – **Graphic User Interface.** Defines the graphic objects and operation for application's input / output user interaction.
- **HTML** - **Hypertext Markup Language**. A set of tags used to mark the structural elements of text (ASCII) files. HTML files include tags that create hyperlinks to other documents on the Internet. HTML can be regarded as a programming language because not only does it define what part each piece of text plays in a topic, but it also specifies to load. HTML is used in SDT on line Help.
- **ISS** – **Instruction Set Simulator,** a simulator that can simulate/step in the CEVA DSPs instruction level. ISS is not aware of the pipeline and does not simulate in cycles nor pin accuracy.
- **Librarian** - A utility that groups together multiple object files (produced by the Assembler) into a single library file that is link-able by the Linker.
- **Linker** - A tool that combines together into one executable the application's entire object files (and selective libraries content). The Linker maps data and code sections to physical addresses and resolves references between the object files and libraries.
- **MPP** – **Macro Pre-Processor**, a proprietary pre-processor that is usually used for assembly files level.

- **Mailbox** - A data dual-port memory area located in the Development Kit emulator and used for communication and data transfer between the CEVA-Toolbox Debugger (PC) and the CEVA DSPs application.
- **Perl – Practical Extraction and Reporting Language** - Programming language for easily manipulating text, files and processes. A Perl interpreter is optionally being activated by the Assembler and Linker for input files manipulation.
- **SDT** – **Software Development Tools**. CEVA-Toolbox Software Development Tools include advanced code generation tools for developing DSP applications in C/C++ or/and assembly, a powerful Graphical User Interface (GUI) Debugger, Profiler and easy to use IDE (Integrated Development Environment). The CEVA-Toolbox SDT operates on PC/Windows based platforms.
- **TCL** – **Test Command Language** (Supported by the CEVA-Toolbox Debugger). Enables the programmer to create sophisticated and complex Debugger's scripts that may contain conditionals, loops, and file I/O.

9.3 License

IMPORTANT: READ CAREFULLY BEFORE USING THIS PRODUCT

CEVA SOFTWARE END USER LICENSE AGREEMENT
REDISTRIBUTION OR RENTAL NOT PERMITTED

ATTENTION: THIS IS A LICENSE, NOT A SALE. THIS PRODUCT IS PROVIDED EXCLUSIVELY TO LICENSEES THAT HAVE RECEIVED EXPRESS AUTHORIZATION TO RECEIVE THIS PRODUCT FROM CEVA LTD., CEVA TECHNLOGIES INC. AND CEVA INC. IF YOU HAVE NOT RECEIVED SUCH EXPRESS AUTHORIZATION, YOU MAY NOT USE THIS PRODUCT AND YOU MAY NOT BECOME A PARTY TO THIS LICENSE. UNAUTHORIZED USE IS EXPRESSLY PROHIBITED.

THE FOLLOWING LICENSE DEFINES HOW AUTHORIZED LICENSEES MAY USE THIS PRODUCT AND CONTAINS LIMITATIONS ON WARRANTIES AND REMEDIES.

BY INSTALLING THE CD-ROM OR USING THE SOFTWARE OR OTHER ELECTRONIC CONTENT ACCOMPANYING THIS LICENSE, YOU ACKNOWLEDGE THAT YOU HAVE READ THIS LICENSE AND ACCEPT ITS TERMS. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE CD-ROM OR DELETE ANY FILES FROM YOUR COMPUTER TERMINAL AND RETURN ALL ACCOMPANYING PRINTED MATERIALS WITHIN TEN (10) DAYS TO THE PLACE YOU ACQUIRED IT.

THE SOFTWARE TOOLS, EXCLUDING THE C/C++ COMPILER

1. THE PRODUCT.

A. CEVA LTD., CEVA INC. and CEVA Technologies Inc. (together "CEVA") provide you with storage media containing the CEVA-Toolbox (Software Development Tools - these Software Development Tools, excluding the C/C++ Compiler are referred to as the "Software"), user manual, and accompanying documents (the "Documentation"). The Software and the Documentation are together referred to as the "Product". This license (the "License") sets forth the

terms and conditions of your use of the Product. Please note that the copyright and all other rights in the Product shall remain with CEVA.

B. The embedded editor in the IDE, which is part of the Software is based on the Scintilla and SciTE editor - Copyright 1998-2003 by Neil Hodgson, and is subject to certain conditions and License by Neil Hodgson, that appear in full below at this end user license agreement.

C. In addition to the Software, the storage media contains a C/C++ Compiler (the "Compiler"), which has been developed by based upon software developed by the Free Software Foundation, Inc., and is subject to the GNU General Public License, version 2 (the "Public License"). The Public License appears in full at the end of the user manual. Some of the Compiler libraries is also based upon software developed by the University of California, Berkeley and its contributors, and is subject to certain conditions (the "Berkeley Conditions"). The Berkeley Conditions appear in full at the end of the user manual. The Compiler is provided on the storage media with the Software free of charge.

2. LICENSE GRANT.

A. CEVA grants you a non-exclusive and non-transferable license to use the Product at a single business location. Except as expressly permitted herein, you may use the Product solely and exclusively for internal purposes within your organization. You may not transfer the Product, without CEVA's prior written approval.

B. Unless otherwise agreed in advance in writing by CEVA, which may require the payment of fees to CEVA, you may use the Software on up to the maximum number of computer terminals as set forth in your purchase order, or on CEVA's invoice, on a single local area network, at a single business location, on a single network server.

C. Unless otherwise agreed in advance in writing by CEVA,

which may require the payment of fees to CEVA, you may not:

(1) allow access to the Product to anyone outside of your organization;

(2) modify, translate, reverse engineer, decompile, disassemble, or create derivative works based on the Software, except to the extent this restriction is expressly prohibited by applicable law;

(3) rent, lease, grant a security interest in, or otherwise transfer rights to the Product;

(4) remove or alter any trademark, logo, copyright or other proprietary notices, legends, symbols or labels on the Product, or in copies you have made of the Product; or

(5) duplicate the Documentation. Additional copies of Documentation may be purchased from CEVA.

D. Unless otherwise provided in writing by CEVA, you may:

(1) make a single copy of the Software for archival purposes, and the copy must contain all of the original Software's proprietary notices; and

(2) if you have purchased a license for multiple copies of the Software, make the total number of copies of the Software (but not the Documentation) stated on the packing slip(s), invoice(s), or Certificate(s) of Authenticity, provided any copy must contain all of the original Software's proprietary notices. The number of copies on the packing slip(s), invoice(s), or Certificate(s) of Authenticity is the total number of copies that may be made for all platforms.

E. The Compiler is subject to the Public License, which permits you to copy, distribute, and modify the Compiler, in accordance with the terms therewith. For up to three years from the date that you acquire the Compiler from CEVA, CEVA shall provide you upon request a complete machine readable copy of the source code for the Compiler for a charge that does not exceed CEVA's cost of physically distributing such source code to you. The Compiler is also subject to the Berkeley Conditions.

F. You expressly acknowledge that the Product and all intellectual property rights in the Product are owned by CEVA. You further acknowledge that the Software was developed by CEVA through the application of methods and standards of judgment developed and applied through the expenditure of substantial time, effort, and money. You will not take any action to jeopardize, limit or interfere in any manner with CEVA's ownership of or rights with respect to the Product. The Product is protected by copyright and other intellectual property laws and by international treaties. The Product contains trade secrets. You agree to protect all copyright and other proprietary rights of CEVA in the Product, including trade secrets, and you will honor and comply with reasonable written requests made by CEVA to protect CEVA's contractual, statutory, and common law rights in the Product.

3. UPDATES. Subject to payment by you of the applicable fees, CEVA will provide you with updates to the Product. Each such update shall be subject to the terms and conditions of this License as if each such update were the original Product referred to in this License.

4. TERM AND TERMINATION. This License is effective until terminated. The License will terminate automatically without notice from CEVA if you fail to comply with any provision of this License. You agree upon any termination of this License to destroy the original copy of the Product together with all updates and copies of the original or any updates that you have in your possession. The limitations of warranties and liability provided below shall continue in force after any termination.

5. LIMITED WARRANTY.

A. CEVA DOES NOT WARRANT THAT YOUR USE OF THE SOFTWARE WILL BE UNINTERRUPTED OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR-FREE.

B. CEVA WARRANTS THAT THE MEDIA CONTAINING THE SOFTWARE, IF PROVIDED BY CEVA, IS FREE FROM DEFECTS IN MATERIALS AND WORKMANSHIP AND WILL SO REMAIN FOR NINETY (90) DAYS FROM THE DATE YOU ACQUIRED THE SOFTWARE. CEVA'S SOLE LIABILITY FOR ANY BREACH OF THIS WARRANTY SHALL BE, IN CEVA'S SOLE DISCRETION:
(I) TO REPLACE YOUR DEFECTIVE MEDIA OR SOFTWARE;

OR (II) TO ADVISE YOU HOW TO ACHIEVE SUBSTANTIALLY THE SAME FUNCTIONALITY WITH THE SOFTWARE AS DESCRIBED IN THE DOCUMENTATION THROUGH A PROCEDURE DIFFERENT FROM THAT SET FORTH IN THE DOCUMENTATION;

OR (III) TO REFUND THE FEE YOU PAID FOR THE SOFTWARE. REPAIRED, CORRECTED,

OR REPLACED SOFTWARE AND DOCUMENTATION SHALL BE COVERED BY THIS LIMITED WARRANTY FOR THE PERIOD REMAINING UNDER THE WARRANTY THAT COVERED THE ORIGINAL SOFTWARE, OR IF LONGER, FOR THIRTY (30) DAYS AFTER THE DATE

(A) OF DELIVERY TO YOU OF THE REPAIRED OR REPLACED SOFTWARE, OR (B) CEVA ADVISED YOU HOW TO OPERATE THE SOFTWARE SO AS TO ACHIEVE SUBSTANTIALLY THE SAME FUNCTIONALITY DESCRIBED IN THE DOCUMENTATION. CEVA WILL USE REASONABLE COMMERCIAL EFFORTS TO REPAIR, REPLACE, ADVISE OR, REFUND PURSUANT TO THE FOREGOING WARRANTY WITHIN THIRTY (30) DAYS OF BEING SO NOTIFIED.

C. YOUR SOLE AND EXCLUSIVE REMEDY IN THE EVENT OF A DEFECT IS EXPRESSLY

LIMITED AS PROVIDED ABOVE. IF FAILURE OF STORAGE MEDIA FOR THE SOFTWARE IS THE RESULT OF AN ACCIDENT, MISUSE, OR ABUSE, THEN CEVA SHALL HAVE NO RESPONSIBILITY UNDER THE TERMS OF THIS LIMITED WARRANTY. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS, WHICH VARY FROM JURISDICTION TO JURISDICTION. NO DEALER, AGENT, OR EMPLOYEE OF CEVA IS AUTHORIZED TO MAKE ANY MODIFICATIONS, EXTENSIONS, OR ADDITIONS TO THIS LIMITED WARRANTY.

6. LIMITATION OF WARRANTIES. EXCEPT AS EXPRESSLY PROVIDED ABOVE, (I) CEVA DOES NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE PRODUCT OR THE COMPILER; AND (II) THE PRODUCT AND THE COMPILER ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, AS TO THEIR PERFORMANCE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, USE, OR THOSE ARISING BY LAW, STATUTE, USAGE OF TRADE OR COURSE OF DEALING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE AND THE COMPILER IS BORNE BY YOU. SHOULD

THE SOFTWARE PROVE DEFECTIVE IN ANY RESPECT, YOU AND NOT CEVA OR ITS DEALERS, LICENSEES, AGENTS, EMPLOYEES, OR SUPPLIERS ASSUME THE ENTIRE \COST OF ANY REPAIRS AND CORRECTIONS. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. SOME JURISDICTIONS DO NOT

ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

7. LIMITATION OF LIABILITY. UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT, CONTRACT, OR OTHERWISE, SHALL CEVA, OR ITS SUPPLIERS OR RESELLERS BE LIABLE TO YOU OR ANY OTHER PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND,

INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, LOSS OF BUSINESS INFORMATION, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES PURPORTEDLY CAUSED BY THE PRODUCT OR THE COMPILER. IN NO EVENT WILL CEVA BE LIABLE FOR ANY DAMAGES IN EXCESS OF THE AMOUNT RECEIVED FROM YOU FOR A LICENSE TO THE PRODUCT, EVEN IF CEVA SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY THIRD PARTY. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM CEVA'S

NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS EXCLUSION AND LIMITATION MAY NOT APPLY TO YOU.

8. HIGH RISK ACTIVITIES. THE SOFTWARE AND THE COMPILER SOFTWARE ARE NOT FAULT-TOLERANT AND ARE NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF THE SOFTWARE OR THE COMPILER COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE ("HIGH RISK ACTIVITIES"). ACCORDINGLY, CEVA AND ITS SUPPLIERS

SPECIFICALLY DISCLAIM ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR HIGH RISK ACTIVITIES.

9. MISCELLANEOUS. Except as described in the Public License regarding the Compiler and in the License for the Scintilla and SciTE software regarding the embedded editor in the IDE only, this License represents the complete agreement concerning the License granted hereunder and may be amended only by a writing executed by both parties. THE ACCEPTANCE

OF ANY PURCHASE ORDER PLACED BY YOU IS EXPRESSLY MADE CONDITIONAL ON YOUR ASSENT TO THE TERMS SET FORTH HEREIN, AND NOT THOSE IN YOUR PURCHASE ORDER. If any provision of this Agreement is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable and the remaining provisions of this License shall remain in full force and effect. Unless otherwise agreed in writing, all disputes relating to this Agreement shall be subject to the jurisdiction of the county of the state courts of the state of California, U.S.A. located in San Francisco. This License shall be construed and governed in accordance with the laws of California, excluding any otherwise applicable conflict of law provisions (except to the extent applicable law, if any, requires otherwise). The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded from application to this License.

* * *

The embedded editor in the IDE, which is part of the Software is based on the Scintilla and SciTE editor - Copyright 1998-2003 by Neil Hodgson, and is subject to the following License for Scintilla and SciTE:

License for Scintilla and SciTE

Copyright 1998-2003 by Neil Hodgson - neilh@scintilla.org

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

* * *

The C/C++ Compiler is based upon software developed by the Free Software Foundation, Inc. and is subject to the following GNU General Public License:

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy,

distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free software.
If the software is modified by someone else and passed on, we want its
recipients to know that what they have is not the original, so that any
problems introduced by others will not reflect on the original authors'
reputations.

Finally, any free program is threatened constantly by software patents.
We wish to avoid the danger that redistributors of a free program will
individually obtain patent licenses, in effect making the program proprietary.
To prevent this, we have made it clear that any patent must be licensed for
everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification
follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION
AND MODIFICATION

0. This License applies to any program or other work which contains a
notice placed by the copyright holder saying it may be distributed under
the terms of this General Public License. The "Program", below, refers
to any such program or work, and a "work based on the Program" means
either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it, either
verbatim or with modifications and/or translated into another language.
(Hereinafter, translation is included without limitation in the term
"modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered
by this License; they are outside its scope. The act of running the Program
is not restricted, and the output from the Program is covered only if its
contents constitute a work based on the Program (independent of having been
made by running the Program). Whether that is true depends on what the Program
does.

1. You may copy and distribute verbatim copies of the Program's source code as
you receive it, in any medium, provided that you conspicuously and

appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your

rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code,

even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

The C/C++ Compiler is also based upon software developed by the University of California, Berkeley and its contributors and is subject to the following conditions:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors. 4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

10. Index