CEVA
The DSP Powerhouse

CEVA-Toolbox™

Binary Tools Reference Guide

Rev 15.1.0

August 2015

DSP Division

Documentation Control

History Table:

Version	Date	Description	Remarks
Rev 10.0	01/12/2012	First tracked version	Added on 24/6/2013
Rev 10.0.4	14/01/2014	Added TL4 support	Added on 14/01/2014
Rev 10.1	23/04/2014	Added new Linker switch	Added on 23/04/2014
Rev 10.3	23/10/2014	Changed Assembler switch	Added on 23/10/2014
Rev 15.1.0	11/08/2015	Updates for XM4 V15.1	

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

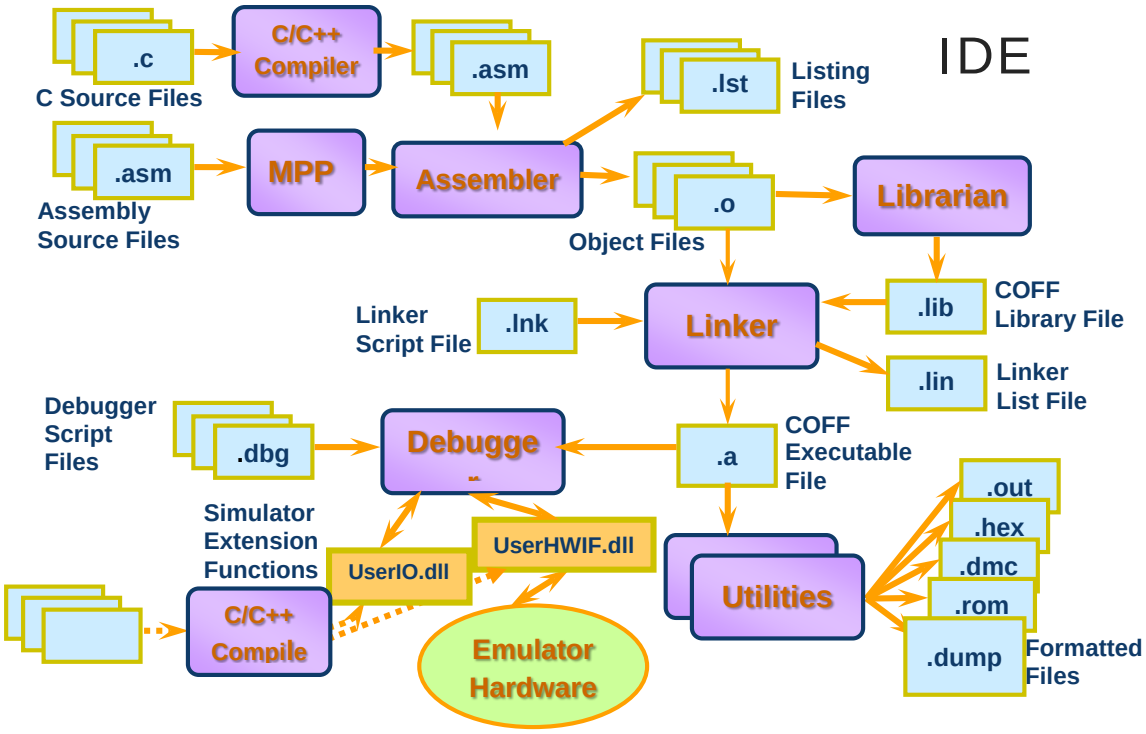
Table of Contents

List of Tables

List of Figures

1. Introduction

CEVA-Toolbox™ Software Development Tools (SDT) Diagram



1.1 Guide Scope

This guide describes the CEVA, Inc. proprietary **COFF** (Common Object File Format) macro Assemblers and the Linker for the **CEVA DSPs** assembly language. In addition, the guide describes the way to generate programmable load files, Linker maps, symbol tables and **COFF** libraries. The Assembler and Linker are part of the CEVA-Toolbox Software Development Tools (SDT) provided by CEVA.

CEVA-Toolbox™ Assembler is referred as:

- **tl3asm** for CEVA-TeakLite-III™
- **tl4asm** for CEVA-TL4XX™
- **CEVA-X16/24/32asm** for CEVA-X™ family (e.g. cevax16asm for the CEVA-X1620)
- **cevaxcasm** for CEVA-XC™
- **xm4asm for CEVA-XM4™**

The CEVA-Toolbox Linker is referred to as **cofflnk** (COFF Linker).The information included in this reference guide is valid for both the Windows and Linux versions. Wherever needed, relevant sections detail the differences.

This guide assumes the reader is familiar with assembly programming principles and with the DSP Core instruction set. The instruction set is detailed in the CEVA DSPs (CEVA-CEVA-TeakLite -III/ CEVA-TL4xx / CEVA-X/CEVA-XC/CEVA-XM4) Architecture Specifications. It explains the various invocation parameters, all Assembler directives, the Linker script files format and options, some programming hints, error messages and restrictions.

Notes:

1. ***CEVA DSPs*** *family of* ***licensable*** *DSP cores consists of* ***CEVA-Teak*** *(including* ***CEVA-TeakLite,*** *CEVA-TeakLite-II,* *CEVA-TeakLite-III and* ***CEVA-Teak*** *DSP cores),* ***CEVA-X*** *(comprised of three members: 16, 24 and 32-bit versions),* ***CEVA-XC*** *and* ***CEVA-XM4***
2. *.Throughout the guide, the* ***term*** *DSP* ***CEVA DSPs*** *refers to the core family members specified above.*

1.2 Applicable Documents

1. CEVA-Toolbox Debugger Reference Guide
2. CEVA-Toolbox C/C++ Compiler Reference Guide
3. CEVA DSPs Architecture Specifications
4. CEVA-Toolbox IDE Reference Guide

2. Assembler Macro Preprocessor

The Assembler Macro Preprocessor (MPP) includes the following topics:

- **Macro Preprocessor Invocation**
- **Macro Preprocessor Operators**
- **Macro Preprocessor Directives**

2.1 MPP Invocation

CEVA-Toolbox mpp preprocessor can be invoked directly from the command line:

mpp [options] sourcefile > outputfile

Table 1 below describes the options available when invoking the MPP.

Table 1 MPP Invocation Options

Option	Description
-i PathName	Searches for included files in the provided path if not found in current directory. Multiple directories can be given, separated by **";"**.
-d Sym[=val]	Defines a symbol and optionally assign to it an expression or value. In order to define multiple symbols, use –d option before each symbol.
-s	Filters out all statements falling under a false MPP directive condition (.IF, .ELSE). By default, these lines are printed as comments.
-c	Filters out all user's comments.
-w	Prints warning messages in the output file.
-h	Prints a help message.
-o FileName	Forces a particular output filename (replaces the "> outputfile" part).
-m	Instructs the MPP to filter out false blocks (as in the -s flag), and all the MPP directives encountered. This leaves intact only effective user's code and user's comments.
-n	Instructs the MPP to avoid adding line number directives to its output (Refer to Appendix A .X, .LINE MPP directives).
-quiet	Do not print the banner message.

Option	Description
-msgFullPath	Display error and warning messages with files full path.
-MM	Generate a make rule for the input file.

Notes:

1. *Uppercase letters can be used for selecting the options (-I, -D, -S, -C, -W, -H, -O, -M, -N).*
2. *An environment variable named **MPP** can be set with a command to specify a path for searching included files, for example:. in DOS, "SET MPP=c:\user\include;x:\ceva-x16\inc". If both, environment variable **MPP** is set and the **-i** option is used, then the environment variable is ignored. By default, the active (current) directory is always searched first*
3. ***MPP** returns an error code of 1 upon any error detection*
4. *The usage of **"-d Sym"** is equivalent to writing a **".EQU Sym 1"** statement in the source file first line. A conditional statement of the kind **".IFDEF Sym"** that will be encountered in the source file by the **mpp**, will be treated as **TRUE**. Using **-d Sym=Val** is equivalent to writing a **".EQU Sym Val"** statement. A conditional statement of the kind **.IF Sym == Val** will be handled **TRUE** in this case.*
5. *The usage of .IF X==Val in the source file when X is not defined,it is equivilant to the expression ".IF X==0". In other words, undefined variables are defined as zero. In addition, for undefined variables the .IFDEF condition is a false condition.*
6. *When a file is not found and the include command line option is used (-i), error is given for all include directories.*

2.2 Macro Preprocessor Operators

The macro preprocessor supports the following one character string operator:

' OPERATOR

The paste operator implemented as a single quote ('), enables pasting in C-like **define**, for **example**:

```
.EQU Index 10

.EQU String Abc

mov #Label'Index, r0 ; which translates to mov #Label10, r0

mov String'Index, y ; which translates to mov Abc10, y
```

2.3 Macro Preprocessor Directives

The following directives are supported by the **CEVA-Toolbox** Macro Pre-Processor (in addition to the Assembler directives):

Note: Refer to the **Appendix 6.1** for additional non-user directives used internally by the **CEVA-Toolbox** Compiler and **mpp**

2.3.1. .EQU

The .**EQU** directive is equivalent to the C '**#define'** directive, i.e., the macro preprocessor treats equates as literal (which may be nested). The first format is used for simple definitions of constants represented by symbols; the second format is used for smart C-like macros.

```
.EQU Symbol FreeText   (Symbol = string presentation   FreeText = string
substitute)
 ; or
.EQU Symbol(ParameterList) FreeText
```

Notes:

1. *Equates to the next PC (.EQU var $) do not make sense and should not be used*
2. *Expansion is delayed until the final stage, that is, nesting is possible.*
3. *Up to approximately 30000 equates may be defined in one module.*
4. *Maximum length of each equate body is limited to about 1000 characters.*
5. *Maximum number of parameters is 20.*
6. *In the second format, the opening parenthesis must immediately follow the symbol name.*
7. *Equates may be redefined. The preprocessor will generate a warning message.*
8. *If the equated symbol is not followed by free text, a default value of 1 is assumed. This can be used to define symbols for use by the conditional directives, without giving a value to the symbol. The **.PURGE** directive can then be used to undefine this symbol.*

- Examples:
- .EQU AAA BBB
- .EQU BBB 111

  mov #AAA, r0 ; translates to mov #111, r0
- .EQU BBB 22 ; redefinition

  mov #AAA,r1 ; translates to mov #22, r1
- .EQU min(a,b) (((a) <= (b)) ? (a) : (b))

  add ##min(3,4), a0 ; translates to add ##(((3)<=(4))?(3):(4)), a0
- .EQU Flag ; equivalent to .EQU Flag 1

.IFDEF Flag

.INCLUDE “file” ; file will be included

.ENDIF

.PURGE Flag

2.3.2. .INCLUDE

.INCLUDE "FileName"

The **.INCLUDE** directive is similar to the C **'#include'** directive, i.e., it instructs the Assembler to read and merge another module (file) into the source file at the line where this directive is located. Conventional completely specified DOS path names of files may be used to access files outside the working directory. Alternatively, an environment variable, named **MPP**, may be set (using the DOS command SET) to tell the macro preprocessor to search the file in the path specified by the **MPP** variable in case the file is not found in the current (working) directory. Multiple directories may be specified separated by a semicolon (;). In addition, one can overwrite the environment variable path, by using the **-i** option when invoking the macro preprocessor (Refer to **MPP Invocation** chapter for details).

Notes:
1. *Included nested files are limited up to 14 levels.*
2. *Up to 50 paths may be specified and each is limited to 80 characters.*
3. *It is possible to use environment variable to denote the path to the included file (for example: .INCLUDE %WORKDIR%\filename.inc).*

2.3.3. .PURGE

.PURGE symbol

The **.PURGE** directive is the equivalent of the C '**#undef'** directive.
It deletes the definition of a symbol.

Example:

```
.PURGE MySymbol
```

2.3.4. .ELSE

The **.ELSE** directive marks the end of a previous conditional block and the beginning of a new conditional block. It is used to control the assembly conditionally as follows:
If the Boolean expression associated with the previous conditional block is evaluated as **false**, then the new conditional block (following the **.ELSE** directive) will be included in the source.

Note: *The .ELSE directive is defined as a conditional directive and can be used (in conjunction with others) to create different versions of a program using a single source file, depending on some assembly time condition.*

2.3.5. .ELIF

.ELIF BooleanExpression

The **.ELIF** directive marks the beginning of a nested conditional block. It is used to control the assembly conditionally as follows:
If the Boolean expression is evaluated as **true**, then the next conditional block is included in the source.
If the Boolean expression is **false**, then the conditional block is ignored.

Note: *The .ELIF directive is defined as a conditional directive and can be used (in conjunction with others) to create different versions of a program using a single source file, depending on some assembly time condition.*

2.3.6. .ENDIF

The **.ENDIF** directive marks the end of a conditional block. (See also previous conditional directives).

Examples of conditional directives:

- .IF Flag>0
  mov #0x0f, r1

.ELSE

mov #0,r2

.ENDIF

- .IFDEF MySymbol

clr a0

.ELIF Flag<2*4

mov #1,a1l

.ELSE

mov #2,a1l

.ENDIF

Note: *The .ENDIF directive is defined as a conditional directive and can be used (in conjunction with others) to create different versions of a program using a single source file, depending on some assembly time condition.*

2.3.7. .IF

.IF BooleanExpression

The **.IF** directive marks the beginning of a conditional block. It is used to control the assembly conditionally as follows:

If the Boolean expression is evaluated as **true**, then the source lines in the conditional block (following the .**IF** directive up to another conditional directive) are included in the source.

If the Boolean expression is **false**, the conditional block is ignored.

Note: *The .IF directive is defined as a conditional directive and can be used (in conjunction with others) to create different versions of a program using a single source file, depending on some assembly time condition..*

2.3.8. .IFDEF

.IFDEF Symbol

The **.IFDEF** directive marks the beginning of a conditional block. It is used to control the assembly conditionally as follows:

If the symbol is defined previous to the current section location counter, then the source lines in the conditional block are included in the source.

If the symbol is undefined, the conditional block is ignored.

Note: *The .IFDEF directive is defined as a conditional directive and can be used (in conjunction with others) to create different versions of a program using a single source file, depending on some assembly time condition.*

2.3.9. .IFNDEF

.IFNDEF Symbol

.IFNDEF directive marks the beginning of a conditional block. It is used to control the assembly conditionally as follows:

If the symbol is **not** defined previous to the current section location counter, then the source lines in the conditional block are included in the source.

If the symbol is defined, the conditional block is ignored.

Note: *The .IFNDEF directive is defined as a conditional directive and can be used (in conjunction with others) to create different versions of a program using a single source file, depending on some assembly time condition..*

2.3.10. .ENDM

The **.ENDM** directive terminates definition of a new macro. A macro definition cannot be nested inside another macro definition. See **.MACRO** section for details and examples.

2.3.11. .MACRO

.MACRO Symbol [ParameterList]

The **.MACRO** directive starts the definition of a new macro. A macro is composed of a declaration, macro terminating statement (see **.ENDM**) and a macro body of one or more assembly instructions in between. When the macro is declared, it is given a unique name and an optional parameter list. As in C, macros are treated as literals. Unlike C, the parameter list is defined or referenced without surrounding parentheses. The main difference between macro and equate is that macros can expand over multiple lines. Macro definitions may not include other macro definitions, but Macros can use previously defined Macros; hence, Macro nesting is possible.

Notes:

1. *About 1000 Macros are allowed in each module.*
2. *As default, macros may not contain more than **15000** characters inside the Macro's body.*
3. *A maximum of 20 parameters is allowed. Each parameter is limited to 200 characters.*
4. *Macro names are not allowed with embedded white space characters.*
5. *Macro definitions appear in the listing file as a comment. The expansion of Macros in the listing is controllable by a switch. An additional **.X** directive is generated in the listing by the macro preprocessor for each macro that has more than one line in its body, for the purpose of synchronizing reports on erroneous lines by the Assembler.*
6. *Macro definitions cannot be nested. However a macro definition can use a previously defined macro (forward references are not allowed).*

- Macro Definitions Examples:
- .MACRO MyMac

  mov #0, r0

  mov #1, r1

.ENDM

- .MACRO MyMacWithArgs N0,N1,N2

  mov #N0, r0

  mov #N1, r1

  mov #N2, r2

.ENDM

- .MACRO NestedMac

  clr a0
  MyMac ; nesting previously defined macro

.ENDM

- Macro usage examples:

- MyMac
  ; macro is expanded to:
  mov #0, r0
  mov #1, r1

- MyMacWithArgs 4,5,6
  ; macro is expanded to:
  mov #4, r0
  mov #5, r1
  mov #6, r2

- NestedMac
  ; macro is expanded to:
  clr a0
  mov #0, r0
  mov #1, r1

3. Disassembler Description

3.1 Description

Single Decoder executable file that handles all CEVA COFF based object files. The Disassembler is a command line utility called 'disasm.exe', which can receive an object file or .a file COFF as input. The Disassembler generates an informative assembly file as output when used over an object file or over an .a file.

3.2 Usage

<tools_path>/disasm.exe <COFF name> [command line switches]

3.3 Assembly File Description

3.3.1. Header

Informative commented header for each generated assembly file which includes the following information:

- Date and time
- Input object file name and full path
- Decoder command line used in the current activation
- Object file information
  - Core
  - RTL
  - File Type (object file/.a file)
  - SDT Version
  - Operating System

```
;; CEVA-DSP - SDT - Automated File Generated by Decoder Build #xxx
;; ---------------------------------------------------------------
;;
;; Date: xx/xx/xx
;; Time: xx:xx:xx
;;
;; Object File Information:
;;   - Core          : xxx
;;   - RTL           : xxx
;;   - File Type     : xxx
;;   - SDT Version   : xxx
;;   - Operating System: xxx
;;
;; Decoder Command Line Arguments:
;;   - xxx

IP1
IP2
IP3
...
```

3.3.2. Sections

- Set the correct section with its exact type: .CSECT, .DSECT, .text, .data, .bss etc.
- Print meta sections definition correctly when using the Disassembler over an object file

3.3.3. Function Comments

Informative function description comment are added by default before each .func_start directive:

- Function Name
- Function Profiling Level
- Function Code Size
- Function Estimated Cycle Count, each instruction is a cycle - with bkrep consideration
  - This Cycle Count comment is printed at the end of the function

3.3.4. Labels

Displays all labels for a specific address, each in a separate line.

3.3.5. Instruction Packet Descriptive Comment

Before each instruction packet there a comment describes the following information:

- Instruction packet address
- Instruction packet encoding
- Instruction packet size in bytes
- Instructions included in the instruction packet, each with his size in bytes
- Control words included in the instruction packet, each with his size in bytes

```
;; Address      : 0xxxxxxxxxx
;; Encoding     : 0xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (xx Bytes)
;; Instructions : I1  (xx Bytes)
;;              I2  (xx Bytes)
;; Control Words: CW1 (xx Bytes)
;;              CW2 (xx Bytes)
label_1:
label_2:
IP1
```

3.4 Directive Support

Support for all assembly based directives:

- DB, DW ,DD & DUP if needed
- .align
- .inline
- .newline
- .ORG
- .func_start, .func_end…

3.5 Operators Support

Support for all assembly based operators:

- INCODE
- SIZEOF
- +/- offset…

3.6 Instruction Packet Syntax

- Full mapping: Instruction unit for each instruction in the packet
- Special operands appear in the immediate syntax if they were used in the original assembly file
- Each instruction in the packet is written in a multiple line format by default (can be modified by a switch)

3.7 'bkrep' Indentation

- Set bkrep braces in a dedicated line (not in parallel to an instruction packet)
- bkrep body has 1 tab indentation
- Keep correct indentation for nested bkrep loops

3.8 Command Line Switches

-o <file name>	Sets the output assembly file name Default name is <object file name>_dec.s
-h / -help	Prints a descriptive help page to the command line Ignores all other switches and exit
-oneLineIp	Forces a 1 line instruction packet
-noHeader	Removes the descriptive header comment
-noIpComment	Removes instruction packet comments
-noFuncComment	Remove Functions descriptive comments
-noComments	Combines the following switches • -noHeader • -noIpComment • -noFuncComment
–enc [0x]<encoding> [core]	-enc Disassembly of a single instruction packet • Needs to issue an error if used with a file name argument • Prints to the screen the first instruction packet with its size and encoding • Can be used with no command switch except core
–raw <file name> [core]	-raw Disassembly of multiple instruction packets. The encoding is provided in a file which contains raw data only (not a COFF) • Creates a file and dumps the information to it • Same format as a regular code section • Can be used with no command switch except core
–sec:<sec1>[,<sec2>,…<secN>]	–sec:<sec1>[,<sec2>,…<secN>] Gets the disassembly of the given sections only
–func:<func1>[,<func2>,…<funcN>]	–func:<func1>[,<func2>,…<funcN>] Gets the disassembly of the given functions only
–code:<start address>,<end address>	–code:<start address>,<end address> Gets the disassembly of the given code range only
–data:<start address>,<end address>	–data:<start address>,<end address> Gets the disassembly of the given data range only

3.9 Supported Cores

- CEVA-X
- CEVA-XC
- CEVA-TL321x
- CEVA-TL4xx
- CEVA-XM4

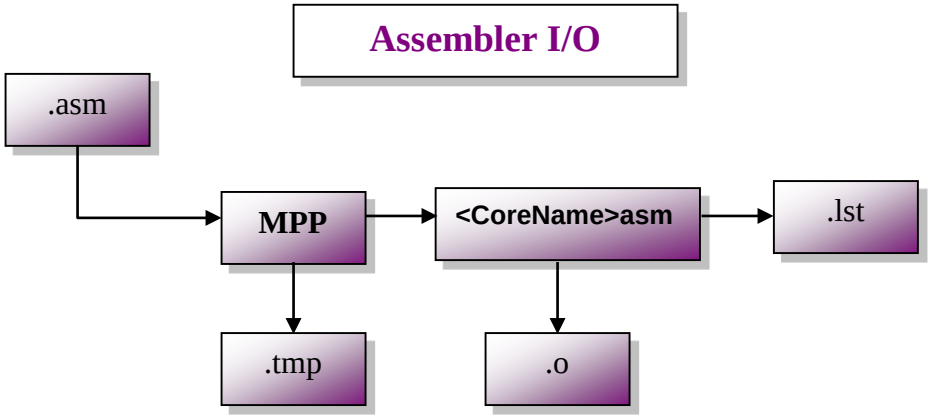
3.10 Notes

- When running the Disassembler over an .a file some of the original relocation entries are not available.
- One of the more important relocation entries which are missing is .INLINE.
- The Linker performs the alignment process and removes the relocation entry.
- 'nop' instructions are added by the Linker where needed (serial or in parallel to previous instructions)
- When running the Assembler and Linker again on the disassembled file may cause the following error: “unable to relocate symbol”. It is caused when there is no room to encode a label properly due to packet size limitation, which is a result of the 'nop' instructions added in parallel.

To overcome this issue there are several possible actions that may be taken:

1. The Linker can be called with the “-noAlign” command line switch (over the original project, before the disassembly process). It causes the Linker to ignore all .INLINE and .ALIGN directives.
2. Remove the parallel 'nop' instructions manually from the disassembled file. The problematic instructions can be easily spotted when running the Assembler over the disassembled file: all problematic instruction packets triggers an Assembler warning: “warning W469: a label was encoded in an instruction that cannot be fully extended, may cause linking error when relocating this label”.

4. Assembler's Description



This chapter includes the following topics:

- **Assembler Highlights**
- **Assembler Invocation**
- **Sections**
- **Meta-Sections**
- **Segments**
- **Output Files**
- **Instruction Set syntax**
- **Labels**
- **Arithmetic and Logic Operations**
- **Assembler Operators**
- **Assembler Directives**

4.1 Programming Address Mode Registers Warnings

4.2 Assembler Highlights

The **CEVA-Toolbox** Assembler fully supports the CEVA DSPs native instructions set (case sensitive). In addition, the Assembler includes a rich repertoire of directives and operators providing the user with maximum programming capability. The Assembler supports the following highlights:

- Comprehensive architectural restriction checker with optional instructions sequence restriction overcoming mechanism.
- Smart branch and call instruction selection mechanism (***sbr*** and ***scall*** respectively).
- Program **Sections** modularity (**Segments / pages** modularity is supported by the COFF Linker).
- Smart immediate constant when more than one option is available.
- Providing the user with full control over program and data memory allocation at link time.
- Preparing the object file output (.o) for a full symbolic debugging capability with the Debugger.
- Preparing the object file output (.o) with all the information required for Assembly and/or C/C++ Source Level Debugging
- C/C++ like operators and conventions, and various macro types, all of which allow easy development of code and data structures.
- C/C++ cross Compiler debugging directives. See the *Debugger and C/C++ Compiler Reference Guides* for more details.
- Assembly Source Level Debugging support by applying the **-g** command line option.
    - Pre-Assembler Perl script manipulation on the input assembly file.

Object files created by the Assembler are linked by COFFLINK, via a Linker script file, into an executable file. The Debugger software can load this file for symbolic (Source Level) **simulation** or **emulation**. Using provided utilities, these loadable files may be converted to the **Intelhex** or other formats for burning EPROMs (Refer to The Intelhex – Generating PROM Burnable Files appendix.

The Macro Assembler is comprised of two parts: the MPP and the main program that analyzes the assembly instructions and generates the object and listing files.

The macro preprocessor performs the following:

1. Merges included files.
2. Replaces macros and equated strings.
3. Filters the relevant portions of the input file in case of conditional assembly.
4. Prepares line number information for the Assembler.

The Assembler main program has the following two stages:

1. Assembly stage - where syntax is checked and the object file is built.
2. Restriction stage - where the specific **CEVA DSPs** architectural restrictions are checked.

4.3 Assembler Invocation

CEVA-Toolbox Macro Assembler invocation in PC is done from MS-DOS command line, where the argument used is the name of the source file. In Linux the Assembler is invoked in the same way

Using MS-DOS or Linux, type:

<CoreName>asm [options] FileNamesList
For instance: xm4asm crt0_nmi_interrupt_base_prologue.asm

Notes:

1. *Apply **tklasm/teakasm/tkl2asm/tl3asm/cevaxcasm/xm4asm**, to invoke the relevant Assembler according to the Core used. For the **CEVA-X** only, apply the following:*
   Cevax16asm for the Ceva-x 16 bits
   Cevax24asm for the Ceva-x 24 bits
   Cevax32asm for the Ceva-x 32 bits
2. *Invoking the **CEVA-Toolbox** Assembler without argument (no Assembly file) results in screen output list of all Assembler options including short descriptions.*
3. *When multiple (same) command line options appears in the command line, only the last occurrence will take effect.For example, **tklasm –p** tes1.asm test2.asm test3.asm.*
4. *Multiple input files support was added in order to reduce the overall time of subsequent activations that require a license request for each activation - the Assembler asks for one license for all assembly files applied in the command line.*

Use invoke the assembler without arguments or with "–h" command line option to get online help.
Example:

```
>xm4asm -h
CEVA-XM4(TM) Assembler V15.1.0 (Build #1213)  Aug  2 2015  [for PC/Windows]
Copyright (C) 2015 CEVA Inc. All rights reserved.


Usage:
   xm4asm [options] source_files_list
Options:
   -cevax<CONFIG>        set the core configuration from a list of:
                         1620 , 1622  , 1641 , 1643
   -d                    don't produce 'short DUP'
   -dataBigEndian        encode data sections in Big Endian format (relevant for
CEVA-XC323, CEVA-XC4210 and CEVA
)
   -dontExpandShort      don't replace '#' with '##'
   -e                    perform .EQU substitution
   -erCodeForWarn        exit with error code for warning(s) too
   -h                    print this help
   -ignoreWarnings,w1,w2,...,wN (where w1-wN are warning numbers)
                         don't generate the warning messages specified
   -IPinfo               print extra instruction packet encoding information
                         into the listing file
   -l listing_file       use the specified file for a listing
   -L                    use stdout for a listing
   -msgFullPath          write files full path in error and warnings messages
   -noMemSize            don't print code and data size into the listing file
```

CEVA-TL3xxx and CEVA-TL4xx notes below.

Notes:

1. *The Assembler returns a non-zero error code if any errors occur. If invoked with the **-erCodeForWarn** option, the Assembler returns a non-zero error code even if only warnings occur (does not create an object file).*
2. *By default, numeric constants are treated as unsigned numbers. For example: in the following statement, AAA will be treated as 0x2000 (8192): **EQU AAA 0x8000/4**. By invoking the Assembler with the **-signed** option, the value of AAA will change to the negative number -8192 (since 0x8000 is -32768 in this case).*
3. *By default, the Assembler ignores leading zeros from numeric expressions. For example, 017 is equivalent to 17; however, for programmers who want to use the octal number format, the **-octal** switch can be used. In this case, the number 017 is treated as an octal number with a value of fifteen (decimal 15).*
4. *In reference to **–p** option, it is possible to pass parameters to the MPP through the Assembler command line also by the environment variable MPP_FLAGS.*
5. *All command line options can be repeated more than once, only the last occurrence will take effect.*
6. *New default behavior for the location of the listing file. The generated listing file will be placed in the same directory the object file (*.o) was directed to using the -o switch. Using -l switch is still stronger then the default.*
7. *Unknown command line options. No fatal error is being generated for typing errors of command line options. Warning is being generated with the failure switch name.*
8. *When using the overcomeerrors switch, the error message that caused the nop addition*
9. *is printed into the listing file. This is also done for the –overcomeWarnings switch. In CEVA-Teak, CEVA-TeakLite, CEVA-TeakLite-II. When using the -w2 or -w3 invocation option, the -pipeline invlocation option is passed to the Assembler as well.*

4.4 Sections

Dividing the program into blocks makes modular programming simple because it enables the programmer to think in terms of blocks of code and blocks of data, defined as sections (i.e. .CODE, .DATA, and directives). Sections with identical name can be re-declared in order to create multiple section instances, which are appended to each other (as part of the linking stage) to create one single section. Assembler and Linker directives enable the programmer to easily create and relocate sections. The Assembler and Linker programs support the old section types with special labels (.DATA/.CODE, requiring a section name prefix) and many directives and operators to enhance software development and to support symbolic, source level debugging. This is in addition to supporting regular sections and labels that do not require the section name to be prefixed to each **label reference** (.DSECT/.CSECT). Overlays in both the data and program space are allowed.

4.5 Meta-Sections

It is possible to define a child section and bind it to a parent section, such that the child section will be mapped by default directly after to the parent section in the linking stage (unless explicitly mapped to other address). The parent section is defined as meta-section.

Example:

```
.CSECT metasec$mysec      ; defines a child section named “mysec” bind to
                          ; parent section “metasec”
label:  nop
        nop
```

At the linking stage, if "mysec" is explicitly mapped (e.g. mysec at 0x1234), the Linker will map it as requested, otherwise, "mysec" will be mapped directly after the meta-section "metasec".

Note: *The formal child section name does not include the meta-section name. That is, when referencing CODE and .DATA type sections with full name notation (i.e. SectionName.Label), only the child name is required.*

Example:

```
mov mysec.label, r0
```

4.6 Segments

Some DSP applications require more than the maximum address range available by the specific DSP core (data space, program space or in both). The problem is magnified due to the usage of C/C++ language in a growing number of DSP software applications, which enables programmers to develop more complicated applications resulting in an increase in program and data space requirements.

In order to overcome the problem of having program or data space size larger than the size that the CEVA DSPs family DSPs can handle, the application's code and data can be divided into several segments (referred also as pages). Each of these segments fit the size of the DSP Core data or program spaces. In CEVA-TeakLite, program and data segment switching is transparent to the core and requires extra external logic and segment registers (for example, external registers or memory mapped I/O). In CEVA-Teak CEVA-TeakLite-II, program space segment switching is supported internally to the cores by a dedicated register.

The **CEVA-Toolbox** Software Development Tools were adapted to work in this expanded environment, providing the user with fully configurable simulation and emulation tools in order to simulate and emulate the user's real environment.

Note: *Refer to the* Segmentation Support in CEVA-Toolbox Software Development Tools, *application note for further information.*

4.7 Output Files

CEVA-Toolbox Assembler outputs an object file (**.o** extension) and a listing file (**.lst** extension). The **MPP** (macro-preprocessor) can be activated independently to view its output for pre-Assembler debugging purposes.

Size report in the listing file:

The Assembler prints a list of sections size and total code / data size to the Assembler listing file (.lst).

- For example:

  ...

  ==========

  CODE AND DATA SIZE:

  ==========

  Code sections size:

  - CODE :0
  - sec1 :10
  - sec2 :20

  Code total size in words:

  - ----------
  - 30

  Data sections size:

  - ----------
  - dsec1 :A
  - dsec2 :14

  Data total size in words:

  - ----------
  - 1E

Assembler's build number is printed to the listing file:

The Assembler prints the build number in the banner of the file. For Example:

CEVA-TeakLite™ Assembler 9.1.0 (build #147) (C) 2005 CEVA Inc.

All rights reserved. File: file.asm Page: 1

Command line: tklasm file.asm

4.8 Instruction Set Syntax

Instruction set syntax is fully specified in the respective **CEVA DSPs** Architecture Specification. The following list specifies programming conventions assumed by the Macro-Assembler, but not mentioned in the **CEVA DSPs Architecture Specification**.

Conventions:

1. The Assembler is case-sensitive. Opcode mnemonics, register names, conditions and flag names are all reserved words that should be lower case and not be used as user-defined symbols (labels or variable names). **Note** that Assembler directives are case non-sensitive, but must be all upper case or lower case (no mixing allowed). Symbol names (either variable or label names) must not start with an underscore, since the C/C++ Compiler, by convention, appends an underscore to each C symbol when creating the assembly translation of the C program. The symbolic Debugger assumes that symbols starting with an underscore originate from the C source.

2. The hexadecimal, octal and binary numeric formats are C style, for example: 0x1234, 026 and 0b1010.

3. The syntax used for offset addresses, i.e. the location counter relative jump address used in ***brr*** and ***callr*** instructions, is as follows:

$; used for next instruction address reference

$ + NumericExpression

$ - NumericExpression

NumericExpression + $

Example:

```
brr $+1
```

4. There are two **immediate** value operators (common to all cores), # and ##, for short values (9 bits or less) and long values (16 bits) respectively.
For further information regarding immediate operators refer to **Assembler Operators** chapter.

5. Direct Addressing notation is according to the following:

- [##long immediate] ; 16 bits Direct Addressing
- [direct] ; 8 bits Direct Addressing
- direct ; 8 bits Direct Addressing
- **@**direct ; 8 bits Direct Addressing

Notes:
1. *That **@direct** notation applies modulo 256 on the Direct address. The other formats require Direct addressing between 0 to 255 (an error is generated when the address noted is out of this boundary).*
2. *The 8 bit Direct Address is automatically concatenated to the page field (8 MSB bits) to create a 16 bit data address.*

6. To ensure full architectural restriction checking associated with the ***bkrep*** instruction, the following considerations are required:

1. A **bkrep** block must be completely contained within a single section.
2. No .**CODE**, .**DATA**, **.CSECT**, **.DSECT** or **.ORG** directives are allowed inside a **bkrep** block.
3. The second operand of the ***bkrep*** instruction, containing the termination address of the block, can contain only a forward reference to a label declared in the same section. The label can be set either a **temporary** label or **permanent** label.

7. To ensure full checking of relative branches and calls, the ***brr*** and ***callr*** instructions can branch to a (relative) address only within the current section. A smart branch or call instruction, ***sbr*** or ***scall***, can be used to let the Assembler generate a relative address if possible, and an absolute address if not.

8. Comments can be added in 2 ways: on a single line (for example: after an instruction), by preceding it with a semicolon (;), or on multiple lines as in C programs (/* comment */).

9, Handling Immediate values. The Assembler considers all immediate values as 32 bit size values. When assigning an 'imm16' argument for instance, a 0x8000 will be considered a positive number – 0x0000_8000.

Extended syntax:

1. The Assembler supports a C-style bkrep syntax, that is, the repeated block is surrounded with braces ('{' ....'}'). Therefore there's no need to specify the end address by a label.

Examples:

- bkrep #6
  {
  mov (r0), r3 || r0+
  push r3
  }

4.9 Labels

Permanent and Temporary Labels

Labels can be either **permanent** or **temporary**. Small differences exist in terms of syntax. To define a **temporary** label, just precede the label name with a % character. To refer to a **temporary** label, precede the label name with either >% (forward reference) or <% (backward reference). In terms of usage, **temporary** and **permanent** labels are quite different: each **permanent** label must be unique in a section or a project, whereas **temporary** labels can appear multiple times in the same module, and even in a particular section. A reference to a **temporary** label will always be the closest **temporary** label definition in either the forward or backward direction depending on the syntax. **Temporary** labels are not visible to the Linker, i.e. they are always local. By default, **permanent** labels are always defined to be local (to the module), unless declared **.PUBLIC** or **.EXTERN** or **.GLOBAL**. In terms of debugging, temporary labels are invisible inside the Debugger, whereas **permanent** labels are visible. **Temporary** labels can be used when the flow of the program is consistently split into two distinct cases and the programmer would like to assign the same label name to each case for ease of understanding. For example: the following program has two pieces of code (case1 and case2) that need to be treated differently depending on whether a number is even or odd:

```
case1:... ; check if number is even or odd
        sbr>%even, neq ; forward reference
        ... ; odd treatment
        sbr case2 ;smart branch to second case
    • %even: ... ; even treatment
        ...
    •
case2: ... ; check if number is even or odd
        sbr >%even, neq ; forward reference
        ... ; odd treatment
```

sbr continue

- %even: ... ; even treatment

...

continue: ...

Note that no conflict exists between the first and second occurrence of the %even: label.**Full Name vs. Simple Labels**

For compatibility reasons with previous versions of the Assembler and Linker, **permanent** labels are of either two types: **full name** or **simple**. The type depends on the section in which they are defined.

Labels that are defined in a **.CODE** or **.DATA** sections, are **full name** labels. They inherit the name of that section as a prefix to the label, separated with the dot (.) operator. A **reference** to such a label needs the specification of the full name, i.e. **section_name.label_name**.
Inside a **.CODE** or **.DATA** section, if a **reference** is made to a label, it is automatically prefixed with the current section name, unless an explicit section name is prefixed by the programmer. Each time a new **.CODE** or **.DATA** section is declared, a new prefix is active which is automatically given to label definitions and added to label references in that section.
By using the **.USE** directive, the default prefix to label references can be changed until a new **.CODE** or **.DATA** section is declared, or a new **.USE** declaration is made.
The directives **.PUSHSEC** and **.POPSEC** can be used inside **.CODE** and **.DATA** sections, to temporarily change the active section for prefixing, and restoring the previous active section, without having to remember the section name. This is useful for macros that can be invoked in other sections. Note that **full name** labels can have the same label name in the same module if they are defined in different sections, due to their different section prefix (the section to which the label belongs).

The **simple** labels on the other hand, are labels that are defined in the **.CSECT** or **.DSECT** sections. They must have a unique name across both program (code) and data memory spaces. By default, they are local, unless the **.PUBLIC** / **.GLOBAL** directive is used to declare them global, in which case they may not be redefined in any other section or module. Simple labels do not inherent any section prefix and are referenced just by their name.

To refer to a **full name** label inside a .CSECT or .DSECT section, just add the appropriate section prefix. To refer to a **simple** label from within a **.CODE** or **.DATA** section, you must instruct the Assembler not to automatically add the section prefix by first issuing a **.USE** 0 command. In this case, this mode will be in effect until a new **.CODE**, **.DATA** or **.USE** directive is used. Refer to the Assembler Directives topic for the various directive details.

4.10 Arithmetic and Logical Operators

The arithmetic and logical operators are a subset of the C language operators. All of the operators are effective at assembly time on resolvable constants. The order of expression evaluation is the same as in C.

Following are the supported operators:

- + Addition operator
- - Subtraction operator
- / Integer division operator
- * Multiplication operator
- **%** Modulo operator
- **&&** Logical-And operator
- || Logical-Or operator
- **&** Bit-And operator
- | Bit-Or operator
- ^ Bit-Xor operator
- >> Arithm. shift-right operator (sign ext)
- << Bit shift-left operator
- **Unary** + Positive operator
- **Unary** - Sign change operator
- **Unary** ~ Bit complement operator
- **Unary !** Logical not operator
- **(expr)** Group operator
- == Equal test operator
- **!**= Not equal test operator
- >= Greater than or equal test operator
- > Greater than test operator
- <= Less than or equal test operator
- < Less than test operator
- **expr ? v1 : v2** Conditional operator (if expr then v1, else v2)

Note: ***v1, v2*** *must be number expressions.*

4.11 Assembler Operators

4.11.1. # (Hash)

The short immediate operator is both an assembly-time and link-time operator which accepts either a short (9-bits or less) value, or a long (16-bit) value.

For instructions that accept both a short and a long immediate operand, if a long (16-bit) value is used with this operator, the Assembler will automatically select the long instruction format; however, if the instruction does not support the long immediate operand, an error will be reported.

For instructions that accept only a long immediate operand, the Assembler will automatically convert the value to long immediate format.

Note: *Automatic conversion from short to long immediate format can be shutdown by the* ***–dontExpandShort*** *command line option..*

Examples:

```
mov #0, r0 ; uses short immediate format (1 word instruction)

mov #0x1000, r1 ; converts to long immediate format (2 word instruction)
```

4.11.2. ## (Double Hash)

The long immediate operator is both an assembly time and link-time operator which accepts a word (16-bit) value.

Example:

```
mov ##label+1, r0
```

4.11.3. $ (Dollar)

The location counter operator is a link-time operator representing the next instruction address. An operand expression may not include both a label and a $ together.
Note that the $ operator is used also in another context of meta-sections. For more information about meta-sections see the Assembler's Meta-Sections chapter.

Example:

```
brr $-1 ; branch to self

mov #$+label, r0 ; illegal
```

4.11.4. : (Colon)

The (:) is the **permanent** label definition operator. It is both an assembly time and link-time operator. Labels can be either **full name** or **simple**, depending on the current active section type in which they are defined. In **.CODE** and **.DATA** sections, labels are defined as full name, otherwise they are **simple**. A **full name** label consists of a symbol preceded by a section name. The offset name is the symbolic name of the label with respect to the current section. **Note** that the directives **.USE**, **.PUSHSEC** and **.POPSEC**, have no effect on the current section. The unabbreviated **reference** to this label in a **.CODE** or **.DATA** section is **CurrentSectionName.OffsetName**. The abbreviated **reference** to this label is **OffsetName**. It is affected by the use of the directives **.USE**, **.PUSHSEC** and **.POPSEC.** Refer to the Labels topic for details.

Note: *The trailing colon (:) is mandatory. Maximum length of an offset name is restricted to 31 characters. It must start in the left-most column with a letter (lower or upper case) and may include any letters, digits or underscore (_) symbols. Preceding as well as trailing blanks and tabs are ignored. Labels are case sensitive. All instruction and register mnemonics are reserved words that may not be used for labels.*

Example:

```
My_Label2:  ; permanent label definition
        mov #0, y

        brr My_Label2, eq ; permanent label reference
```

4.11.5. % (Percent)

The (%...:) is the **temporary** label definition operator. The scope of a temporary label is only within the current section, i.e., the current instance of the current section. The current section terminates with the **.CODE**, **.DATA**, **.CSECT**, **.DSECT** or **.ORG** directives. **Temporary** labels can be used in **bkrep** instructions, and with all other branching instructions. For example: when multiple program pieces exist that are branched to the same cause or condition using the same label, it makes the code more readable.

Note: *The Debugger does not disassemble the **temporary** labels.*

4.11.6. . (Dot)

The (.) dot operator is used as follows:

- As section prefix operator, allows for fully specified **"full name" permanent** label references. It is an assembly-time operator. Offset names are only unique within sections, i.e., the same offset name may be defined in many sections. An offset name is unique only when prefixed with its section name. By default, **full name** label references (label references inside **.CODE** or **.DATA** sections) that do not contain an explicit section prefix, use the prefix of the current **.CODE** or **.DATA** section, i.e., the section where the label is referenced (and not necessarily where it is defined). **.USE**, **.PUSHSEC** and **.POPSEC** directives affect the default behavior of not prefixed **full name** type of labels (Refer to **Labels** topic). The dot operator cannot be used with **simple** labels.
- As an offset from the start of a section operator, allows indicating an offset from the start of the section (that is used on the left side of the dot operator). An immediate number can be used for offsets (might be useful for addressing arrays).
- As a notation for directives, allows the definition of Assembler directives, which are preceded by the dot operator.

Examples:

```
mov #SegName.OffsetName1, r0

mov #SegName.15, r0

.CODE section
```

Note the right way of referencing the base address of a section is **SecName.0**, where only a section name is required as an external reference (Refer to **Macro Preprocessor Operators** for details) and no offset names are required. **SecName.0** must be declared as external, for example:

```
.EXTERN SecName
```

ven though the Assembler does not currently support a type definition or structure definition directive with the use of simple macros, one level structures can be defined. This can easily be used to declare the same structure in multiple sections (Refer also Programming Hints - Data Structures).

4.11.7. @ (At)

The (@) at operator, or modulo-256 operator, allows for automatic label references in Direct Addressing mode. This is a link-time operator. Symbol names preceded with this operator will be treated as Direct memory addresses by the Linker, i.e. their final (relocated) address will be truncated to 8 bits by modulo 256 operation on the address value.

Examples:

```
mov @SecName.OffsetName1, r0
mov a0h, @SecName.Label
>%Label and <%Label Operators
```

The above operators are the forward and backward **temporary label reference** operators. The closest forward or backward reference is used. **Temporary** label references cannot be prefixed with a section name. The scope of a temporary label is only within the current section. The current section terminates with a new **.CODE**, **.DATA**, **.CSECT** , **.DSECT** or **.ORG** directives.

Notes:
(1) By default, references to **temporary** labels are assumed to be forward.

Examples:

```
        add r0, a0
        br >%Ok, lt ; forwards
        ...
%Ok:
        add r1, a0
        brr <%Ok, eq ; backwards


        bkrep #99,>%Loop ; 100 times loop
        ...
        %Loop: nop
```

4.11.8. CODESEGMENT

CODESEGMENT (section_name) OPERATOR

The CODESEGMENT(section_name) operator calculates the code (memory) segment index of the section_name.

Example:

```
mov    #CODESEGMENT(ccc1), a0l
```

Note:

If section_name is mapped in multiple segments then applying

CODESEGMENT(section_name) operator returns the smallest segment index.

4.11.9. CURRCODESEG

The **CURRCODESEG** operator returns the code (memory) segment index of the current section.

Example:

```
mov    #CURRCODESEG, a1
```

4.11.10. DATASEGMENT

DATASEGMENT (section_name) OPERATOR

The **DATASEGMENT (section_name)** operator calculates the data (memory) segment index of the **section_name**.

Example:

```
mov    #DATASEGMENT(ddd1), a0l
```

Note:

If **section_name** is mapped in multiple segments then applying

CODESEGMENT(section_name) operator returns the smallest segment index.

4.11.11. FRACT

FRACT (number,bits) OPERATOR

The **FRACT** operator is an assembly-time operator that calculates the 16 bit integer value representing the floating point operand in a user supplied fractional representation. The fractional representation is specified by indicating the number of bits to the right of the floating point.

Examples:

mov ##**FRACT** (0.25, 15), r0 ; translates to mov ##0x2000, r0

mov ##**FRACT** (3.5, 12), r1 ; translates to mov ##0x3800, r1

In the first example, 1 bit is used for the sign of the number and 15 bits are allocated for representing the fraction 0.25. In this format, the range of values that can be used is from -1.0 to +1.0 (not including the limits). In the second example, 1 bit is used for the sign of the number, 3 bits are allocated for the integer part of the number and 12 bits are used for the fraction part of the number. For more examples and programming hints, see also the section **Fractional Arithmetic Support**.

4.11.12. HIGHADDR

HIGHADDR (symbol)

Calculates the high bits (over bit #15) of the **symbol's** address (16-17 for CEVA-Teak, and 16-31 for CEVA-X & CEVA-TeakLite-III & CEVA-XC).

Example:

```
mov #LOWADDR (label), r5
load #HIGHADDR (label), movpd          ; for CEVA-Teak
rep #70
        movp (r5), (r1) || r5+1, r1+1
```

4.11.13. IMMEDOFFSET

The immediate offset operator is an assembly-time operator that calculates the offset of a label from within the section in which it is defined. The label cannot be an external or forward reference. This operator can effectively be used to calculate the number of consecutive variables defined in a long data section. Refer to the **DIFF Equate** topic for more examples on how to use this operator.

Example:

```
mov #IMMEDOFFSET section.Offset, r0
```

Note: Refer to the link time **OFFSET** operator

4.11.14. INCODE

The **INCODE** operator is a link-time operator. It precedes a label and instructs the Linker to use the address of the label residing in the program space. This operator is used when creating downloadable programs. In this case, a section might be mapped in both the program and data space. Labels in this section are thus defined twice, in both the data space and program space. By default, all label references used in **bkrep**, **call** type and **branch** type instructions refer to the program space and all other label references are assumed to point into the data space. In the case where a downloadable program is created, a **label reference** used in an instruction that is not one of the above might be a reference into the data space, in which case it must be preceded with the operator **INCODE**, in order to indicate specifically the reference to the program space.

Note: *When the section is mapped only in one memory space there is no need to use the **INCODE**. That is the **INCODE** directive is useful only when a section is mapped both in program and data memories.*

Example:

```
mov ## INCODE MyTable, r4 ; take address of MyTable in the Program space
movp (r4)+,(r0)+
```

4.11.15. LOWADDR

LOWADDR (symbol)

Calculate the 16 low (0-15) bits of the **symbol's** address (for **CEVA-Teak**, and 16-31 for CEVA-X & CEVA-XC & CEVA-TeakLite-III)

Example:
See in **HIGHADDR (symbol)** operator.

4.11.16. OFFSET

The **OFFSET** operator is a link-time operator that calculates the offset of a label from within the section in which it is defined. If the label on which this operator is used resides in a data section that is located on a page boundary (specified at link time), this operator can be used for direct addressing mode. Upon linking, the object code that corresponds to the label is patched to reflect the distance between the final label address and the final address of the start of the section in which this label is defined.

Example:

```
mov #OFFSET Section.Variable, r1
```

Refer to **Direct Memory Addressing Support** topic for more examples using this operator and the way it can be used for direct memory addressing.

Note: *Taking into account the fact that **OFFSET** is a link-time operator that operates after all same name sections are appended into one block and **IMMEDOFFSET** is an assembly time operator, it is obvious that the result of the **OFFSET** operator cannot be smaller than the **IMMEDOFFSET**'s result. Both operators generate the same results in cases where are no multiple sections.*

4.11.17. PG

The **PG** (Symbol) link-time operator finds the 256-word memory page of the symbol. It is equivalent to using **SHR(Symbol, 8)**. This link time operator enables the programmer to load the processor's page register with the ***lpg*** instruction, without worrying where the symbol will be eventually located by the Linker.

Example:

```
lpg #PG ( Section.Offset )
```

Notes:

1. *For trouble free data memory accesses using the efficient short direct addressing mode, the programmer must guarantee the following via the Linker:*
   - *The data sections are aligned on page boundaries (using the align Linker option).*
   - *The data sections do not exceed the physical page size of 256 words (using the inpage Linker option).*

   If these conditions are not met, the programmer must change the page bits each time according to the data variable being accessed. Refer to ***Direct Memory Addressing Support*** *topic for more details and examples*
2. *The PG operator can be used with arithmetic expressions of the type +const or -const.*

4.11.18. SHR

The **SHR(Symbol, nBits)** link-time shift-right operator executes a bit shift-right. Symbol is the section in **full name** notation and **nBits** specifies the number of bits to be right shifted. **SHR** differs from **"**>>" which is an arithmetic shift-right assembly-time operator. This link time operation allows efficient loading of the core's page register via the lpg #immediate instruction.

Example:

lpg #**SHR** (Section.Offset, 8)

Notes:

1. *The programmer must align the data sections on the proper page boundary.*
2. *nBits must comply with:*
   1<= nBits <= 15 for CEVA-TeakLite and CEVA-Teaklite-II
   1<= nBits <= 17 for CEVA-Teak
3. *Refer to **Direct Memory Addressing Support** for more examples and the way this operator can be used for efficient direct memory addressing as well as how the PG operator can be used for the same purpose*

4.11.19. SIZEOF

The **SIZEOF(SectionName)** link-time operator calculates the size of a section in **words**. This link time operator helps creating efficient code when the size of a section is a parameter. Examples are initialization programs that need to initialize all the variables allocated to a particular data section.

Example:

```
clr a0
rep #SIZEOF( MyData )-1
mov a0l, (r1)+
```

4.12 Assembler Directives

Assembler directives purpose is to provide program data and control instructions to the assembly process. They allow partitioning of code and data into sections, allocation and initialization of memory, definition of global variables, conditional assembly and control of the appearance of the listing. For all directives (except the **DW**, **sbr** and **scall** directives) the first non-blank character of the line must be a dot (.). Assembler directives are case non-sensitive (as opposed to the instruction mnemonics that have to be in lower case only). Following is a list of Assembler directives supported by CEVA DSP**s** for use by the assembly language programmer.

Notes: *1. **Appendix 6.1** lists all the reserved words that are used as internal directives; these internal directives are used by the **CEVA-Toolbox C/C++ Compiler** and the **MPP** to pass debugging and listing information*

4.12.1. .CODE

.CODE [SectionName]

.CODE [MetaSection$SectionName]

The **.CODE** directive defines the start of a code section. If no section name is supplied, the default code section "CODE" is used. The .CODE directive creates a new code section that can be linked by the Linker with other **.CODE**, **.CSECT**, **.DATA** or **.DSECT** sections into the processor's program (code) or data memory space. When a new code section of this type is created, the previous temporary symbol table is deleted, and the current **.USE** section name is set to the argument of the .CODE directive. All labels defined in the newly created code section are of type **full name**. This means that the section name is implicitly attached to all the labels, and that to refer to such a label, it is necessary to specify the full name (for example: SectionName.Label).

For more information about meta-sections see the Assembler's Meta sections 4.5.

Example:

.CODE MyCodSeg

Notes:

1. *By default, the **DW** (Data Word) directive described below can be used inside a **.CODE** section. To create a table of constants in the program space, link an appropriate **.DATA** or **.DSECT** section into program memory space, by specifying the **.DATA** or **.DSECT** section name in the Linker script file together with the program's code sections. The **-noDataInCode** Assembler command line switch may be used to generate an error in case data sections are declared within code sections.*
2. ***.CODE** section names are by default **PUBLIC** and must be unique across all modules and all code and data sections, regardless of whether the section is only used privately within a module*
3. ***.CODE** sections may be split within a particular module or in different modules, creating multiple sections of this section. These sections are then glued together at link time to form one code section according to the linking algorithm (see sections Linking Algorithm and Multiple Section Definitions for more details).*

4.12.2. .CSECT

.CSECT SectionName

.CSECT MetaSection$SectionName

The **.CSECT** directive defines the start of a code section. The **.CSECT** directive creates a new code section that can be linked by the Linker with other **.CODE**, **.CSECT**, **.DATA** or **.DSECT** sections into the processor's program (code) or data memory space. When a new code section is created, the previous temporary symbol table is deleted. All labels defined in the newly created section are of type **simple**. This means that no section name is attached to labels, and that to refer to such a label, it is enough to specify the label name. **.CSECT** directive must be followed by a section name.

For more information about meta-sections see the Assembler's Meta-Sections 4.5.

Example:

```
.CSECT MyCode
```

Notes:

1. *By default, the **DW** (Data Word) directive described below can be used inside a **.CSECT** section. To create a table of constants in the program space, link an appropriate **.DATA** or **.DSECT** section into the program memory space, by specifying the **.DATA** or **.DSECT** section name in the Linker script file together with the program's code sections. The **-noDataInCode** Assembler command line switch may be used to generate an error in case data sections are declared within code sections.*
2. ***.CSECT** sections may be split within a particular module, or in different modules, creating multiple sections of this section. These sections are then glued together at link time to form one code section according to the linking algorithm (Refer to **Linking Algorithm** and **Multiple Section Definitions** for details).*
3. ***.CSECT** section is generated by the C/C++ Compiler for program code section allocation*

4.12.3. .DATA

.DATA [SectionName]

.DATA [MetaSection$SectionName]

The **.DATA** directive defines the start of a new data section. If no section name is supplied, the default data section **DATA** is used. The DATA sections contain data definitions (not instructions). By default, the Linker maps the **.DATA** sections into the processor's data space regardless of whether it contains initialized data. The Linker can be instructed, however, to map **.DATA** sections into the processor's program (code) space, for example: to include constant tables or filter coefficients. When a new code section of type **.DATA** is created, the former temporary symbol table is deleted, and the current **.USE** section name is set to the argument of the **.DATA** directive. All **permanent** labels defined in a **.DATA** section are of **full name** type, as explained in **Labels** topic; i.e. they should be referenced by specifying (implicitly or explicitly) **SectionName.LabelName**. For more information about meta-sections see the Assembler's Meta-Sections 4.5.

Example:

.DATA MyDatSeg

Notes:

1. *Section names are by default **PUBLIC** and must be unique across all modules and all code and data sections, regardless of whether the section is only used privately within a module.*
2. *Data sections may be split within a particular module, creating multiple sections of this section. These sections are then glued together at link time to form one data section according to the linking algorithm (Refer to **Linking Algorithm** and **Multiple Section Definitions** for details).*

4.12.4. .DSECT

.DSECT SectionName

.DSECT MetaSection$SectionName

The **.DSECT** directive defines the start of a new data section. By default, the Linker maps the **.DSECT** sections into the processor's data space regardless of whether it contains initialized data. The Linker can be instructed, however, to map **.DSECT** sections into the processor's program (code) space, for example: to include constant tables or filter coefficients. When a new code section of type **.DSECT** is created, the former temporary symbol table is deleted, and all permanent labels defined in the **.DSECT** section are of **simple** type, as explained in **Instruction Set Syntax** topic, i.e. they are referenced by specifying just their name. The **.DSECT** directive must be followed by a section name. For more information about meta-sections see the Assembler's Meta-Sections 4.5.

Example:

.DSECT MyData

Notes:

1. Section names are by default **PUBLIC** and must be unique across all modules and all code and data sections, regardless of whether the section is only used privately within a module.

2. **.DSECT** data sections may be split within a particular module, creating multiple sections of a section. These sections are then glued together at link time to form one data section according to the linking algorithm (Refer to **Linking Algorithm** and **Multiple Section Definitions** for details).

3. **.DSECT** section is generated by the C/C++ Compiler for program data section allocation.

4.12.5. DB

DB DataValue [,DataValue [,DataValue...]]

DB NumericExpression DUP DataValue

The **DB** directive allocates one or more data bytes that may be initialized. The **DB** directive is used like an instruction, in the sense that it is the only directive that is not prefixed by a dot (.). **DB** directive must contain a list of one or more data values. Data values may be a numeric expression or uninitialized. Uninitialized values are signified by a "?". Internally, uninitialized values are stored as zeros. **DB** directive may only be used in a data section. The **DUP** operator may be used to repeat the initialization value.

Notes:

1. **DB** directive can be used only in byte addressable cores.
2. Even though it is not possible to create uninitialized data sections, it is possible to ask the Debugger not to load a particular data section during the load process. Moreover, in **simulation** mode, it is also possible to detect read operations from data memory that was not loaded and therefore not initialized. This is achieved by instructing the Linker to mark the data sections as **noload** and the Debugger to enable uninitialized memory checks. Refer to Linker Script File chapter and the Debugger Reference Guide for more details.

Examples:

```
DB ?

DB 1,2,3

DB 3 DUP ? ; assign ?,?,?

DB 2 DUP 5 ; assign 5,5

DB "abcd" ; assign 0x61,0x62,0x63,0x64
```

4.12.6. DW

DW DataValue [,DataValue [,DataValue...]]

DW NumericExpression DUP DataValue

The **DW** directive allocates one or more data words that may be initialized. The **DW** directive is used like an instruction, in the sense that it is the only directive that is not prefixed by a dot (.). **DW** directive must contain a list of one or more data values. Data values may be a numeric expression, a symbolic expression or uninitialized. Uninitialized values are signified by a "?". Internally, uninitialized values are stored as zeros. **DW** directive may only be used in a data section and, like instructions, may be preceded by a label. The **DUP** operator may be used to repeat the initialization value.

Note:

Even though it is not possible to create uninitialized data sections, it is possible to ask the Debugger not to load a particular data section during the load process. Moreover, in **simulation** mode, it is also possible to detect read operations from data memory that was not loaded and therefore not initialized. This is achieved by instructing the Linker to mark the data sections as **noload** and the Debugger to enable uninitialized memory checks. Refer to Linker Script File chapter and the Debugger Reference Guide for more details.

Examples:

```
DW ?

DW 1,2,3

DW 3 DUP ? ; assign ?,?,?

DW 2 DUP 5 ; assign 5,5


DW "abcd" ; assign 0x6162,0x6364

PermLabel:
DW PermLabel
```

4.12.7. DD

DD DataValue [,DataValue [,DataValue...]]

DD NumericExpression DUP DataValue

The **DD** directive allocates two or more data words that may be initialized. The **DD** directive is used like an instruction, in the sense that it is the only directive that is not prefixed by a dot (.). **DD** directive must contain a list of one or more data values. Data values may be a numeric expression, a symbolic expression or uninitialized. Uninitialized values are signified by a "?". Internally, uninitialized values are stored as zeros. **DD** directive may only be used in a data section and, like instructions, may be preceded by a label. The **DUP** operator may be used to repeat the initialization value.

Note:

Even though it is not possible to create uninitialized data sections, it is possible to ask the Debugger not to load a particular data section during the load process. Moreover, in **simulation** mode, it is also possible to detect read operations from data memory that was not loaded and therefore not initialized. This is achieved by instructing the Linker to mark the data sections as **noload** and the Debugger to enable uninitialized memory checks. Refer to Linker Script File chapter and the Debugger Reference Guide for more details.

Examples:

```
DD ?

DD 1,2,3

DD 3 DUP ? ; assign ?,?,?

DD 2 DUP 5 ; assign 5,5


DD "abcd" ; assign 0x61626364

PermLabel:
DW PermLabel
```

4.12.8. .EXTERN

.EXTERN Symbol1 [,Symbol2 [,Symbol3...]]

The **.EXTERN** directive allows the usage of symbols defined in another (external) assembly module (file). The Linker resolves these symbols. **Full name** labels should be fully specified, i.e., prefixed with the appropriate section name, unless the **.USE** directive is used. **Simple** labels do not require any prefix. In order to declare a symbol External (no prefix), one must use the **.USE** directive (refer to **.USE**).

Examples:

.EXTERN Sec1.Offset2, Sec.Offset3

.USE Sec2

.EXTERN Offset4, Offset5

4.12.9. .FF

The **.FF** is the form feed directive. It causes the start of a new page when the listing is printed.

4.12.10. .FORMAT

.FORMAT LinesPerPage [,CharactersPerLine]

The **.FORMAT** is the format directive. It causes the listing to be formatted according to the specified values. By default, 60 lines per page and 80 characters per line are created. **Note:** It is possible to use the **-format** Assembler invocation switch (see **Error! Reference source not found.**) as a substitute to the **.FORMAT** directive (applicable to SDT V8.4 and higher).

4.12.11. .FUNC_START/END

.FUNC_ START/END N FunctionName

The .FUNC_START and .FUNC_END directives are used for marking function's start and end. These Assembler directives are used by the Profiler for generating functions' profiling information.

These directives can also be used for hand written assembly routines, which the user wishes to include in the profiling report (see the Debugger and Compiler Reference Guides chapters for more details on profiling).

These directives are also used by the Complier by default. This allows the Debugger to provide profiling information for C source files complied without debug info.

The number N is the profiling level (default assembly level is 3). The profiling level definitions are:

1. For file-io library functions
2. For regular library functions
3. For source file functions
4. N >= 4, for user-defined level, designed to allow the user the capability to profile only designated functions

These levels can be used as a filter for the Profiler when running the Debugger Profiler (For more information see the Debugger Reference Guide '**show memory profiler'** CLI**).**

Syntax:

.func_start N <fname> - where N is the profiling level and <fname> is the function name

.func_end N <fname> - where N is the profiling level and <fname> is the function name

Example:

```
.text
.func_start 1 _foo     ; mark function _foo start.
foo
add #0x100,a1
....
ret {t}
```

```
.func_end 1 _foo      ; mark function _foo end.
```

4.12.12. .GLOBAL

.GLOBAL Symbol1 [,Symbol2 [,Symbol3...]]

The **.GLOBAL** directive combines the **.EXTERN** and **.PUBLIC** directives, i.e., it can be used to specify symbols that will be imported from external modules and/or symbols that will be exported to other modules. It is useful for header files, since the same directive can be used for both the importing and exporting module. The disadvantage of this directive is that it could lead to multiple definitions of the same symbol in more than one module, which would cause an unresolved Linker error.

Example:

.GLOBAL Sec1.Offset1, Sec1.Offset2, Sec2.Offset3

4.12.13. .INPAGE

This directive passes the inpage check request to the Linker so that in linking time each section that contains the **.INPAGE** directive will be treated as if a Linker's attribute was written next to it in the Linker script file. This directive enables the enforcement of 'inpage' checking on a certain section independently of the Linker's script file. For more details on the Linker's attribute, see the Assembler & Linker's Guide.

Example:

```
.DSECT data_sec
.INPAGE
lebel: DW 8
```

Note: *The .INPAGE directive is legal only inside the scope of data sections (.DATA / .DSECT). When used inside a code section (.CODE / .CSECT), a warning is generated and the directive is ignored.*

4.12.14. .LIST

.LIST BooleanNumericExpression

The **.LIST** directive is used to switch source listing generation on and off. The default is 1 (on).

4.12.15. .LOAD

.LOAD RegisterName DataValue

.LOAD RegisterName FieldsValue

CEVA-Teak Only!

The **.LOAD** directive supports programming the Addressing Mode Registers in the **CEVA-Teak** only. This is an Assembler time directive. Address Mode registers include **ar0-ar1** and **arp0-arp3**. The use of this directive is to indicate to the Assembler the current contents of these registers for the purpose of restriction checking later in specific instructions that use their contents (**rJ** and **rI** pointers). It is the responsibility of the programmer to load these registers (using the **mov** instructions) or to use their default values.

The use of the **.LOAD** directive comes in 2 formats as the following example indicates:

.LOAD arp0 0x61b0 ;arp0 -> rJ=r7, rI=r0, consecJ=+1
; pmJ=-2, consecI=-1, pmI=0

;or in symbol format

.LOAD arp0 r7, r0, +1, -2, -1, 0; equivalent to the above
mpy (r7), (r0) || mpy (r7+),(r0-) ||add3 p0,p1,a1 || r7-2

Notes:

1. Assembler should be invoked with **-script load.pl** option when using the **.LOAD RegisterName FieldsValue** format.
2. Refer to .RLOAD
3. .RLOAD RegisterName DataValue
   .RLOAD RegisterName FieldsValue directive

4.12.16. .ORG

.ORG NumericExpression

.ORG $+NumericExpression

The **.ORG** directive sets the location counter of the current section to the value specified by the argument. The **.ORG** directive creates a new section with the same name and type as the current section. Internally, a **.CODE**, **.DATA**, .**CSECT** or **.DSECT** directive is generated. When the object file is linked, the Linker locates the section relative to the starting address of the section as specified in the Linker script file. Effectively, the **.ORG** directive thus creates a "hole" inside the appropriate section block that cannot be used by any other section.

Example:

.CODE ccc

.ORG 0x15

Label: nop ; will be at 0x115 if Linker is asked to put section ccc at 0x100

4.12.17. .POPSEC

The **.POPSEC** directive sets (restores) the current **.USE** section (of type **.CODE** or **.DATA**) with the value obtained by popping the top entry from a section name stack. In conjunction with the **.PUSHSEC** directive, this directive is useful for writing nested macros that, when finished, will not have any effect on the current **.USE** section.
Using of this directive inside **.CSECT** and **.DSECT** sections is not allowed (i.e., allowed only in **.CODE** and **.DATA** sections).

Example:
See example at part Safe Macros Using .PUSHSEC and .POPSEC section

4.12.18. .PUBLIC

.PUBLIC Symbol1 [,Symbol2 [,Symbol3...]]

The **.PUBLIC** directive is used to declare the symbols in its argument list as being exportable to other modules. These symbols can be declared in other modules with either the **.EXTERN** or **.GLOBAL** directives, and the Linker will be able to resolve them. Like the **.EXTERN** and **.GLOBAL** directives, unqualified symbol names, i.e., symbol names not prefixed by a section name, will use the current .**USE** section name by default, unless a **.USE 0** statement was issued. Without a **.PUBLIC** declaration, symbols are local to the module (file) and cannot be referenced by other modules.

4.12.19. .PUBLICSEC

.PUBLICSEC <Sections list>

The **.PUBLICSEC** directive is used to declare all the labels appearing in the sections list as public.

Note: Sections list should contain at least one section

4.12.20. .PUSHSEC

The **.PUSHSEC** directive pushes the current section name (defined by the last **.CODE**, **.DATA or .USE**) onto the top of the section name stack. In conjunction with the **.POPSEC** directive, this directive is very useful for writing nested macros that, when finished, will have no effect on the current **.USE** section. Use of this directive inside **.CSECT** and **.DSECT** sections is not allowed (i.e., allowed only in **.CODE** and **.DATA** sections).

Example:

See example at part Safe Macros Using .PUSHSEC and .POPSEC section.

4.12.21. .RLOAD

.RLOAD RegisterName DataValue

.RLOAD RegisterName FieldsValue

CEVA-Teak Only!

The **.RLOAD** directive supports programming the Addressing Mode Registers in the CEVA-Teak only. This is an Assembler time directive. Address Mode registers includes **ar0-ar1** and **arp0-arp3**. The purpose of this directive is to indicate the Assembler the current contents of these registers for the purpose of restriction checking later in specific instructions that use their contents (**rJ** and **rI** pointers). Unlike the **.LOAD**, this directive inserts the proper **mov to Address Mode registers** instruction to the program, so it frees the programmer of the responsibility to load these registers (using the ***mov*** instructions).

The use of the **.RLOAD** directive comes in 2 formats as the following example indicates:

```
.RLOAD arp0 r7, r0, -1, +s, +1, +s   ; symbolic format
                                     ; ar0 -> rJ=r7, rI=0, consecjJ=-1,
                                     ; pmJ=+consecI=+1, pmI=+s

 nop
 mpy (r7), (r0) || mpy (r7-), (r0+) ||
 add3 p0, p1, a1 || r7+s, r0+s


.RLOAD arp0 ##0x626b                 ; DataValue format
                                     ; ar0 -> rJ=r7, rI=0, consecjJ=-1,
                                     ; pmJ=+consecI=+1, pmI=+s
nop
```

The Assembler inserts **mov ##0x626b, arp0** preceding the ***nop*** instruction

Notes:

The Assembler should be invoked with the **-script load.pl** option when using the **.RLOAD RegisterName DataValue/FieldsValue** formats.

Refer to .LOAD RegisterName FieldsValue directive

4.12.22. .IGNORE_WARNINGS

.IGNORE_WARNINGS W1,W2,…,Wn (Where w1-wN are warning numbers).

The IGNORE_WARNINGS directive disables generation of warning messages specified in the list. Can be used to avoid warning messages that are irrelevant for the assembly code. Similar to the C/C++ preprocessor operator #pragma warning(disable:N)

Example:

.INCLUDE "init_chip.asm"

.IGNORE_WARNINGS 387,386,6

CEVA-X and CEVA-XC Only!

4.12.23. .INLINE

Make sure the next instruction packet does not cross fetch line boundaries. A fetch line size is 256 bits.

Notes:

1. The .INLINE directive is allowed for usage only within a code type section: .CSECT/.CODE
2. The Assembler automatically sets the .INLINE directive before ‘bkrep2i’ instructions

4.12.24. .NEWLINE

Make sure the next instruction packet starts at the beginning of a fetch line. A fetch line size is 256 bits.

Note:

The .NEWLINE directive is allowed for usage only within a code type section: .CSECT/.CODE

4.12.25. sbr

The **sbr** directive can be used as a replacement for the ***br***/***brr*** instructions. The Assembler will replace the ***sbr*** directive by either the ***br*** instruction, or the ***brr*** instruction, according to the distance of the label from the branch instruction.

The smart branch (**sbr**) directive is a convenient way to pass the burden of deciding whether to use **br** or **brr** to the Assembler. The Assembler replaces **sbr** by either **br** or **brr**.

Notes:

1. The offset range of the **brr** is –63 to 64
2. **sbr** accepts only labels referencing addresses - no number strings accepted.

Example:

```
label:
        nop
        nop
        sbr label ; Since the label is near will be converted to the brr label
```

4.12.26. scall

The **scall** directive can be used as a replacement for the ***call***/***callr*** instructions (see note below). The Assembler will replace the **scall** directive by either the ***callr*** instruction, or the **call** instruction (in this sequence), according to the distance of the label from the call instruction.

The smart call (**scall**) directive is a convenient way to pass the burden of deciding whether to use ***call*** or ***callr*** to the Assembler. The Assembler replaces **scall** by either ***call*** or ***callr***.

Example:

```
label:
        ret

label1:
        nop
        nop
        scall label ; Since label is near will be converted to callr label for all CEVA DSPs
```

Notes:

scall *accept only labels referencing addresses - Number strings are not accepted.*

4.12.27. .TITLE

```
.TITLE "text"
```

The **.TITLE** is the print out Title directive. Each new page following this directive will have the title printed on the top of page, under predetermined header (company logo and version number), the date and time of printing and the page number.

4.12.28. .USE

.USE [SectionName]

The .USE directive specifies the current section name to be used when encountering an unqualified **permanent** symbol reference in a **.CODE** or **.DATA** section. When no argument is specified, the name of the current section is used. The current **.USE** section name is affected by the following directives: **.CODE**, **.DATA**, **.ORG**, and **.POPSEC**. It is invalid when used in a **.CSECT** or **.DSECT** sections.

Notes:

1. **.USE 0** is the default value until changed as explained above
2. When an instruction is located in a **.CODE** section refers to a **simple** label defined in a **.DSECT** or **.CSECT** sections, the instruction must be preceded by a **.USE 0** statement to prevent prefixing of the current section name. This will be in effect until another **.USE** is declared.
3. Refer to .PUSHSEC and .POPSEC directives.
4. Refer to Safe Macros Using .PUSHSEC and .POPSEC section.

4.13 Programming Address Mode Registers Warnings

For CEVA-Teak core only

From SDT V9.1 the Teak Assembler issues warnings when using the programming address mode (PAM) registers. The Assembler notifies the user of its assumed state of the PAM registers, and the set value the assembler uses for the encoding of some instruction.

This should help the user avoid cases of inconsistency between his assumptions of the PAM registers state, and of that the Assembler actually assumes for the encoding. Such gaps may happen as certain values may have already been declared/set in previously encoded sections.

To illustrate the problem refer to the following example:

Consider the below instruction in an assembly file:

mov b1h,(r1) || mov b1l,(r5) || r1+1,r5+1

It is using the default values of registers ARP0/1, and when it is being executed the correct post modification is being performed - 1. However, if at run time, the value in ARP0 register will change, so that the post modification of the pointers is 2 instead of 1 (value change from 0x0021 to 0x0084) then although the encoding hasn't changed, when this instruction is executed it will perform a post modification of 2.

Warning examples:

test.asm(3) : warning W512: Encoding of this instruction assumes use of PMCJ1, values set 0.

Use .LOAD/.RLOAD directive to declare/set the assembler with different values.

For more information about using programming address mode registers, refer to CEVA-Teak spec.

test.asm(3) : warning W509: Encoding of this instruction assumes use of ARP1, values set 1.

Use .LOAD/.RLOAD directive to declare/set the assembler with different values.

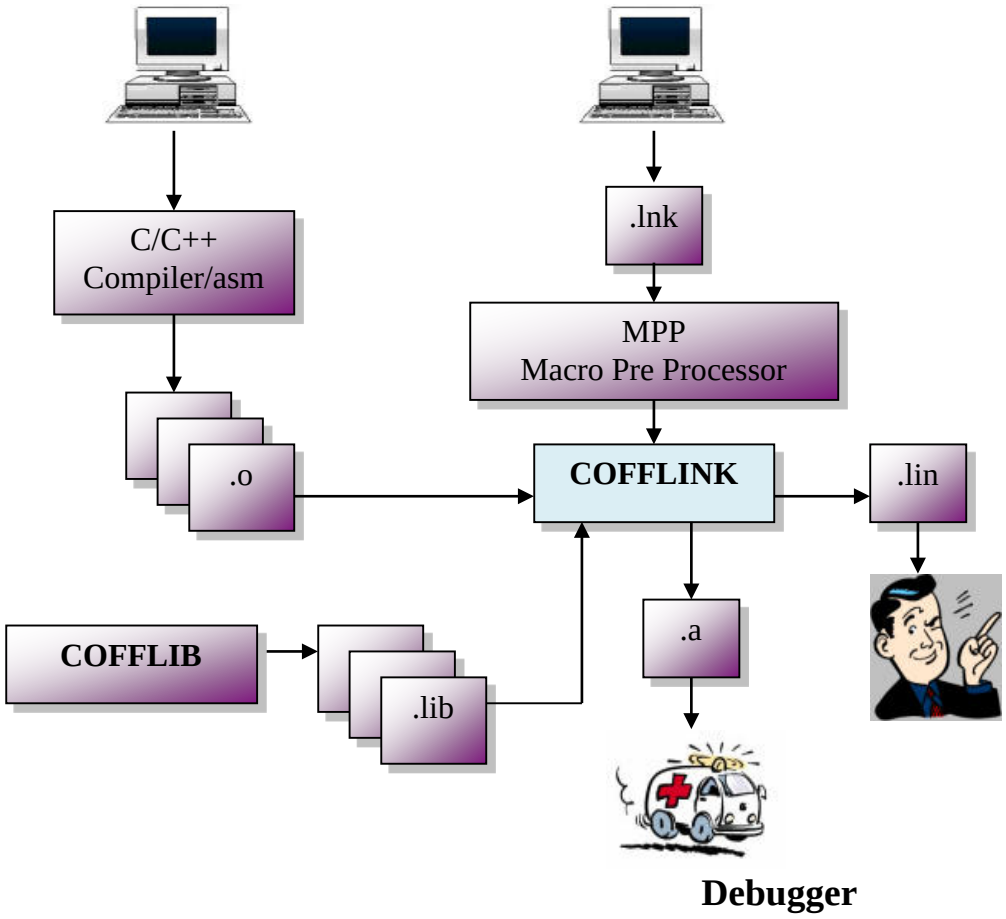
For more information about using programming address mode registers, refer to CEVA-Teak spec.

Work around:

In order to avoid these cases the user is encouraged (by the warning text) to use the dedicated assembly directives: .LOAD/.RLOAD, which specifies to the Assembler the PAM registers values to use when encoding. It is up to the user to decide when to use .LOAD (no cycle penalty) and when to use .RLOAD (one cycle penalty) but when this warning appears it is strongly suggested to use the directives and not ignore the warnings.

For more information regarding the programming address mode registers please refer to the CEVA-Teak specification in which they are described in details.

5. Linker Description



This chapter includes the following topics:

- **Linker Highlights**
- **Linker Invocation**
- **Linker's Command Line Options**
- **Linker Script File**
- **Linker Script Directives and Attributes**
- **Overlays Groups**
- **Linking Algorithm**
- **Generating Library Files**
- **Generating PROM Burnable Files**

5.1 Linker Highlights

The **CEVA-Toolbox** Linker is designed to link object modules created by the Assembler (version 8.0 or higher) or by the COFFLIB library Archive. The Linker supports the following features:

- Locating sections at absolute locations or relative to other sections (with an address **alignment** option), and to overlay sections. The Assembler does not support absolute location directives except for the **.ORG** directive. All linking/locating directives must be specified to the Linker.
- Overlays in both Code and Data memory spaces, for efficient on-chip data memory usage and program downloading applications.
- Code and data memory **segments** / pages for memory space requirements exceeding CEVA DSPs addressing space. When declaring a **segment <number>** in the Linker script file, all sections specified after, will be located within the declared memory segment <number>. Additionally, the Linker (V8.5 and newer) offers a comprehensive multi program paging mechanism. This mechanism provides the user with the ability to automatically create a multi-segment code at link time, without the need to recompile or modify source code or objects. This means, the user code is written in the same manner regardless of the final location of each function in the COFF executable file. The Linker, under the new command line option: **-multiProgPage**, locates the code as requested via the Linker script file, and adds a special code handling to each cross segment function. Note that the switch-to-segment process is totally transparent to the user, done by the Linker and is customizable. For more details see the Linker's Comprehensive Multi Program Paging Support chapter.
- Direct addressing support, **CEVA DSPs** processors include the **lpg #immediate** instruction, the Assembler supports link-time **@** and **PG** operators, and the Linker supports the **align** and **inpage** attributes; all can be used to support worry-free Direct Addressing.

- Libraries input.
- User-defined memory classes. Two address spaces/classes are predefined (code and data - default) and correspond to the processor's physical **code** and **data** memory spaces. Three more classes for CEVA-X and CEVA-XC: code_ext, data_ext and unified.
- An open system. Commands are entered via a script file will be referred to hereafter as the Linker script file.
- Unreferenced C/C++ functions information - For C/C++ files compiled with the **–g[0/1/2/3]** Compiler's command line option, it is possible to locate functions that are not called by any other function. This allows the user to remove such functions from the application and hence decrease its COFF file size. This information is printed to the **.lin** file under the new **-unrefFuncs** command line option.
  Note: Applicable from SDT V8.4 and higher.
- Additional information included in the COFF file - The COFF file contains new information to be used by the Debugger. The additional information consists of RTL version, **psync_fast** (applying to CEVA-Teak, CEVA-TeakLite and CEVA-TeakLite-II only), number of mapped code and data segments and fast floating point. This information enables the Debugger to automatically adjust its working mode with respect to the COFF file loaded, and hence frees the user from manually doing it by CLI commands. It may also be helpful for keeping track of various COFF files compiled for different needs. **Note:** Applicable from SDT V8.4 and higher.

CEVA-X and CEVA-XC core only:

- Post Linker Optimizer:
  An additional optimization pass is added to the Linker (SDT V9.1) for code size reduction (in future releases cycle count reduction will be added as well). The new optimization pass is performed at the last stages of the linking process. The new optimizer module takes a global view of the application and therefore is able to perform optimizations which the Compiler cannot due to its overall picture limited

to a single file. The new Post-Linker optimization pass is activated by default and can be disabled by applying the Linker's '-noOs' command line option. The code size reduction percentage that is achieved by this optimization can be one or even two digits, determined by the application nature and its' mapping. The Linker's '-Oinfo' command line info can be used for printing a report to the listing file.

Note: *The Linker is installed as part of the CEVA-Toolbox Software Development Tools (SDT) installation. Refer to the* **SDT Installation & Licensing Scheme Guide** *for further information.*

5.2 Linker Invocation

Activate **cofflnk**, directly from command line prompt according to the following syntax:

cofflnk [options] ScriptFileName.lnk

Where:

- **options** – List of command lines options specified in Linker **Command Line Invocation** topic.

- **ScriptFileName.lnk** – User script file including Linker directives as specified in **Linker Script File Directives** chapter.

Notes:
1. *cofflnk is applied to all CEVA DSPs Assembler object files output regardless of the core's type.*
2. *When multiple (same) command line options appears in the command line, only the last occurrence will take effect.*

5.3 Linker's Command Line Options

Linker executable, **cofflnk.exe**, can be invoked directly from the DOS command line, and **cofflnk** can be invoked in Linux using the following way:

cofflnk [options] SriptFileName.lnk

Table 2 below specifies the available command line options.

Table 2 Linker Command Line Options

Option	Description
-**allowUndefSections**	Allows to put in the Linker script file sections that are not defined in the project (*.o and *.lib files). When applying this option, the Linker will generate warnings only (and not errors). This option is useful for a script file that is shared between several derivative of the same project where certain sections sometimes are used in the project and sometimes are not.
-format length, width	Format the listing pages to include <length> lines and <width> characters in each line, similar to the .FORMAT directive (default is 60, 80).
-funcRef	Used for C/C++ files compiled with either **–g[0/1/2/3]** Compiler's invocation option to prints functions reference table into the listing file that details all the callers to each function.
-G	Dumps global symbols to the .lin file.
-h	Displays help information.
-l lin_file	Forces output file name for Linker listing (full name is required).
-L	Output is directed to the standard output.
-libRefInfo	The symbols that are referenced from libraries objects are printed into the listing file.
-m	Creates sections map (denotes the range of each mapped section and the total code and data memory size in ***.lin** file). In addition, the output includes all the unmapped code and data memory areas (holes) that are not used by the linked application (SDT V8.4 and higher).
-multiProgPage	Supports automatic program segment crossing. For more details, see the Linker's Comprehensive Multi Program Paging Support chapter.
-multiProgPageFuncPtr	Supports automatic program segments crossing under the assumption that if a function is called via a function pointer, then the calling and the called functions are in the same segment. For more details, see the Linker's Comprehensive Multi Program Paging Support chapter.
-noOs	Disable the Post Linker Optimizer pass.

Option	Description
-o a_file	Forces output file name for executable (full name is required).
-Oinfo	Generates Post Linker Optimizer information into the listing file.
-p[,mpp options]	Invokes the MPP on the .lnk file (with its options, if specified) before the Linker (each option should be written following a comma, without blanks). **Note**: MPP directives can be used in the .lnk file (only if –p is applied). When applying the –p option it is necessary to add the –n mpp option as well in order to suppress line information directives in which the Linker does not support in the script file level. Example: coffInk -p,-n,-i,d:\inc_files my_script.lnk ; MPP is invoked first with -i option. When applying the -p Linker's option, the Linker will search for the MPP location in the following order: The location of the Linker executable file. The locations specified in the PATH environment variable.
-quiet	Do not print the banner message.
-r	Similar to **-m**, creating sections map (and unmapped data and program memory areas) **in addition** to all relocation entries. **Notes**: Relocation Entries are placeholders set and reserved for symbols references by the Assembler to be filled out by the Linker and used by the Debugger.
-removeUnRefFunc	Removes all unreferenced function from the final COFF.
-s	Dumps symbol table to the .lin file.
-S	Use script file from the standard input. To exit Linker interactive mode, apply a single or double ^z (control+z) keystroke character sequence.
-unmentionedSections	Generates a list of sections that are not mapped in the .lnk file (and left for the Linker's default mapping).
-unrefFuncs	Used for C/C++ files compiled with either **-g[0/1/2/3]** Compiler's invocation option to indicate the functions that are not called by any other functions and hence, can be removed (by the user) in order to save code size.
-w	Activates line wrapping in listing file.
-x	Creates cross-reference table.
-relativePath	Stores relative paths for any file entry in the object files. This option supports relative project tree that enables copying the project tree to another location. This is the default option used by the Linker.

Option	Description
-absolutePath	Stores an absolute path for any file entry in the object files. This option is for prior tools version compatibility.
-msgFullPath	Adds files' full path in error and warnings messages.
-mapUnmentionedLo	Changes the default mapping from 'next' to 'lo' to all sections that do not appear (mapped) in the Linker's script file (unmentioned).
-mapUnmentionedSmallest	Changes the default mapping from 'next' to 'smallest' to all sections that do not appear (mapped) in the Linker's script file (unmentioned).
-mapDefaultLo	Changes the default from 'next' to 'lo' to all sections that are mapped in the Linker's script file without any mapping directive.
-mapDefaultSmallest	Changes the default from 'next' to 'smallest' to all sections that are mapped in the Linker's script file without any mapping directive.
-noLst	No listing file is generated. (*.lin)
-sortUnmentioned	Performs a sort by section size before mapping to all sections that do not appear (mapped) in the Linker's script file (unmentioned). That is, the mapping order will start from big to small.
-tl<mssVersion>	Sets CEVA-TLxxx core target of the output executable file: TL3210/TL3211/TL410/TL420
-internalCode<MemorySize>	Allows configuring the internal code memory size for CEVA-X1622/CEVA-X1641/CEVA-X1643/CEVA-XC323 N can be: 64, 96, 160, 288, 544 or 1056 for CEVA-XC321 N can be: 32, 64, 128 or 256 for TL321x N can be: 4, 8, 16, 32, 64, 128, 256 or 512 for TL4 N can be: 8, 16, 32, 64, 128, 256 or 512 The default value for CEVA-TL3210 is –internalCode64. The default value for CEVA-TL3214 is –internalCode64.
-internalData<MemorySize>	Allows configuring the internal data memory size for CEVA-X1622 N can be: 64, 128 or 256 for CEVA-X1641/CEVA-X1643/CEVA-XC323 N can be: 64, 128, 256, 512, or 1024 for CEVA-XC321 N can be: 64, 128, 256 or 512 for CEVA-TL321x N can be: 0, 6, 8, 12,16, 24, 32, 48, 64, 96, 128, 192, 256 or 512 for CEVA-TL41x N can be: 0, 6, 8, 12,16, 24, 32, 48, 64, 96, 128, 192, 256, 384 or 512 for CEVA-TL42x N can be: 0, 6, 8, 12,16, 24, 32, 48, 64, 96, 128, 192 or 256

Option	Description
-alignAllSections[,c:N][,d:M]	Aligns code sections to N and data sections to M. If one of the memory types is aligned using the switch, the second memory type will not have a default alignment value. If no alignment is explicitly defined for both memory types using the switch, the code sections will be aligned to 0x20, and the data sections will be aligned to 0x4. This switch affects only sections which are not explicitly aligned in the '.lnk' file
-labelLimit	Truncates all label usage to 16 bits in compatibility mode in order to optimize code size.
-detectRtlBug22	Detects CEVA-X RTL Bug #22 occurrences in the .a file Relevant for CEVA-X only
-fixRtlBug22	Fix CEVA-X RTL Bug #22 occurrences in the .a file Relevant for CEVA-X only
–ignoreRtlBug22	Ignore CEVA-X RTL Bug #22 occurrences in the .a file Relevant for CEVA-X only
-PIC	Disable code size optimization over code sections which are being referred from data sections, to allow them be position independent. Relevant only for CEVA-X, CEVA-XC and CEVA-TL3/4.
-separateSymTbl	Generates a separated text file for the symbol table the symbol table will be located under a file name is sortedSymbolTable.txt
-cevax1620	Set the .a file core type to CEVA-X1620
-cevax1622	Set the .a file core type to CEVA-X1622
-cevax1641	Set the .a file core type to CEVA-X1641
-cevax1643	Set the .a file core type to CEVA-X1643
-cevaxc161	Set the .a file core type to CEVA-XC161
-cevaxc321	Set the .a file core type to CEVA-XC321
-cevaxc323	Set the .a file core type to CEVA-XC323
-cevaxc4210	Set the .a file core type to CEVA-XC4210
-cevaxc4200	Set the .a file core type to CEVA-XC4200
-cevaxc641	Set the .a file core type to CEVA-XC641
-dataBigEndian	Encodes data sections in Big Endian format (relevant only for CEVA-XC323 core)
-removeNoloadSec	Removes raw data of all sections marked as 'noload' from the target COFF
-i <include directories>	Add objects and libraries include directories. Used for objects that were provided using the .lnk script file, the –obj command line option and by the –objEnvVar command line option.

Option	Description
-I <include directories>	Add objects and libraries include directories. Used for libraries that were provided using the .lnk script file, the –lib command line option and by the –libEnvVar command line option.
-obj <obj1,obj2...>	Comma-separated list of object file names that will be added to the linking process. When this switch is used there is no need to provide an .lnk file. Those objects will be ordered by the Linker after the objects given in the .lnk file (if provided).
-objEnvVar=<envVar>	The given environment variable lists comma-separated object file names that will be added to the linking process. When this switch is used there is no need to provide an .lnk file. Those objects will be ordered by the Linker after the objects given using the –obj command line option (if provided).
-lib <lib1,lib2...>	Comma-separated list of library file names that will be added to the linking process. Those libraries will be ordered by the Linker after the libraries given in the .lnk file (if provided).
-libEnvVar=<envVar>	The given environment variable lists comma-separated library file names that will be added to the linking process. Those libraries will be ordered by the Linker after the libraries given using the –lib command line option (if provided).
-enableExtMemTL41x	Activates the external memory in CEVA-TL410 & CEVA-TL411 where the external memory is disabled by default.
-errForWarn <warn1,warn2...>	Comma separated list of warning numbers which should be treated as errors
-secInfo	**Print detailed section information, such as address and size of variables and functions of each section**

Tip: *For best results (memory utilization wise), it is recommended not to mention in the Linker's script file any section that has no restriction on its location and to apply the following combination of command line options:* ***-sortUnmentioned -mapUnmentionedSmallest****..*

Notes:

1. *New default behavior for the location of the listing file. The generated listing file will be placed in the same directory the object file (*.o) was directed to using the -o switch. Using -l switch is still stronger then the default.*
2. *Unknown command line options. No fatal error is generated for typing errors of command line options. Warning is generated with the failure switch name.*

Examples:

1. **cofflnk** -m -s myproj.lnk

In this example, the Linker is invoked with the **myproj.lnk** input script file, generating

myproj.a executable file. In addition, memory mapping (**-m**) and symbol information (**-s**) are generated in **myproj.lin**.

2. **cofflnk** -p,-n,-d,Var=0 myproj.lnk
In this example, the Linker is invoked with the **myproj.lnk** input script file, generating **myproj.a** executable file. In addition, the **Var** variable is defined and assigned to **0** (this, for example enables the usage of conditions (**.IF** , **.ELSE** etc.) for **Var** in myproj.lnk file.

5.4 Linker Script File

The Linker script file has several script file blocks: **classes**, **objects**, **libraries**, **code**, and **data**. Multiple instances of each script file block are allowed. Among all, only the **objects** script section is mandatory. Script blocks order is meaningless, but the order of the contents within each script section (after combining multiple instances) is very important - it dictates the order of the link algorithm. Code and data assembly file sections (**.DATA/.CODE/.DSECT/.CSECT**) not explicitly mentioned in the script file, are linked according to a default algorithm. Refer to the **Linking Algorithm** section for details.

The optional **classes** script block is used to define a list of memory classes, each associated with a memory range in either the program (**code**) space or the **data** space. The default classes are **code** and **data**, each having a memory size of the full specific CEVA DSPs range. The **objects** script section contains an ordered list of all object files to be linked.

The **libraries** script section contains an ordered list of all library objects to be linked.

Each script block is declared using a script block label composed of the name of script block followed by a colon (:) that must be on a separate line by itself. Following is a structure of a Linker script file:

objects:

List of object files

libraries:

List of Library files

classes:

List of memory class (data and/or code) declarations

class1: ; defined in **classes** block

List code or data sections with optional directives and attributes

class2: ; defined in **classes** block

List code or data sections with optional directives and attributes

...

classN: ; defined in **classes** block

List code or data sections with optional directives and attributes

In addition, Linker script file syntax supports multiple address spaces for code and data (paging) using the **segment** keyword. The user can specify, prior to each assembly file section (or group of sections), the segment in which the following assembly file section will reside. The segment number for mapping is reset to 0 when a new script block starts or new segment keyword is used. If no **segment** number is specified, the default **segment** 0 is used.

e.g.

code:

```
sec1          ; is mapped by default into segment 0
segment 2     ; instructs the Linker to map the following sections into segment 2
sec2
```

Only one file or assembly file section per line is permitted. Lines may contain a trailing comment noted by a semi-colon (;). Blank lines are permitted. The script file is case sensitive.

Following are two typical examples of Linker script files. The first is for linking together 3 object files, declaring two user-defined memory classes (in addition to the two predefined classes) and for locating particular assembly file sections at specified addresses in specified sections.

Example 1:

```
objects:
        file1.o
        c:\mypath\subdirectory\file2.o
        ..\..\file3.o
classes:
        xram [d:0000,d:03ff]
        yram [d:fc00,d:ffff]
code:
        Section1 ; will be located at 0x0000
        Section2 at 0x8 ; will be located at 0x0008
    segment 1 ; the next section will be in address space 1
        Section3

data:
    segment 2                   ; the next section will be in address space 2
        Section4 at 0x8000 inpage
        Section5 align 0x100 inpage ; will be after section4

xram:
        Section6                ; will be located at 0x0000
        Section7 align 0x100 inpage ; section7 starts in any address multiple of
                                    ; 0x100 words following Section6 and warns
                                    ; if it exceeds a size of 0x100 words.

 yram:
        Section8                ; will be located at 0xfc00
```

The second example uses libraries and creates a data overlay in the same data memory space.

Example 2:

objects:

file1.o

c:\mypath\file2.o

..\file3.o

libraries:

file4.lib

..\lib\file5.lib

code:

Section1 ; will be located at 0x0000

Section2 **at** 0x100 ; will be located at 0x0100

Section3 **lo** ; Linker will start trying to locate from address 0

data:

Section4

Section5 **inpage** ; will be after Section4

{ ; start defining an overlay group

Section6 ; located after Section5

Section7 **align** 0x100 **inpage**

Section8 **at** Section6 **noload**

} ; end an overlay group

Note: *When using C source files in the project, some specific rules should be applied when writing the Linker script file. Please refer for details to Linking Guidelines application note and comments inside default Linker script file*

5.5 Linker Script File Syntax

The syntax of the Linker's script file consists of reserved keywords, mapping directives and mapping attributes. The complete list of reserved keywords, directives and attributes in alphabetical order is as follows:

Keywords:

- **classes:**
- **code:**
- **data:**
- **code_ext:**
- **data_ext:**
- **unified:**
- **libraries:**
- **objects:**
- **segment**
- **internal**
- **external**
- **share [segments] [SegmentList]**

Directives:

- **at**
- **hi**
- **lo**
- **next [list]**

Attributes:

- **align**
- **inpage**
- **noload**
- **size**
- **smallest**
- **clone** SectionName
- **symbol**

The optional location directives and attributes **at**, **align**, **inpage**, **lo, hi, smallest** and **next** enable the user to specify the Linker the absolute memory location of the various data and code assembly file sections (included in the object files mentioned in the objects script section). Each memory class defined in the classes script block (including the default **code** and **data** classes) is described separately in a different script section.

To locate an assembly file section (of types **- .CODE**, .**CSECT**, **.DATA** or **.DSECT**), one and only one of the following location directives must be explicitly or implicitly associated with each assembly file section:

at ConstantNumericExpression
at SectionName
at hi
at lo
at next
at next (AddressExpressionList)
lo
next
next (AddressExpressionList)
hi

An **AddressExpressionList** is a list of symbolic addresses, for example:

(SegA, SegB, SegC)
(SegA+10, SegB-5)
(100)
(next, lo+100)
(hi-0x100)

All the above location directives can be combined with a Numeric Offset, **smallest**, **align**, **inpage**, **noload, or clone** SectionName attributes, for example:

```
at (SegA + 7 align 0x100)
next (SegB+6,SegC) - 10
at hi - 0x200 inpage
lo smallest
at 0x500 noload
lo clone SectionName
```

Important:

Notice that the Linker surrounds each sub-expression after each + or - operator, by parentheses. This means that:

```
at next (segA - 0xff +4)
at (SegB -100 - 0x15 + 3)
```

is actually interpreted as

```
at next (segA -(0xff +4))
at (SegB - (100 - (0x15 + 3)))
```

In order to map sections to multiple memory address spaces (segments / pages), the **segment** keyword should be applied prior to the mapping of the assembly file section.

Note:
When linking CEVA-X or CEVA-XC object files the ‘segment’ keyword is meaningless.

5.5.1. Memory Classes – Classes Script Block

A memory class defines a list of logical memory types of a target executable file. Classes are used to guarantee that particular data structures or programs fit into physical memory devices or memory limits imposed by a particular hardware configuration.

Script class includes the following features:

1. Up to 14 classes may be defined in a script file

2. Defining script **classes** blocks is entirely optional

By default, two memory classes are predefined – the code and data memory classes, each having the default range of the entire program (code) and data memory spaces respectively (0x0000 – Core specific code/data maximum range size).

For CEVA-X and CEVA-XC object files linking only:
Three more default memory classes are predefined – 'code_ext', 'data_ext' and 'unified'. All these classes represent the external memory of the core, where the other two classes mentioned above (code, data) represent the internal core memory.
Default configuration:

Internal memory -

code = 64KB (addresses: 0 - 0xFFFF)

data = 64KB (addresses: 0 - 0xFFFF)

External memory -

code_ext = 4GB (addresses: 0 - 0xFFFFFFFF)

data_ext = 4GB (addresses: 0 - 0xFFFFFFFF)

unified = 0

3. Each class is assigned by memory range (defining the **lo** and **hi** addresses of the class) in either the program (code) space or the data space.
   Classes can also be defined in unified area.
   Classes can be defined in internal / external code / data.

4. Each memory class defined in the Linker script file should include a script section mapping block specifying the assembly file sections (**.CODE**, **.CSECT**, **.DATA** and **.DSECT**) declared in the object files and belong to that memory class. During the linking process the Linker verifies that the appropriate sections fit into these ranges.
5. Memory classes may overlap in their address ranges; however, an assembly file section cannot implicitly overlap in two different classes. This means that once a particular assembly file section**,** in a particular class, is located in either program (code) or data memory space, another section, in the same (or in another class) cannot be mapped into the same memory addresses (within the same memory space) occupied by the first section. This is so unless an **overlay** group is explicitly declared, as described below. However, when using a **segment** declaration to switch between memory spaces, overlapped addresses between two sections that belong to different segments is possible.
6. Additions for the Linker's comprehensive program paging support.
   Extended definition of classes in the .lnk file: It is now possible in the classes' definition to define sharing between program memory pages. When doing so, the Linker will take this extra information into account when taking decisions on whether a function should be called as far (through the page switch function) or as short (by a regular 'call' instruction).

The Linker behavior in such a case is as follow:

a) Any call to shared memory will be done without going through the page switch routine, unless the calling function is located in a segment that is not defined as shared in the class definition where the called function is located (it can happened only in case that a specific segment list is asked to be shared for a class).

b) Any call from shared memory to an address that is not in the same shared class will be done through the page switch routine (this is because the Linker does not have function call stack information and therefore does not know the segment of the function that initiated the call to the shared class).

Syntax:

ClassName [C|D|c|d|U|u:StartAddress, C|D|c|d|U|u:EndAddress] [internal|external] [share [segments] [SegmentList]]

Note that memory type character (C|c or D|d or U|u) should match in both start and end addresses.

Or:

ClassName [C|D|c|d|U|u:StartAddress, memoryRangeLength] [internal|external] [share [segments] [SegmentList]]

Example:

classes:

xram [d:0000,d:03ff] ; user-defined class for on-chip xram

yram [d:fc00,d:ffff] ; user-defined class for on-chip yram

eprom [c:8000,c:bfff] ; user-defined class for external eprom

fast [c:4000, 0x2000] ; user-defined class for fast memory

For CEVA-X and CEVA-XC object files linking only:

myCodeInt [C:1000 ,0x200] internal ;define a class in internal code memory

myDataExt [D:20000,0x1000] external ;define a class in external data memory

myUnified [U:50000,0x5000] external ;define a class in external unified memory

Note that by default, code and data classes are always predefined:

code [c:0000, c:ffff] ; defined by default – For **CEVA-TeakLite,** CEVA-TeakLite-II, CEVA-X and CEVA-XC **only**

code [c:00000, c:3ffff] ; defined by default – For the **CEVA-Teak** only

data [d:0000, d:ffff] ; defined by default – For all CEVA DSP**s**

Additions for the Linker's comprehensive program paging support

class1 [C:0000, C:1000] share segments ; Share all program segments in the given ; range.

class2 [C:3000, C:4000] share segments 1,2 ; Share only program segments 1 and ; 2 in the given range.

class3 [C:5000, C:6000] share segments; Share all program segments in the given ; range.

Notes:

1. ***code*** *upper limit Is the Core specific* ***code*** *maximum range size. Once classes are defined, one should specify the* ***sections*** *that belonged to each class. In the example above, one can add 8 script sections mapping blocks, named* ***code***, ***data***, ***xram***, ***yram***, ***eprom, myCodeInt, myDataExt and myUnified****. Using Linker directives and attributes, the programmer can instruct the Linker to locate some or all the sections into specific memory locations. Refer to the* ***Linker Script File Directives*** *and more in* ***Script Code*** *and* ***Script Data*** *topics for details*
2. *Sections defined in the object files, that are not explicitly mentioned in any of the class script sections, are mapped as described by the default linking algorithm:* ***.CODE*** *and* ***.CSECT*** *sections are mapped into the default* ***code*** *script section (sequentially after the last section already mapped).* ***.DATA*** *and .****DSECT*** *sections are mapped into the default* ***data*** *script section, (sequentially after the last section already mapped).*

5.5.2. Code Memory Class – Code Script Block

A memory **code** (reserved keyword) class specifies the assembly file sections (**.CODE**, **.CSECT**, **.DATA** and **.DSECT**) defined in the object files. Code sections are linked in order of appearance (and subject to Linker directives) into the CEVA DSP**s** default program (code) memory **class**. The assembly file sections in the code class are not required.CODE and .CSECT the only required sections. ROM tables, for example, are .DATA or .DSECT sections that can also be linked into the program (code) space. The syntax for specifying a section (within class definition) is as follows:

SectionName [**at** [**hi** | **lo** | **next** [(list)]] SymbolicNumExpr]
[**align** NumExpr] [**noload**]

or

SectionName [**lo** | **hi** | **next | smallest**] [**align** NumExpr] [**noload**]

All directives and attributes are optional. All directives can be appended with a +/- constant numeric expression offset (for example: at SecA + 0x0f1) associated with the **align** or **noload** attributes.

- **at** directive specifies an exact address in which to map a section.

- **align** attribute specifies that the section must be mapped on the next address, which is a multiple of the specified numeric expression.

- **lo** directive is used to instruct the Linker to map starting from the memory class' lowest address (default 0 for code class), even if other assembly file sections have already been mapped at higher addresses, obviously, without causing overlapping of assembly file sections.

- **hi** directive is used to instruct the Linker to map and fill the memory space relative to the memory class' highest address, (i.e. address 0xffff in the code class).
- **next** directive specifies that the assembly file section must be mapped immediately after the previous assembly file section.

- **Smallest** attribute instructs the Linker to search for the **smallest** hole (in the memory class that the section is to be located in) that can still fit the section. The **smallest** attribute purpose is to ease the linking process by providing a mechanism that better utilize the memory space that is essential in very big applications.

- **SymbolicNumExpression** is a C-style numeric expression that may contain one or more assembly file section names that have already been located (no forward references are allowed).

Example:

code:

sec1

sec2 **align** 0x100

sec3 **lo**

sec4 **at** sec2 + 0x600 **noload**

romtbl **next**

Note: *Refer to **Overlays Groups – General** and **Code Overlays Groups** for script code overlays.*

5.5.3. Data Memory Class – Data Script Block

A memory **data** (reserved keyword) class contains data assembly file sections to be linked in the order of appearance (and subject to Linker directives and attributes) into the default CEVA DSP**s** data memory space. The syntax for specifying an assembly file section (except within overlays) is as follows:

SectionName [**at** [**hi** | **lo** | **next** [(list)]]
SymbolicNumExpr] [**align** NumExpr] [**inpage**] [**noload**]

or

SectionName [**lo** | **hi** | **next | smallest**]
[**align** SymbolNumericExpr] [**inpage**] [**noload**]

All directives and attributes are optional. All directives can be appended with a +/- constant numeric expression offset (e.g. at SecA + 0x0f1) associated with the **align**, **noload** or **inpage** attributes.

- **at** directive specifies an exact address in which to map a section.
- **align** attribute specifies that the section must be mapped on the next address, which is a multiple of the specified numeric expression.
- **inpage** attribute is used for data assembly file sections that must not cross the physical page boundaries in the data space.
- **lo** directive is used to instruct the Linker to map starting from the memory class lowest address (default 0 for code class), even if other assembly file sections have already been mapped at higher addresses, obviously, without causing overlapping of assembly file sections.
- **hi** directive is used to instruct the Linker to map and fill the memory space relative to the memory class highest address (refer to relevant CEVA DSP**s** specification).
- **next** directive specifies that the assembly file section must be mapped immediately after the previous assembly file section.
- **Smallest** attrib**u**te instructs the Linker to look for the **smallest** hole (in the memory class that the section is to be located in) that can still fit the section. Applying the

smallest attribute eases the linking process by providing a mechanism that better utilizes the memory space, which is essential in very big applications.

- **noload** attribute causes the loader of the Debugger not to load that assembly file section at load time.
- **SymbolicNumExpression** is a C-style numeric expression that may contain one or more assembly file section names that have already been located (no forward references are allowed).

Example:

```
data:

    sec1

    sec2 at 0x240              ; locate Sec2 at 0x240

    sec3 align 0x100 inpage    ; locate Sec3 at page size multiples depending also
                               ; on Sec2 size and warn if crossing page boundaries

    sec4 inpage noload         ; warn when crossing page boundaries, do not load
                               ; the section on load time
```

Quite often data memory space on the DSP chip is limited, so the programmer is forced to use the same address space for different data assembly file sections. This can be accomplished using data overlays. Refer to **Overlays Groups - General** and **Data Overlays Groups** for details.

5.5.4. Code_ext Memory Class – code_ext Script Block

For CEVA-X and CEVA-XC object files linking only:

A **code_ext** (reserved keyword) memory that is used to map the assembly file sections defined in the object files into the external program memory. Unless explicitly mapped, code sections are linked in order of appearance (and subject to Linker directives) by default into the CEVA-X code memory (internal). Once code memory is full, the remaining code sections are linked into the CEVA-X default external program memory **code_ext** (reserved keyword) memory by default. The syntax for specifying a section (within class definition) is as follows:

SectionName [**at** [**hi** | **lo** | **next** [(list)]] SymbolicNumExpr]
[**align** NumExpr] [**noload**]

or

SectionName [**lo** | **hi** | **next** | **smallest**] [**align** NumExpr] [**noload**]

All directives and attributes are optional. All directives can be appended with a +/- constant numeric expression offset (for example: at SecA + 0x0f1) associated with the **align** or **noload** attributes.

- **at** directive specifies an exact address to which a section is to be mapped.
- **align** attribute specifies that the section must be mapped on the next address, which is a multiple of the specified numeric expression.
- **lo** directive is used to instruct the Linker to map starting from the memory class' lowest address (default 0 for code_ext class), even if other assembly file sections have already been mapped at higher addresses, obviously without causing overlapping of assembly file sections.
- **hi** directive is used to instruct the Linker to map and fill the memory space relative to the memory class' highest address, (i.e. address 0xffff in the code class).
- **next** directive specifies that the assembly file section must be mapped immediately after (next to) the previous assembly file section.

- **Smallest** attribute instructs the Linker to search for the **smallest** hole (in the memory class that the section is to be located in) that can still fit the section. The **smallest** attribute purpose is to ease the linking process by providing a mechanism that better utilize the memory space that is essential in very big applications.
- **SymbolicNumExpression** is a C-style numeric expression that may contain one or more assembly file section names that have already been located (no forward references are allowed).

Example:

```
code_ext:
  sec1
  sec2 align 0x100
  sec3 lo
  sec4 at sec2 + 0x600 noload
  romtbl next
```

5.5.5. Data_ext Memory Class – data_ext Script Block

For CEVA-X object files linking only:

A **data_ext** (reserved keyword) memory is used to map the assembly file sections defined in the object files into the external data memory. Unless explicitly mapped, data sections are linked in order of appearance (and subject to Linker directives) into the CEVA-X default (internal) data class. Once internal data memory is full, the remaining data sections are linked to the CEVA-X external data memory **data_ext** (reserved keyword) type by default. The syntax for specifying a section (within class definition) is as follows:

SectionName [**at** [**hi** | **lo** | **next** [(list)]]
SymbolicNumExpr] [**align** NumExpr] [**inpage**] [**noload**]

or

SectionName [**lo** | **hi** | **next | smallest**]
[**align** SymbolNumericExpr] [**inpage**] [**noload**]

All directives and attributes are optional. All directives can be appended with a +/- constant numeric expression offset (e.g. at SecA + 0x0f1) associated with the **align**, **noload** or **inpage** attributes.

- **at** directive specifies an exact address to which a section is to be mapped
- **align** attribute that specifies that the section must be mapped to the next address, which is a multiple of the specified numeric expression.
- **inpage** attribute is used for data assembly file sections that must not cross the physical page boundaries in the data space.
- **lo** directive is used to instruct the Linker to map starting from the memory class lowest address (default 0 for code class), even if other assembly file sections have already been mapped at higher addresses, obviously without causing overlapping of assembly file sections.
- **hi** directive is used to instruct the Linker to map and fill the memory space relative to the memory class highest address (refer to relevant CEVA DSP**s** Architecture specification).

- **next** directive specifies that the assembly file section must be mapped immediately after (next to) the previous assembly file section.
- **Smallest** attribute instructs the Linker to search for the **smallest** hole (in the memory class that the section is to be located in) that can still fit the section. Applying the **smallest** attribute eases the linking process by providing a mechanism that better utilizes the memory space.
- **noload** attribute causes the loader of the Debugger not to load that assembly file section at load time.
- **SymbolicNumExpression** is a C-style numeric expression that may contain one or more assembly file section names that have already been located (no forward references are allowed).

Example:

```
data_ext:

  sec1
  sec2 at 0x240              ; locate Sec2 at 0x24
  sec3 align 0x100 inpage    ; locate Sec3 at page size multiples depending also
                             ; on Sec2 size and warn if crossing page boundaries
  sec4 inpage noload         ; warn when crossing page boundaries, do not load
                             ; the section on load time
```

5.5.6. Unified Memory Class – Unified Script Block

CEVA-X object files linking only:

A **unified** (reserved keyword) memory is defined as an external memory, containing code and data assembly file sections to be linked in order of appearance (and subject to Linker directives and attributes) into the CEVA-X unified memory space. The syntax for specifying an assembly file section is as follows:

SectionName [**at** [**hi** | **lo** | **next** [(list)]] SymbolicNumExpr]
[**align** NumExpr] [**noload**]

or

SectionName [**lo** | **hi** | **next | smallest**] [**align** NumExpr] [**noload**]

All directives and attributes are optional. All directives can be appended with a +/- constant numeric expression offset (for example: at SecA + 0x0f1) associated with the **align** or **noload** attributes.

- **at** directive specifies an exact address to which a section is to be mapped.
- **align** attribute specifies that the section must be mapped to the next address, which is a multiple of the specified numeric expression.
- **lo** directive is used to instruct the Linker to map starting from the memory class' lowest address (default 0 for code_ext class), even if other assembly file sections have already been mapped at higher addresses, obviously without causing overlapping of assembly file sections.
- **hi** directive is used to instruct the Linker to map and fill the memory space relative to the memory class' highest address, (i.e. address 0xffff in the code class).
- **next** directive specifies that the assembly file section must be mapped immediately after (next to) the previous assembly file section.
- **Smallest** attribute instructs the Linker to search for the **smallest** hole (in the memory class that the section is to be located in) that can still fit the section. The **smallest**

attribute purpose is to ease the linking process by providing a mechanism that better utilizes the memory space that is essential in large applications.

- **SymbolicNumExpression** is a C-style numeric expression that may contain one or more assembly file section names that have already been located (no forward references are allowed).

Example:

```
unified:
  sec1
  sec2 align 0x100
  sec3 lo
  sec4 at sec2 + 0x600 noload
```

5.5.7. Objects List – Objects Script Block

Script objects list contains an ordered list of all object files to be linked.

- Default objects extension of .o is assumed when extension is missing
- Absolute and relative paths can be defined
- Environment variable can be used in order to define a path

Note: *Unlike linking libraries, the Linker inserts all the list objects to the resultant executable, even if it contains dead code or data (unreferenced and hence unnecessary). For C/C++ applications, see the* ***-unrefFunc*** *command line option (Table 2)*

Example:

objects:

```
mod1.o
\path\mod2.o
mod3.xyz
mod4                    ; mod4.o (default extension is appended)
..\obj\mod5.o
%WORKDIR%\mod6.0        ; WORKDIR is environment variable
```

5.5.8. Libraries List – Libraries Script Block

The **libraries** (reserved keyword) script section contains an ordered list of all library object files to be linked.

- Default libraries extension of .lib is assumed when extension is missing
- Absolute and relative paths can be defined
- Environment variable can be used in order to define a path

Note: *Libraries are not linked unless required, that is, 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. Since system libraries (of the Compiler) include one function per object file, it ensures that the minimal necessary code is inserted. For this reason, it is advised to create user libraries in the same way.*

Example:

```
libraries:

lib1

\path\lib2.lib

lib3.xyz

%WORKLIB%\mylib.lib      ; WORKLIB is an environment variable
```

5.5.9. segment

5.5.10. segment <number>

For all CEVA-DSPs except CEVA-X and CEVA-TeakLite-III:

The use of segments (reserved keyword) expands CEVA DSP**s** memory space addressing capabilities. The **segment** directive must be followed by a **segment** number, which specifies the address space number in which the assembly file section will be placed. All the assembly file sections following **segment** declaration will be placed within the specified **segment**, unless a new segment declaration has been issued or a new memory class is started. The Linker maintains different segment number for Program and Data memories that are updated according the **segment** keywords and the type of memory class they are defined in. If no **segment** declaration is specified, then default **segment** 0 is used. Implicitly mapped section (that are not mapped explicitly by the user in the .lnk file, but appears in the object files and left for the Linker's default mapping algorithm) are mapped to the current Program / Data segment as maintained by the Linker. If the current segment is different than 0, the Linker issues a warning that can be disabled by the **–cancelSegWarn** command line option.

Since each **segment** represents a unique address space, each **segment** has its own location counter. Thus, switching to a new **segment** will cause sections to be placed starting from address 0, or from the first address free in that **segment**.
For more information on paging / **segment** refer to the *Segments (Paging) SDT Support* application note.
For more information on the Linker's program paging refer to the Output Files section 4.7.

Example:

code:

```
segment 1       ; start segment 1
        SecA    ; SecA will be placed in segment 1 at 0
```

```
        SecB    ; SecB will be placed in segment 1 after SecA
segment 0
        SecC    ; place SecC in segment 0 at address 0
segment 1       ; return to segment 1
        SecD    ; SecD will be placed in segment 1 after secB
```

5.5.11. at

The **at** directive should be followed by an address expression. An address expression can be a numeric value, a simple numeric expression, the Linker directive **lo** or **hi**, a reference to a previously located section (no forward references are allowed) or a reference to a next expression. Applying the **at** directive in the Linker script file, means the Linker must locate the associated section **at** the indicated address expression, subject to an optional alignment. Error will be generated if the address is already occupied.

Examples:

```
Sec A at 0x500          ; locate SecA at address 0x500
Sec B at SecA+0x100     ; locate SecB at address 0x600
Sec C at lo             ; locate SecC at lowest address of the class
```

5.5.12. lo

The **lo** directive stands for the lowest address of the memory class. For the predefined code and data classes, lo=0x0000. The use of **lo** directive in the Linker script file instructs the Linker to start searching from the lowest address of the memory class, subject to an optional alignment constraint. By default, if no location directive is assigned to the first section in the class, it will be located at the class first address i.e. at **lo**.

Example:

code:

```
Sec0 at 0x100           ; locate Sec0 at 0x100
Sec1 at lo              ; locate Sec1 at 0x0000
Sec2 at 0x1000          ; locate Sec2 at address 0x1000
Sec3 lo                 ; try to locate Sec3 before Sec0 or Sec2 if possible
```

5.5.13. hi

The **hi** directive stands for the highest address available in the memory class. For the predefined Data classes, hi=0xffff. For the Code class, it is core's dependent – CEVA-TeakLite / CEVA-TeakLite-II = 0xffff, CEVA-Teak = 0x3ffff. The Linker will start searching for a hole big enough to contain the specified section. It will start searching for such a hole from the highest address of the memory class downwards.

Note the **hi** directive, when used in conjunction with the **at** directive, forces the Linker to locate the specified section exactly **at** the highest address of the memory class.

Examples:

```
; Sec1 size is 0x100 words
; Sec2 size is 0x300 words.

classes:
        yram [d:fc00,d:ffff]
yram:
        Sec1 hi ; locate Sec1 end address at 0xffff)
        Sec2 at hi - 0x200 ; locate Sec2 at 0xfdff
        Sec3 hi ; Depending on Sec3 size it is located either before or after Sec2
```

5.5.14. next

next (List)

The **next** directive is used to indicate the next available free hole's address that fits the section. It can be followed by an address expression list in which case it means the next available address after all specified addresses. Applying the **next** directive (or when no directives are given at all) instructs the Linker to start searching from the address immediately following the high address of the previous section that was located, subject to an optional alignment constraint. Using **next (List)** has a similar effect to the simple **next** case described above, except that the search begins with the address following the maximal high address of all sections in the list. When the list contains a constant numeric expression, the high address is one less than the value of the expression (so that the search can start at the value).

Example:

data:

```
Sec1 at 0x1000              ; locate Sec1 at address 0x1000
{
  Sec2                      ; locate Sec2 after Sec1
  Sec3 at Sec2              ; locate Sec3 at the beginning of Sec2
  Sec4 at next (Sec2, Sec3) ; locate Sec4 after the longer section among
                            ; Sec2 and Sec3
 Sec5 next                  ; locate Sec5 after end of Sec4
}
```

5.5.15. align

The **align** attribute should followed by a numeric constant value. It is used to force the Linker to place a section at an address that is an integer multiple of the numeric constant. It can be appended to any other location expression.

Examples:

```
SecA

SecB align 0x100 ; locate SecB after SecA on address that is multiple of 0x100

SecC at SecB+50 align 4 ; locate SecC at SecB+50, but align to a multiple of 4
```

5.5.16. inpage

The **inpage** attribute is used with data sections requesting the Linker to check that the associated section does not cross a physical page boundary (256 data words long), and issue a warning if it does.

Examples:

data:

```
Sec1 inpage    ; locate Sec1 at 0x0000, make sure it does not cross page boundary

Sec2 at 0x180 ; locate Sec2 at 0x180

Sec3 inpage    ; locate Sec3 after Sec2, make sure it does not cross page boundary
```

5.5.17. noload

The optional **noload** attribute is used for data sections linked with other sections, but intended not to be loaded by the loader of the Debugger at load time. This is useful for simulating uninitialized data memory. When enabled, the Debugger reports read operations from uninitialized memory as an exception error. Refer to the Enable Exception memory debugger command in the *CEVA-Toolbox Debugger Reference Guide* for details.

The **noload** option can be used also in **Overlay** of data or code where only one section image of the overlay group is loaded at load time, while the other sections' images (from the overlay group) will be loaded at run-time by the application as required.

5.5.18. size

The Linker **size** attribute purpose is to extend a predefined section or to create a new section with a specified size. The **size** attribute can be implemented on sections defined in CRT0 file (such as: MALLOC_SECT_, STACK_SECT_, …) according to the specific application's requirements instead of changing and recompiling CRT0.C. In addition, it can be used for defining reserved memory areas in the **.lnk** file level (such as: in emulation – Mailbox, OCEM, BIU, Monitor, MMIO, …) so it will not be overwritten by the application's code / data.

Notes:

1. *The additional data words will be set to zero.*
2. *When used to create a new section, this section will have a **noload** attribute.*
3. *The requested size should be greater or equal to the (existing) section's size in order to have effect. Otherwise, the original size will be left untouched and the Linker will issue a warning.*

Examples

```
SecA                              ; SecA section will be allocated next
                                  ; to the last section defined

Reserved_Area at 0x1000 size 0x400 ; Reserved_Area section will be
                                  ; located at 0x1000 and will be
                                  ; 0x400 words size

SecC                              ; SecC section will be allocated next
                                  ; to Reserved_Area section

SecD size 0x200                   ; SecD will be located next to SecC
                                  ; with size of 0x200 words
```

5.5.19. smallest

The **smallest** mapping attribute instructs the Linker to search for the smallest hole (in the memory class that the section can be located in) that can still fit the section. Applying this option eases the linking process by providing this mechanism that better utilize the memory space, which is essential in very big applications. When combining the smallest attribute with the **lo** or **hi** directives, the Linker will locate the section at the lowest or highest addresses (respectively) of the smallest fitting hole.

Using 'smallest' by other directives:

- Lo:
- Find the smallest hole and locate at the hole start address.
- Hi:
- Find the smallest hole and locate at the hole high address minus the section size.
- Next:
- Same as with lo (also same as smallest alone)
- Align N:
- Find the smallest hole and locate the section in the first address aligned to N.

Example:

...

data:

```
my_sec smallest hi     ; search for the smallest memory hole and locate the section
                       ; at its highest address.
sec2 smallest
sec3 smallest align 0x20 ; Find smallest hole and locate sec3 in the first
                         ; address aligned to 0x20 in the hole.
```

Note:

When working with Meta-Sections and using the smallest attribute, it is not guaranteed that both sections (meta and the section) will be located into the same hole.

5.5.20. clone SectionName

The **clone** attribute provides the ability to locate a section multiple times in the same memory type (e.g. data, code_ext and other memory classes). Cloning of a section is supported with the following Linker's script file syntax:

CLONE_SEC_NAME [mapping Directive and Attributes] clone SOURCE_SEC_NAME

Motivation:

Section cloning can be used in situations in which a section should be mapped in both RAM and ROM, in the same memory type (class), e.g. ‘code’, ‘data_ext’, etc. For example: Assume the application resets all the initialized variables to initial value set by application memory load. This initialization is implemented in the Complier's start up routine (CRT0.C), which copies the ‘const_data’ and ‘.data’ sections images, set to be located in ROM (startup values), to another image that is located in RAM (work copy). The method to enable such sections duplication is via the clone mechanism.

E.g:

Initialization code -

```
.EXTERN const_data_clone
mov #const_data_clone,r0   ; r0 points to the source - the instance of
                           ; const_data clone section.
mov #const_data,r1         ; r1 points to the destination - the original
                           ; const_data section.
bkrep #N                ; N is the size of const_data section
{
 ld {dw} (r0)+,a0
 st {dw} a0,(r1)+
}
```

Linker's script file excerpt:

```
...
data:                                          ; Locate in data memory class
        Const_data lo                          ; Map the original section
        const_data_clone  next clone const_data ; Map the clone section
                                               ; using the same name as in
                                               ; the Assembly file
                                               ; (const_data_clone)
```

5.5.21. symbol

The **symbol** attribute allows defining a new label during link process.
This feature allows using undefined label names in C/C++ and assembly level while defining the actual label with its value in link time.

The **symbol** attribute does not define new memory: it only defines a new standalone global label which belongs to the class it is defined in

E.g:

data:

```
…
; label1 global symbol can be referenced from any source file, any reference
; to this label will be linked to address 0x20100
label1 at 0x20100 symbol

; label2 global symbol can be referenced from any source file, any reference
; to this label will be linked to address 0x30300
label2 at 0x30300 symbol
…
```

5.5.22. func

The **func** (or **FUNC**) attribute allows extracting a function from a section which contains it and using it as a standalone section.
This feature allows using function names in the Linker script file and map them in the regular mapping scheme.

E.g:

code:

```
…
foo at 0x10000 FUNC
…
```

5.6 Overlays Groups – General

Overlay groups support programs download in cases when code or data assembly file sections may overlap other code or data assembly file sections respectively. Linker script overlay groups support both **code** and **data overlays**. This is useful when a relatively small RAM program is available and used to run different applications or program sections downloaded from slow EPROMs located in the data space, at different run times. To create downloadable programs, a particular assembly file section may be linked twice: once in the program space (where it is downloaded to and executed) and once in the data space (where it is loaded from). Refer to **Linking Algorithm** topic for further details and examples.

Note: In the Overlays group one (or more) sections are defined as **noload**, so in the run-time these sections will be copied from code memory to data memory (or vice versa) by the ***movp*** or ***movd*** instructions respectively.

This chapter includes the following topics:

- **Code Overlays Groups**
- **Data Overlays Groups**

Notes and Limitations concerning Overlays

1. Assembly file sections defined in an overlay group may overlay only onto other assembly file sections in the same group. Assembly file sections defined in an overlay group may not overlay onto assembly file sections outside the group.
2. Neither directives nor attributes may be assigned to the overlay group as a whole, i.e. on the line containing the open brace (‘{‘). Directives and attributes can only be assigned to the individual assembly file sections that are members of the group.
3. The default location directives given to assembly file sections within the group are **at** and **next**.
4. Holes created inside an overlay group (as a result of using the **at** directive) cannot be filled by sections mapped externally the overlay group. That is, the section that follows the overlay group will be mapped by default after the highest address of the group.

5.6.1. Code Overlays Groups

Assembly file sections within a script's code-type block may be overlaid to allow multiple views of the same program address space. To overlay assembly file code sections, the sections must belong to an **overlay** group that can be viewed as one logical assembly file section. **Overlay** groups are created by surrounding the assembly file sections with braces **{ }**. An **overlay** group is restricted to fit inside its class boundaries (just as any normal assembly file section). Multiple **overlay** groups are allowed per class, but may not be nested or overlap each other. The next address following an **overlay** group is the highest next address of all the assembly file member sections. In terms of syntax, data and code **overlays** are identical. Note that to facilitate downloading, it is possible to use the Assembler **SIZEOF** and **INCODE** operators to obtain a section's size and refer to the symbol address in the program space linked into both program and data memory spaces. Refer to **Assembler Operators** and **Assembler Directives** for more details. Refer also to the **Download Support** topic for a typical downloadable application. The following example defines a code overlay:

code:

```
Reset at 0x0000
{
        Prog1
        Prog2 at Prog1
}
```

Refer to **Data Overlays Groups** for further details and limitations concerning **overlay** groups.

5.6.2. Data Overlay Groups

Assembly file sections within script data-type classes may be overlaid to allow multiple views of the same data address space, i.e. C-style unions. To overlay **.DATA** and **.DSECT** sections, these assembly file sections must belong to an **overlay** group, which can be viewed as one logical assembly file section. **Overlay** groups are created by surrounding the member assembly file sections with braces { }. An **overlay** group is restricted to fit inside its class boundaries (just as any normal assembly file section). Multiple **overlay** groups are allowed per class, but **overlay** groups may not be nested. The next address following an **overlay** group is the highest next address of all the assembly file member sections.

Figure 1 in the following page and the script file portion that follows provides an example of a data **overlay** group defined in memory class MyClass.

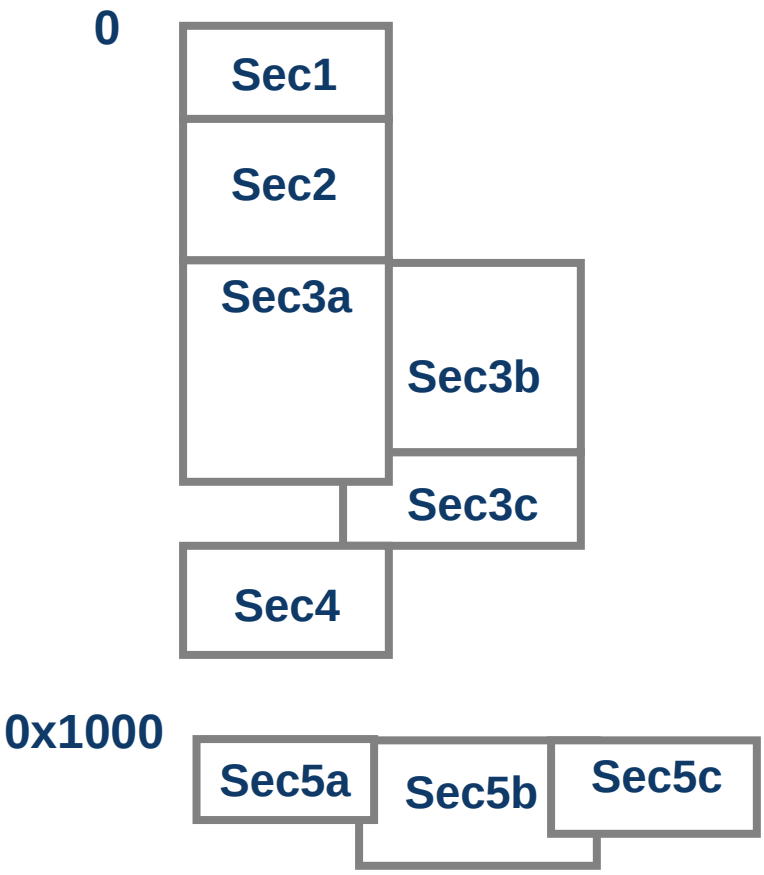


Figure 1: Overlay Group Example

The following script file portion implements the example illustrated in Figure 1

```
MyClass:    ;data type memory class
     Sec1
     Sec2
     {          ;open overlay group
       Sec3a
        Sec3b    at  Sec3a  noload
       Sec3c    noload
    }        ;close overlay group
     Sec4
    {        ;open overlay group
       Sec5a       at   0x1000
      Sec5b       at   Sec5a noload
       Sec5c       at   Sec5a noload

     }         ;close overlay group
```

5.7 Linking Algorithm

Following is the Linking Algorithm description for the **code** and **data** script sections:

1. The program space assembly file sections (i.e. **.CODE**, **.CSECT**, **.DATA** and **.DSECT** sections that are explicitly specified in a script code-type classes) are mapped in the order of appearance in the current code-type class. If an assembly file section has no associated location directives, it is mapped immediately **next** to the end of the previous assembly file section. If the **at** directive is used, the assembly file section will be mapped starting at the specified address. If the **align** attribute is used, the assembly file section will be mapped at the next address on the specified alignment boundary. If the **next** directive is used, it is mapped immediately after the end of the previous assembly file section. If the **lo** directive is used, mapping starts from the lowest address upwards. If the **smallest** attribute is used, the Linker will search for the smallest memory hole that can fit the section. Assembly file sections can be grouped together to form an **overlay group**. An overlay group is treated as if it is a logical assembly file section. Its size is measured between the lowest and highest addresses of all the assembly file sections in the group. An assembly file section following an **overlay group** will be located immediately next to the highest address of the group.

   Segments can have multiple declarations in the Linker script file. When a segment is declared using the **segment** keyword (with its own number), the next sections specified in the list are placed into that segment next to the last section placed into same segment (declared earlier) or according to any section's placement directive. Following the **segment** declaration, the first section defaults to the segment's address 0 (if no segment are specified). **Note** that every segment specified in the Linker script file represents a distinct address space, and thus possesses its own location counter.

2. Remaining **.CODE** and **.CSECT** sections that have not yet been mapped (i.e. those that are not mentioned in the script file) are mapped immediately next to the last assembly file section mapped into the program space (into the code class) according to rule 1 above. If no assembly file sections were specified in the **code** class, then mapping begins at the lowest address. The remaining **.CODE** and **.CSECT** sections are mapped in order of appearance within the object files list found in the **objects** block.

3. The data space assembly file sections (i.e. **.DATA**, **.DSECT**, **.CODE** and **.CSECT** sections explicitly specified in the data script block) are mapped in the order of appearance in the current segment. If an assembly file section has no associated location directives, it is mapped immediately next to the end of the previous assembly file section. If the **at** directive is used, the assembly file section will be mapped beginning at the specified address. If the **align** attribute is used, the assembly file section will be mapped at the next address on the specified alignment boundary. If the **inpage** attribute is used, the Linker will check that the assembly file section does not cross a physical page boundary (any address of type 0xYY00). If the **next** directive is used, it is mapped immediately next to the end of the previous assembly file section. If the **lo** directive is used, mapping starts from the lowest address upwards. Assembly file sections can be grouped together to form an overlay group. The assembly file section group is treated as if it is a logical assembly file section. Its size is measured between the lowest and highest addresses of all the assembly file sections within the group. An assembly file section following an overlay group will be located immediately next to the highest address of the group.

4. Remaining **.DATA** and **.DSECT** sections that have not yet been mapped (i.e. those that are not explicitly mentioned in the script file) are placed immediately next to the last assembly file section specified and mapped into the data space (in the data class) according to rule 3 above. If no assembly file sections are specified in the script **data** class, mapping begins at the lowest address. The remaining **.DATA** and **.DSECT** sections are mapped in order of appearance in the ordered object files list.

5. **Data** and **code overlays** are allowed only when explicitly specified. Implicit overlays resulting from the use of **at** or **align** attributes will generate an error. **Note** that overlaying of address ranges can occur when two assembly file sections are placed into two different segments, since every segment possesses its own address space.

6. The Linker issues an error when a relocation size conflict is detected, i.e., the size of a symbol's address after having been fully resolved requires more bits than available in the operand field of the instruction (e.g. ***brr*** and ***callr*** instructions).
   In another example:
   ```
   mov     #label, operand
   ```
   If the **label's** address above is located beyond 0xff then an error will be generated.

7. The Linker issues a warning when section **inpage** attribute is violated. This happens when a data assembly file section (associated with the **inpage** attribute) crosses a physical page boundary (every 256 words is a defined as a physical page, so every address 0xYY00 is considered such a page boundary).

8. The Linker will mark assembly file sections associated with the **noload** attribute, so that the loader of the Debugger will not load that assembly file section into memory at load time.

5.8 Output Files

The Linker outputs an object file (**.a** extension) and a listing file (**.lin** extension). The **MPP** (macro-preprocessor) can be activated independently to view its output for pre-Linker debugging purposes.

Linker's build number and command line are printed to the listing file:

The Linker prints its build number and the command line in the banner of the file. For **Example**:

COFF Linker 9.1.0 (build #141) (C) 2005 CEVA Inc. All rights reserved.

File: file.lnk Page: 1

Command line: %tkltools%\cofflnk.exe -m file.lnk

5.9 Linker's Comprehensive Multi Program Paging Support

Not relevant for CEVA-X and CEVA-TeakLite-III:
This chapter includes the following items:

- Introduction
- How to Build a Multi Program Paging Application?
- Pointers to C/C++ Functions
- Assembly Application Related Notes
- Debugging Related Notes

5.9.1. Introduction

Program (and data) paging was already supported by SDT version 8.4 and older (down to 8.0). Applications could be partitioned into multi program pages while the user was required to perform the following steps:

1. Map the different routines into the different program segments / pages as needed.
2. Write an intermediate routine that switches to the target segment where the "far" (crossing segment / page) routine is located and calls the target routine. After the far routine completion, the intermediate routine is responsible for switching back to the original source segment and return address.
3. Map the intermediate switching routine in all the program segments / pages in the same address (possibly in a shared memory area).
4. Go back to the source code and replace manually each far routine call into a call to the intermediate routine.

V8.5 Linker (under the -multiProgPage command line option) takes this process few steps ahead by automating steps 2-4 (step 1 is planned to be automated as well in a future version). That is, the user is only required to map his application to the different segments. The rest is done automatically by the Linker - An intermediate (customizable) switching routine is already implemented while the Linker automatically detects and replaces every far call by the intermediate routine call.

The solution is solely in the Linker level - **No source code modifications are required!!!**

An example for an application that uses the Linker's Comprehensive Program Paging support can be found in the tools' *examples* directory.

The Linker's comprehensive program paging support was designed primarily for C/C++ applications (compiled under the –mmulti-prog-page C/C++ Compiler command line option), nevertheless, any Assembly application that conforms with the C/C++ Compiler calling convention can use this feature as well. For more information see the Assembly Application Related Notes chapter.

Memory overhead when using the Linker's multi program paging:

Implementation of the multi program paging consumes the following memory resources:

1. **The switch routine –** located in the program memory.
   This is a constant size (less than 100 words) regardless of the number of cross function calls. Furthermore, it can be reduced (The assembly file is supplied) if some of the functionality is not needed, for instance calling functions with unknown number of arguments, such as printf.
2. **The jump table –** located in program memory.
   The additional code is only for the called function that actually crosses a page (functions that needs to be accessed from a different page). It requires 2 words for every such function.
3. **Data auxiliary table –** located in data memory.
   The additional data is only for cross pages calls and its 2 words for every function that needs to be accessed from a different page.

In total, memory requirement calculation is as follows:

Let N be the number of functions that are being called from a different page then their own, then the total memory expense will be:

- Program memory = ~100 words + 2 * N
- Data memory = 2 * N.

5.9.2. How to Build a Multi Program Paging Application

The Linker is capable of mapping sections; hence, the user application should be broken into unique sections, each including one function or routine. This allows maximal flexibility at the mapping stage as each function can be mapped separately.

Compiling the Application:

C/C++ applications should be compiled with the **–mmulti-prog-page** Compiler's command line option. This option instructs the Compiler to plant extra debugging information for the Linker to determine the number of parameters passed to the function via the stack. In addition, it puts every function in a unique section name and generates a unique prog_data section name (prog_data section contains a jump table for functions that include a C/C++ switch statement). This is extremely important since the function and its' prog_data must be mapped in the same segment.

The **–c** Compiler command line option that stops the Compiler driver before the linking stage should also be applied. This enables the user to customize the Compiler's default Linker script file and map his functions to the different segments according to his needs.

Preparing the Linker Script File:

In order to create a Linker script file, it is always recommended to start from the Compiler default Linker script file and to customize the generated file. The Compiler default script file can be achieved by running the Compiler (without the **–c** option) on a simple C/C++ (dummy) program with the exact command line options the real program will be compiled with, and then to use the generated Linker's listing file (.lin) to create the default script file.

Note that different compilation options (e.g. -emul, -mfile-io, -mffp, -mindexed-mailbox, -mteak-cpcX, etc') may lead to different linking script files (generated by the Compiler), thus, it is important to use the relevant options of the final

application when creating a default Linker's script file through the Compiler – there is not ONE default Linker script file.

After the basic Linker script file was generated, the user should adapt the file as follows:

- Add the application's objects to the objects list (between crt0.o and crtn.o)
- Add the **mprgpage.o** object to the objects list (between crt0.o and crtn.o). The file is located in the tools' libs directory.

  Note that the mprgpage.asm source file is provided as well in the tools' libs directory for user customization. The file implements the following:

  - The intermediate paging switch routine that is called whenever a far function call is found by the Linker in the user's application (parts of this routine may be changed by the user, see the source code for more details).
  - Two template tables (empty sections) that are updated by the Linker and used by the paging switch routine during run-time. These tables hold far functions' information (These tables should not be changed by the user).

  Note: The source file contains parts that are highly dependent on the hardware paging mechanism, which may be customized by the user. Nevertheless, some parts are dependent on the Linker's implementation and handle the information tables prepared by the Linker. These parts are not to be changed by the user.
- Map the application's sections to the different segments and addresses as required using the standard mapping directives / attributes and the Linker script keywords.

Linking the Application:

Apply the **-multiProgPage** command line option when invoking the Linker. This option instructs the Linker to **automatically** do the following:

- Build the tables that will be used by the paging switch routine during run-time. These tables include information of functions that are called from a segment different from the one they are mapped in.
- Map these sections (tables) to all the segments in the same address. This is done after all explicitly mapped sections are mapped.
- Replace each "far" (crossing segment) call by a call to the intermediate paging switch routine (through a jump table).

5.9.3. Pointers to C/C++ Functions

C/C++ function pointers can be used freely to reference functions in other segments. However, since a pointer can be loaded with a function address in one segment and used later in another, all function calls through pointers are considered as **far** (crossing segment) regardless of the source and target segments. This behavior will be optimized in a future version to optionally allow the user full control of pointer references by applying a special C/C++ or Assembly level directives.

5.9.4. Assembly Application Related Notes

In order to allow the Linker to automatically adjust Assembly routine calls to "far" calls the following conditions must be met:

- All arguments must be passed to the Assembly routine via accumulators without using the stack for this purpose. Since the switch routine preserves the C/C++ compiler calling convention this actually restricts all assembly functions to receive at most 4 arguments. If this condition cannot be met the paging switch routine must be manually adapted to the special calling convention.
- The Assembly routine must be declared by a global label (using the .GLOBAL Assembler directive).

5.9.5. Debugging Related Notes

The intermediate paging switch routine found in **mprgpage.asm/o** is compiled without the **–g** Assembler's command line option. In Simulation mode, this makes the debugging smooth when the user wants to jump directly to the far target function without stepping through the switching routine. However, in Emulation mode, the Debugger behavior is slightly different (the Debugger's default is 'enable source fast step', see the CEVA-Toolbox *Debugger Reference Guide* for more details). That is, if the user executes 'source step' on the "call far_function" instruction, the Debugger behaves as 'source next'. This is because between the caller source which includes debug info and the callee source of the far_function which includes debug information as well, there is the code of the switching routine that wasn't assembled with debug information (by default). For resolving this issue, 2 options are provided:

- Execution of the 'disable source fast step' CLI command in the Debugger level, which might result in slower steps.
- Assembling the switching routine with the **–g** Assembler command line option, which will result in stepping through the switching routine code as well.

5.10 Post Linker optimization

An additional optimization pass is added to the Linker (SDT V9.1) for code size reduction (in future releases cycle count reduction will be added as well). The new optimization pass is performed at the last stages of the linking process. The new optimizer module takes a global view of the application and therefore is able to perform optimizations in which the Compiler cannot due to its overall picture limited to a single file. The new Post-Linker optimization pass is activated by default and can be disabled by applying the Linker's '-noOs' command line option. The code size reduction factor that is achieved by this optimization pass can vary between one to double digits determined by the application nature and its mapping. The Linker's '-Oinfo' command line info can be used for printing a report to the listing file.

5.10.1. Code Size Optimization

The Post Linker optimization pass performs various code size oriented optimizations. These optimizations can significantly reduce the code size of the application by a factor varies between one to double digits according to the application type and its mapping. Combined with the CEVA-X complier code size optimization switches, the final COFF file generated targets for minimum code size. Refer to the Complier manual for more details on the code size related switches.

An example of one of the Post Linker optimizations:

When labels are used inside assembly code, the Assembler assumes that the address of the label is the maximum value that can be encoded - a 32 bit number. This assumption causes the Assembler to add a 32 bit immediate extension to each instruction using a label, when in fact the label value at link time can be rather smaller and there is no need for the extension. In other words the label can fit within the instructions' immediate bits. From SDT V9.1 on, the Linker adds a dedicated pass on the linked object files, to locate these unnecessary extensions. When found, the Linker replaces the long instruction packet encoding with the optimized instruction packet taking into account the actual value of the address label used in the instruction that is known at link time.

6. COFF Utilities

This chapter includes the following topics:

- **The Cofflib – Generating Library Files**
- **The Coffutil – Extracting information from the COFF**
- **The Coffdump – Generating Dump Files**
- **The Intelhex – Generating PROM Burnable Files**
- **The Hex2dmc & Hex2rom – Creating DMC and ROM Downloadable formats**

6.1 The Cofflib – Generating Library Files

It is possible to create libraries from one or more object files using the **COFFLIB** utility. This utility converts object files (.O) into library files (.LIB) that are linkable binary files that combine together one or more objects.

Following is the Cofflib invocation format:

cofflib CommandOption LibraryName

where *CommandOption* can be selected from Table 3 in the next page.

Table 3 Cofflib Command Options

Command Option	Description
-a ObjectFilesList	Adds the specified modules to the library.
-findSymbol SymbolName	Display the name of the object that contains the given symbol.
-h	Displays Help information.
-quiet	Do not print the banner message.
-r ObjectFilesList	Removes the specified modules from the library.
-s ObjectFile1 ObjectFile2	Substitutes module1 with module2 in the library
-secNames	Display all the sections names found in the library
-v	Views the library contents, date and version
-x ObjectFilesList	Extracts the specified module from the library
-xAll	Extracts all the modules from the library

For example, suppose you have 3 files, each containing two library functions:

file1.asm produces *file1.o* containing *func1a* and *func1b*

file2.asm produces *file2.o* containing *func2a* and *func2b*

file3.asm produces *file3.o* containing *func3a* and *func3b*

Then the following command creates a library named *MYLIB.LIB* with these objects/functions:

cofflib -a *file1.o file2.o file3.o MYLIB.LIB*

The following command verifies the results:

cofflib -v *MYLIB.LIB*

which should return the following list:

Module file1.o (time: Thu May 15 15:45:27 2003, version: 8.5)

size: 1683, offset: 296

format: new (compatible with version 8.0 and up)

func1a

func1b

Module file2.o (time: Thu May 15 15:45:27 2003, version: 8.5)

size: 1453, offset: 2196

format: new (compatible with version 8.0 and up)

func2a

func2b

Module file3.o (time: Thu May 15 15:45:27 2003, version: 8.5)

size: 1535, offset: 3896

format: new (compatible with version 8.0 and up)

func3a

func3b

It is possible to retrieve an object file from an archived library file using the **–x** / -xAll (extract**)** command line options. For example: to get back file2.o from the above library, enter:

cofflib -x *file2.o MYLIB.LIB*

Finding the object in the library containing a symbol.
Assuming the object *file2.o* has s a symbol _foo then for the following command line:

cofflib -findSymbol *_foo MYLIB.LIB*

Will return:
The symbol *_foo* is found in object *file2.o*

Display all section names found in the library objects.
Assuming *file1.o* has only section *.text* in it, then for the following command line:

cofflib -secName *MYLIB.LIB*

Will return:

Sections list for object *file1.o*:

.text

Notes:

1. **It is prohibited to put two files with identical names in a same library.**

2. **The -x command line option creates a copy of the file(s) only. The extracted file(s) remains intact in a library.**

3. **Cofflib adds section names found in objects to the publics stored in the library. This allows the user to reference section names stored in libraries from his object files.**

6.2 The Coffutil – Extracting Information from the COFF

With the help of **COFFUTIL**, it is possible to generate information about ***.a** file contents. Table 4 specifies the options that can be used with this utility.

Table 4 Coffutil Command Line Options

Option	Description
-addSecName	Includes section name & size in the output raw-data file.
-b basename	For multi-paging application, **basename** and the page number will be used for the output files (file for each page)
-c	Extracts code and data memory sections contents (raw-data).
-incNoLoadSec	Includes section contents in the raw-data file, even if mapped with the '**noload'** mapping attribute.
-o	Used an object file (*.o) as an input file.
-quiet	Do not print the banner message.
-s	Generates all symbol table information in .map file.
-spilt	Splits code and data to different files.
-tl3_to_tl	Converts the CEVA-TL3210 out format into the segmented format of the CEVA-Teaklite, assuming 64KW per segment.
-?	Displays Help information.
-h	Displays Help information.

Notes:

1. *In a multi segment program, the usage of the –**c** option will generate an output file for each segment. Each output file generated will have the **.out** name (default name) concatenated to the segment index.*
   Example:
   Suppose MyFile.a contains 3 segments (0,1, 2), the output of coffutill will be:
   MyFile0.out
   MyFile1.out
   MyFile2.out
2. *Use the **-b** option in order to change the default name*
3. *If no segments are used (meaning, all memories contents will be in segment 0), the output will be dumped also to the standard output*

6.3 The Coffdump – Generating Dump Files

It is possible to generate a dump of an executable / object file using the **coffdump** application. This utility parses executable / object files and interprets the binary COFF format into a readable ASCII format.

The **coffdump** utility provides the following information:

1. **Coff header values** – all general file information including creation date, DSP Core type, tools version, number of segments and so forth.
2. Sections **– details all sections including headers, data, reallocation entries and line table.**
3. Symbol/Strings entries **– all symbols / strings found in the COFF file. For instance: label names.**

Table 5 specifies the options that can be used with the coffdump utility.

Table 5 Coffdump Command Line Options

Option	Description
-quiet	Do not print the banner message.
-noInfo	Used **Coffdump** in the old format (prior to 8.4).
-i	Enable to view the coff information with an ini viewer (the output file must have .ini extension).
-h	Displays Help information.

6.4 The Intelhex – Generating PROM Burnable Files

Most EPROM programmers do not accept executable files as input. A utility can be used to convert executable files **(.a)** into byte-wise Intel-Hex format files (.hcl,.hch,.hdl,.hdh). The conversion can be activated in two ways:

- **Via batch file**
- **Directly from the command line**

6.4.1. Generating PROM files Via the Batch File – Coff2hex

Coff2hex batch file is used to create Intel-Hex format files according to the specific DSP Core (identified automatically):

- **coff2hex** CoffExecutableBaseFileName [options] [adj]

Where, **CoffExecutableBaseFileName** is the base name of an executable file without the mandatory (.a) extension.

Table 6 specifies the options that can be used with the **coff2hex** utility.

Table 6 Coff2hex Command Line Options

Option	Description
-**nosplit**	Generates low & high bytes in the same output file.
-**nosplitSwapBytes**	Similar to **nosplit**, but bytes are swapped (high before low)
-incNoLoadSec	Include sections contents even if mapped with the 'noload' attribute.
-o	Specifies the output file names
-h	Displays Help information.
-recordWidth N	Sets the record width to be (1..255, default is 16) Terminology: Record – one line of data in an intelhex format file. Allow the determination of the number of data elements in one record, and thus control the final width of each record in the output file

The output from **Coff2hex.bat** is as follows:

- When using one of the nosplit options:
  1. CoffExecutableBaseFileName.hc
  2. CoffExecutableBaseFileName.hd
- When used without any of the nosplit options:
  1. CoffExecutableBaseFileName.hcl
  2. CoffExecutableBaseFileName.hch
  3. CoffExecutableBaseFileName.hdl
  4. CoffExecutableBaseFileName.hdh

6.4.2. Generating PROM Files Directly from the Command Line – Intelhex

To convert directly from the command line, use the program **Intelhex.exe** (DOS version) or **intelhex** (Linux version). The input must be prepared in the appropriate format, that is, an ordered list of hexadecimal addresses NNNN (with c: or d: prefix for indicating the memory space) followed by the hexadecimal value MMMM as described below:

C:NNNN MMMM

C:NNNN MMMM

...

D:NNNN MMMM

D:NNNN MMMM

...

The **Coffutil** program can be used to obtain this file from the binary executable file. Following is a utilities sequence example for the above utilities including **Sorthex**.

Sorthex utility rearranges **Coffutil** output by separating and sorting code and data:

coffutil -c *CoffFile > DataFile*

sorthex *DataFile SortFile*

intelhex *SortFile*

The output files from **Intelhex** program are the four byte-wise INTEL-HEX files shown above. Normally, there is no reason to directly invoke this program from the command line and it is usually used through the Coff2hex batch file.

6.5 The Hex2dmc & Hex2rom – Creating DMC and ROM Downloadable formats

Following are stages performed by the **Hex2rom** and **Hex2dmc** batch files:

1. Isolate **code** and **data** memory sections' contents

coffutil -c *FileName.a > FileName.hex*

2. Apply **Sorthex** utility in order to merge and sort data & code memory according to the specific DSP Core:

sorthex *FileName.hex FileName.srt*

Note: *The structure of sorted files (following Sorthex) is as follow:*
code *memory is sorted according to addresses in ascending order (from low to high) followed by sorted* ***data*** *memory according addresses ascending order (from low to high).*

3. Apply **Hex2dmc** utility in order to create DMC data format according to the specific DSP Core:

hex2dmc *FileName.srt FileName.dmc*

4. Apply **Hex2rom** utility in order to create ROM data format according to the specific DSP Core:

hex2rom FileName.srt FileName.rom
The user may select either of the following batch files to perform the stages described above. These batch files generate ROM and DMC formats.

The following items are further described:

- Generating ROM format from .lnk file using LINK2ROM batch
- Generating DMC format from .a file using COFF2DMC batch
- Generating ROM format from .a file using COFF2ROM batch

6.5.1. Generating ROM Format from .lnk File using LINK2ROM Batch

The Link2rom batch file invokes the Linker and then generates ROM data for the linked coff.

Usage:

link2rom [options] LinkerScriptFileBaseName [**-splitOddEven**]

Where **LinkerScriptFileBaseName** is the base name of the Linker script file without the mandatory (.lnk) extension.

Table 7 specifies the options that can be used with the **link2rom** utility.

Table 7 Link2rom Command Line Options

Option	Description
-p	Invokes MPP before linking.
-m	Creates sections map.
-s	Creates symbol table.
-splitOddEven	Splits results to two files of odd and even addresses.
-x	Creates symbol cross-reference.
-h	Displays Help information.

6.5.2. Generating DMC format from .a file using COFF2DMC Batch

Usage:

coff2dmc CoffExecutableBaseFileName [options] [-o filename]

Table 8 specifies the options that can be used with the **link2dmc** utility.

Table 8 Coff2dmc Command Line Options

Option	Description
-o	Specifies the output files names
-splitOddEven	Splits DMC results to odd and even addresses
-incNoLoadSec	Include sections contents even if mapped with the 'noload' attribute.
-h	Displays Help information.

The output from **Coff2dmc.bat** is:

- basefileN.DMC N - segment number , DMC format output file
- \ Nd.DMC d - for data segments.

When using **-splitOddEven** flag:

- basefileN.odd odd - low bytes files
- \ N.evn evn - high bytes files
- \ Nd.odd same for data segments
- \ Nd.evn

6.5.3. Generating ROM format from .a File using COFF2ROM

Batch

Usage:

coff2rom CoffExecutableBaseFileName [options] [-o filename]

Table 9 specifies the options that can be used with the **link2rom** utility.

Table 9 Coff2rom Command Line Options

Option	Description
-o	Specifies the output files names
-splitOddEven	Splits DMC results to odd and even addresses
-incNoLoadSec	Include sections contents even if mapped with the 'noload' attribute.
-h	Displays Help information.
CEVA-X related options:	
-dwidth[8/16]	Data element width in output file.
-cwidth[8/16]	Code element width in output file.
-uwidth[8/16]	Unified element width in output file.
	Default is 8 (byte).
-dbank[8/16/32]	Number of data memory banks (output files).
-cbank[8/16/32]	Number of code memory banks (output files).
-ubank[8/16/32]	Number of unified memory banks (output files).
	Default is 16 (byte in each bank)
-dsplit	Output each data block into a different file.
-csplit	Output each code block into a different file.
-usplit	Output each unified block into a different file.

The output from **Coff2rom.bat** is:

basefileN.ROM N - segment number, ROM format output file

\ Nd.ROM d - for data segments.

When using **-splitOddEven** flag:

basefileN.odd odd - low bytes files

\ N.evn evn - high bytes files

\ Nd.odd same for data segments

\ Nd.evn

7. CEVA-TL321x Translator

Relevant for CEVA-TL-321x core only:

This tool allows translating assembly files from CEVA-Teak, CEVA-TeakLite and CEVA-TeakLite-II to CEVA-TL321x assembly.

The translator can be used in three ways:

1. As a standalone utility:

   tl3translator.exe [-options] <inputAsmFile> <outputAsmFile> in this way the assembly output file will contain CEVA-TL321x assembly syntax.

2. Via the CEVA-TL321x assembler:

   tl3asm [-teak|-teaklite] <inputAsmFile> in this way an object file with CEVA-TL321x binary code will be created.

3. Via the CEVA-TL321x compiler:

   tl3cc [-mteak|-mteaklite] <inputCFile> in this way the compiler will create a CEVA-Teak/CEVA-TeakLite assembly file, and it will use the translator to create a CEVA-TL321x object file.

The table below specifies the options that can be used by the Translator:

Table 10 Translator Command Line Options

Option	Description
-teaklite	Use this option when translating CEVA-TeakLite/CEVA-TeakLite-II assembly files. This is the default behavior of the translator.
-teak	Use this option when translating CEVA-Teak assembly files.
-quiet	Do not print the banner message.
-removeOriginalInstruction	Do not print in remark the translated instruction

The table below specifies the directives that the Translator can use while processing a file:

Table 11 Translator Line Directives

Directive	Description
.BEGIN_IGNORE_WARNINGS	Directs the Translator not to issue warnings that are encountered while processing the file
.END_IGNORE_WARNINGS	Directs the Translator to resume issuing warnings
.BEGIN _IGNORE_ TRANSLATE **.END_IGNORE_ TRANSLATE**	Assembly code that is surrounded with those directives will not be translated (assuming it is pure CEVA-TL321X assembly code).

General information:

When translating assembly files from CEVA-Teak, the translator may issue errors and warnings.

- In order to suppress these warnings user may use the directives: .BEGIN_IGNORE_WARNINGS and .END_IGNORE_WARNINGS surrounding the code that triggered the warnings.
- When the translator issues an error, it also suggests how to fix the error.
  For example:
  When trying to translate the Teak instruction **set ##0x8000, mod2 (enable bitrev** for r7), or any instruction that uses modX, the translator will issue the error:
  Instruction that uses modX\sttX registers cannot be translated.
  Please manually translate it to use TeakLite-III configuration registers
  User should find the correct configuration register, and set the correct bit.
  For the mentioned above instructions it should be:

  ; Original Teak assembly code: set ##0x8000, mod2
  .BEGIN_IGNORE_TRANSLATE
  bitrev r7
  .END_IGNORE_ TRANSLATE

8. VGEN Support

8.1 Introduction

CEVA-XC VGEN instructions can be described by listing the actual operations each instruction performs. This chapter describes an easy way to use macro type assembly syntax to describe VGEN operations, which the Assembler converts to actual VGEN instructions, including settings of the configuration registers. In addition, the Debugger is able to show the VGEN operations of each VGEN instruction, in the same assembly like macro syntax.

8.2 Assembler Support

The Assembler supports generation of VGEN instructions and initialization of the relevant configuration registers by using macro type syntax.. Each instruction has a unique way to describe its operations, according to the *CEVA-XC Architecture Specification.*

Several instructions require more than one line to describe a single operation. The number of lines needed for each instruction to describe a single operation is provided in the VGEN syntax below.

8.3 General VGEN Syntax

VGEN_<Type> (<operation type>, <configuration register>[,<section name>]) [
    <operation 1>
    …
    <operation n>
]

↓Translated to

<vmovi 0>
…
<vmovi 5>
<vgen operation> {<number of operations>, <configuration registers type>} vix, viy, voz

8.4 VGEN Types

1. MOV
   Generates *vmovi* instructions that set the configuration registers with the appropriate encoding of the requested operations
2. MOV_OP
   Generates *vmovi* instructions that set the configuration registers with the appropriate encoding of the requested operations, and the corresponding *v*gen instruction with the requested resources
3. MEM
   Generate data declarations that hold the encoding needed to set the configuration registers – according to the requested operations

8.5 Instructions Description

8.5.1. vgenmac3

General Information

Single Line syntax: vozN = [0 | vozN] +/- vixNp*viyMp +/- vixKp*viyUp
Lines for operation: 1

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number need to match the operation number

Encoding

vozN =	[0 | vozN]	+/-	vixNp	*	viyMp	+/-	vixKp	*	viyUp
	(A(i))	(S(i))	(P(2i))		(P(2i+16))	(S(i+8))	(P(2i+1))		(P(2i+17))

Example 1

```
VGEN_MOV_OP (mac3, vig) [
voa0    =       voa0    -   via1l*vib2h     +       via2h*vib5l
voa1    =       0           +       via7h*vib0l     -       via3l*vib6h
]
```

↓Translated to

```
vmovi #0x00006F52, vig0
vmovi #0x00000000, vig1
vmovi #0x0000D0A5, vig2
vmovi #0x00000000, vig3
vmovi #0x201, vig4
vmovi #0x1, vig5
vgenmac3 {2op, cfgg} via, vib  ,voa0
```

Example 2

```
VGEN_MOV (mac3, vie) [
        voa0    =       voa0    - via1l*vig2h + via2h*vig5l
        voa1    =       0       + via7h*vig0l - via3l*vig6h
]
```

↓Translated to

```
vmovi #0x00006F52, vie0
vmovi #0x00000000, vie1
vmovi #0x0000D0A5, vie2
vmovi #0x00000000, vie3
vmovi #0x201, vie4
vmovi #0x1, vie5
```

Example 3

```
VGEN_MEM (mac3, vih, my_data_sect_01) [
        voa0    =       voa0    - via1l*vig2h + via2h*vig5l
        voa1    =       0       + via7h*vig0l - via3l*vig6h
]
```

↓Translated to

```
DSECT  my_data_sect_01
DW 0x00006F52
DW 0x00000000
DW 0x0000D0A5
DW 0x00000000
DW 0x201
DW 0x1
```

8.5.2. vgenasu (v1)

General Information

Single Line syntax: vozN = +/- {vixNp, vixMp} +/- {viyKp, viyUp}
Lines for operation: 1

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number

Encoding

```
vozN =  +/-      {vixNp   ,  vixMp}     +/-         {viyKp     ,viyUp}
        (S(i))   (P(2i+1))   (P(2i))    (S(i+8))   (P(2i+17))  (P(2i+16))
```

Example

```
VGEN_MOV_OP (asu, vig) [
        voa0     =          - {via1l,via6h} + {vib6l,vib2l}
]
```

↓Translated to

```
vmovi #0x0000002D, vig0
vmovi #0x00000000, vig1
vmovi #0x000000C4, vig2
vmovi #0x00000000, vig3
vmovi #0x1, vig4
vmovi #0x0, vig5
vgenasu {1op, cfgg} via, vib  ,voa0
```

8.5.3. vgenasu (v2)

General Information

Single Line syntax: vozN = voxN +/- {viyNp, viyMp}
Lines for operation: 1

Limitations and Additional Information

- vox and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vox registers and the same 'y' for all viy registers used
- Destination vector number need to match the operation number

Encoding

```
vozN = voxN        +/-     {viyNp  ,  viyMp}
                 (S(i+8))  (P(2i+17))  (P(2i+16))
```

Example

```
VGEN_MOV_OP (asu, vig) [
          voa0    =    voa0       + {vib6l,vib2h}
]
```

↓Translated to

```
vmovi #0x00000000, vig0
vmovi #0x00000000, vig1
vmovi #0x000000C5, vig2
vmovi #0x00000000, vig3
vmovi #0x0, vig4
vmovi #0x0, vig5
vgenasu {1op, cfgg} voa0, vib  ,voa0
```

8.5.4. vgenasu2w (v1)

General Information

Single Line syntax: vozNp = +/- vixNp +/- viyMp
Lines for operation: 2

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number
- Destination part must start with '0l' in the first line, and advance in each line: '0h','1l', '1h', etc.

Encoding

vozNp =	+/-	vixNp	+/-	viyMp
	(S(2i+offset))	(P(2i+offset))	(S(2i+16+offset))	(P(2i+16+offset))

Example

```
VGEN_MOV_OP (asu2w, vih) [
        voa0l    =  - vib0l    + vic0h
        voa0h  =  - vib1l   + vic1h
        voa1l    =  - vib2l    + vic2h
        voa1h  =  - vib3l   + vic3h
        voa2l    =  - vib4l    + vic4l
        voa2h  =  - vib5h   + vic5h
        voa3l    =  - vib6l    + vic6l
        voa3h  =  - vib7h   + vic7h
]
```

↓Translated to

```
vmovi #0xFCB86420, vih0
vmovi #0x00000000, vih1
vmovi #0xFCB87531, vih2
vmovi #0x00000000, vih3
vmovi #0xFF, vih4
vmovi #0x0, vih5
vgenasu2w {4op, cfgh} vib, vic  ,voa0
```

8.5.5. vgenasu2w (v2)

General Information

Single Line syntax: vozNp = voxNp +/- vixNp
Lines for operation: 2

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vox registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number
- Destination part must start with '0l' in the first line, and advance in each line: '0h','1l', '1h', etc.

Encoding

vozNp = voxN +/- vixNp
(S(2i+16+offset)) (P(2i+16+offset))

Example

```
VGEN_MOV_OP (asu2w, vih) [
        voa0l   = vob0l   + vic0h
        voa0h   = vob0h   + vic1h
        voa1l             = vob1l   + vic2h
        voa1h   = vob1h   + vic3h
        voa2l             = vob2l   + vic4l
        voa2h             = vob2h   + vic5h
        voa3l             = vob3l   + vic6l
        voa3h   = vob3h   + vic7h
]
```

↓Translated to

```
vmovi #0x00000000, vih0
vmovi #0x00000000, vih1
vmovi #0xFCB87531, vih2
vmovi #0x00000000, vih3
vmovi #0x0, vih4
vmovi #0x0, vih5
vgenasu2w {4op, cfgh} vob0, vic  ,voa0
```

8.5.6. vgenlin2w

General Information

Single Line syntax: vozNp = vixNp*viyMp[b0] + {viyMp[b1],0x00/0x0000}
Lines for operation: 2

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- viy appears twice in each line – both appearances need to use the same vector number and part
- Destination vector number required to match the operation number
- vix part must start with 'nl' in the first line, and advance in each line: 'nh','(n+1)l', '(n+1)h', etc - cyclic from 7 to 0.
- Using 0x0000 turns on the 'shf16' switch

Encoding

vozNp = vixNp * viyMp[b0] + {viyMp[b1], 0x00/0x0000}
(S(2i+offset))

Example

```
VGEN_MOV_OP (lin2w, vih) [
        voa0l   =  vib5l*vig6l[b0] + {vig6l[b1], 0x00}
        voa0h   =  vib5h*vig5l[b0] + {vig5l[b1], 0x00}
]
```

↓Translated to

```
vmovi #0x00000000, vih0
vmovi #0x00000000, vih1
vmovi #0x000000BA, vih2
vmovi #0x00000000, vih3
vmovi #0x0, vih4
vmovi #0x0, vih5
vgenlin2w {1op, cfgh} vib5, vig  ,voa0
```

8.5.7. vgenlinp

General Information

Single Line syntax: vozN/vizN[b0-3] = (vixNp*viyMp[b0]
+{viyMp[b1],0x00/0x0000})[15:8/23:16]
Lines for operation: 4

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- viy appears twice in each line – both appearances are required to use the same vector number and part
- Destination vector number need to match the operation number
- vix part must start with '0l' in the first line, and advance in each line: '0h','1l', '1h', etc.
- Using 0x0000 turns on the 'shf16' switch
- Using '23:16' turns on the 'h' switch`

Encoding

vozN/vizN[b0-3] = vixNp * viyMp[b0] + {viyMp[b1], 0x00/0x0000})[15:8/23:16]
(S(4i+offset+16))

Example 1

```
VGEN_MOV_OP (linp, vie) [
        voa0[b0]  =  (vib7l*vig6l[b0] + {vig6l[b1], 0x00})[15:8]
        voa0[b1]  =  (vib7h*vig5l[b0] + {vig5l[b1], 0x00})[15:8]
        voa0[b2]  =  (vib7l*vig6l[b0] + {vig6l[b1], 0x00})[15:8]
        voa0[b3]  =  (vib7h*vig5l[b0] + {vig5l[b1], 0x00})[15:8]

        voa1[b0]  =  (vib0l*vig6l[b0] + {vig6l[b1], 0x00})[15:8]
        voa1[b1]  =  (vib0h*vig5l[b0] + {vig5l[b1], 0x00})[15:8]
        voa1[b2]  =  (vib0l*vig6l[b0] + {vig6l[b1], 0x00})[15:8]
        voa1[b3]  =  (vib0h*vig5l[b0] + {vig5l[b1], 0x00})[15:8]
]
```

↓Translated to

```
vmovi #0x00000000, vie0
vmovi #0x00000000, vie1
vmovi #0xACACACAC, vie2
vmovi #0x00000000, vie3
vmovi #0x0, vie4
vmovi #0x0, vie5
vgenlinp {2op, cfge} vib7, vig  ,voa0
```

Example 2

```
VGEN_MOV_OP (linp, vie) [
        voa0[b0]  =  (vib7l*vig6l[b0] + {vig6l[b1], 0x00})[23:16]
        voa0[b1]  =  (vib7h*vig5l[b0] + {vig5l[b1], 0x00})[23:16]
        voa0[b2]  =  (vib7l*vig6l[b0] + {vig6l[b1], 0x00})[23:16]
        voa0[b3]  =  (vib7h*vig5l[b0] + {vig5l[b1], 0x00})[23:16]

        voa1[b0]  =  (vib0l*vig6l[b0] + {vig6l[b1], 0x00})[23:16]
        voa1[b1]  =  (vib0h*vig5l[b0] + {vig5l[b1], 0x00})[23:16]
        voa1[b2]  =  (vib0l*vig6l[b0] + {vig6l[b1], 0x00})[23:16]
        voa1[b3]  =  (vib0h*vig5l[b0] + {vig5l[b1], 0x00})[23:16]
]
```

↓Translated to

```
vmovi #0x00000000, vie0
vmovi #0x00000000, vie1
vmovi #0xACACACAC, vie2
vmovi #0x00000000, vie3
vmovi #0x0, vie4
vmovi #0x0, vie5
vgenlinp {2op, cfge ,h} vib7, vig  ,voa0
```

Example 3

```
    VGEN_MOV_OP (linp, vie) [
            voa0[b0]  =  (vib7l*vig6l[b0] + {vig6l[b1], 0x0000})[23:16]
            voa0[b1]  =  (vib7h*vig5l[b0] + {vig5l[b1], 0x0000})[23:16]
            voa0[b2]  =  (vib7l*vig6l[b0] + {vig6l[b1], 0x0000})[23:16]
            voa0[b3]  =  (vib7h*vig5l[b0] + {vig5l[b1], 0x0000})[23:16]

            voa1[b0]  =  (vib0l*vig6l[b0] + {vig6l[b1], 0x0000})[23:16]
            voa1[b1]  =  (vib0h*vig5l[b0] + {vig5l[b1], 0x0000})[23:16]
            voa1[b2]  =  (vib0l*vig6l[b0] + {vig6l[b1], 0x0000})[23:16]
            voa1[b3]  =  (vib0h*vig5l[b0] + {vig5l[b1], 0x0000})[23:16]
    ]
```

↓Translated to

```
    vmovi #0x00000000, vie0
    vmovi #0x00000000, vie1
    vmovi #0xACACACAC, vie2
    vmovi #0x00000000, vie3
    vmovi #0x0, vie4
    vmovi #0x0, vie5
    vgenlinp {2op, shf16 ,cfge ,h} vib7, vig  ,voa0
```

8.5.8. vgenmac

General Information

Single Line syntax: vozN = vozN +/- vixNp*viyMp
Lines for operation: 1

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number

Encoding

```
vozN =    vozN    +/-     vixNp  *   viyMp
          (S(i))  (P(i))  (P(i+16))
```

Example

```
VGEN_MOV_OP (mac, vih) [
        voa0    =       voa0    - via0l*vig0l
        voa1    =       voa1    + via0h*vig0h
        voa2    =       voa2    - via1l*vig1l
        voa3    =       voa3    + via1h*vig1h
        voa4    =       voa4    - via2l*vig2l
        voa5    =       voa5    + via2h*vig2h
        voa6    =       voa6    - via3l*vig3l
        voa7    =       voa7    + via3h*vig3h
]
```

↓Translated to

```
vmovi #0x76543210, vih0
vmovi #0x00000000, vih1
vmovi #0x76543210, vih2
vmovi #0x00000000, vih3
vmovi #0x55, vih4
vmovi #0x0, vih5
vgenmac {8op, cfgh} via, vig  ,voa0
```

8.5.9. vgenmac2w

General Information

Single Line syntax: vozNp = [0/vozNp] +/- (vixNp*viyMp)[35:16]
Lines for operation: 2

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number
- Destination part must start with '0l' in the first line, and advance in each line: '0h','1l', '1h', etc.

Encoding

vozNp = [0/vozNp] +/- (vixNp * viyMp)[35:16]
(A(2*i + offset)) (S(2*i + offset)) (P(2i+offset)) (P(2i+16+offset))

Example

```
VGEN_MOV_OP (mac2w, vie) [
voa0l      =      voa0l       - (vib0l*vig7h)[35:16]
voa0h      =      voa0h     + (vib0h*vig7l)[35:16]
voa1l      =      voa1l       - (vib1l*vig6h)[35:16]
voa1h      =      voa1h     + (vib1h*vig6l)[35:16]
voa2l      =      voa2l       - (vib2l*vig5h)[35:16]
voa2h      =      voa2h     + (vib2h*vig5l)[35:16]
voa3l      =      voa3l       - (vib3l*vig4h)[35:16]
voa3h      =      voa3h     + (vib3h*vig4l)[35:16]
voa4l      =      0           - (vib4l*vig3h)[35:16]
voa4h      =      0           + (vib4h*vig3l)[35:16]
voa5l      =      0           - (vib5l*vig2h)[35:16]
voa5h      =      0           + (vib5h*vig2l)[35:16]
voa6l      =      0           - (vib6l*vig1h)[35:16]
voa6h      =      0           + (vib6h*vig1l)[35:16]
voa7l       =      0          - (vib7l*vig0h)[35:16]
voa7h      =      0           + (vib7h*vig0l)[35:16]
]
```

↓Translated to

```
vmovi #0x76543210, vie0
vmovi #0xFEDCBA98, vie1
vmovi #0x89ABCDEF, vie2
vmovi #0x01234567, vie3
vmovi #0x5555, vie4
vmovi #0xFF, vie5
vgenmac2w {8op, cfge} vib, vig  ,voa0
```

8.5.10. vgenmac32w (v1)

General Information

Single Line syntax: vozNp = [0/vozNp] +/- (vcWp[b0] *vixNp)[27:8] +/- (vcWp[b1]*viyMp)[27:8]
Lines for operation: 2

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number
- Destination part must start with '0l' in the first line, and advance in each line: '0h','1l', '1h', etc.
- vc set and part must remain the same in all op's

Encoding

vozNp = [0/vozNp] +/- (vcWp[b0] *vixNp)[27:8] +/- (vcWp[b1] *viyMp)[27:8]
(A(2i+off)) (S(2i+off)) (P(2i+off)) (S(2i+16+off)) (P(2i+16+off))

Example

```
VGEN_MOV_OP (mac32w, vie) [
  voa0l  =  voa0l   - (vc0l[b0]*vig7h)[27:8]   - (vc0l[b1]*vid7h)[27:8]
  voa0h  =  voa0h   + (vc0l[b0]*vig7l)[27:8]   + (vc0l[b1]*vid7l)[27:8]

  voa1l  =  voa1l   - (vc0l[b0]*vig6h)[27:8]   - (vc0l[b1]*vid6h)[27:8]
  voa1h  =  voa1h   + (vc0l[b0]*vig6l)[27:8]   + (vc0l[b1]*vid6l)[27:8]

  voa2l  =  voa2l   - (vc0l[b0]*vig5h)[27:8]   - (vc0l[b1]*vid5h)[27:8]
  voa2h  =  voa2h   + (vc0l[b0]*vig5l)[27:8]   + (vc0l[b1]*vid5l)[27:8]

  voa3l  =  voa3l   - (vc0l[b0]*vig4h)[27:8]   - (vc0l[b1]*vid4h)[27:8]
  voa3h  =  voa3h   + (vc0l[b0]*vig4l)[27:8]   + (vc0l[b1]*vid4l)[27:8]

  voa4l  =  0       - (vc0l[b0]*vig3h)[27:8]   - (vc0l[b1]*vid3h)[27:8]
  voa4h  =  0       + (vc0l[b0]*vig3l)[27:8]   + (vc0l[b1]*vid3l)[27:8]

  voa5l  =  0       - (vc0l[b0]*vig2h)[27:8]   - (vc0l[b1]*vid2h)[27:8]
  voa5h  =  0       + (vc0l[b0]*vig2l)[27:8]   + (vc0l[b1]*vid2l)[27:8]

  voa6l  =  0       - (vc0l[b0]*vig1h)[27:8]   - (vc0l[b1]*vid1h)[27:8]
  voa6h  =  0       + (vc0l[b0]*vig1l)[27:8]   + (vc0l[b1]*vid1l)[27:8]

  voa7l  =  0       - (vc0l[b0]*vig0h)[27:8]   - (vc0l[b1]*vid0h)[27:8]
```

```
  voa7h  =    0        + (vc0l[b0]*vig0l)[27:8]    + (vc0l[b1]*vid0l)[27:8]
]
```

↓Translated to

```
vmovi #0x89ABCDEF, vie0
vmovi #0x01234567, vie1
vmovi #0x89ABCDEF, vie2
vmovi #0x01234567, vie3
vmovi #0x55555555, vie4
vmovi #0xFF, vie5
vgenmac32w {8op, cfge} vig, vid ,vc0l ,voa0vgenmac32w (v2)
```

General Information

Single Line syntax: vozNp=[0/vozNp]+/-(vixNp*viyMp[7:0])[27:8]+/-(vixKp*viyMp[15:8])[27:8]
Lines for operation: 2

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same ‘x’ for all vix registers and the same ‘y’ for all viy registers used
- Destination vector number required to match the operation number
- Destination part must start with ‘0l’ in the first line, and advance in each line: ‘0h’,’1l’, ‘1h’, etc.
- vixNp must be the same for a single operation (each 2 lines must have the same vixPn)
- vixKp must be the same for a single operation (each 2 lines must have the same vixKn)

Encoding

L1: vozNp = [0/vozNp] +/- (vixNp * viyMp[7:0])[27:8] +/- (vixKp* viyMp[15:8])[27:8]
L2: vozNp = [0/vozNp] +/- (vixNp * viyMp[15:8])[27:8] +/- (vixKp* viyMp[7:0])[27:8] (A(2i+off)) (S(2i+off)) (P(2i+off)) (P(2i+16+off)) (S(2i+16+off)) (P(2i+1+off))

Example

```
VGEN_MOV_OP (mac32w, vie) [
        voa0l    =    voa0l    - (vig3h*vid7h[7:0])[27:8]    - (vig7h*vid7h[15:8])[27:8]
        voa0h    =    voa0h    + (vig3l*vid7l[15:8])[27:8]    + (vig7l*vid7l[7:0])[27:8]

        voa1l    =    voa1l    - (vig5h*vid6h[7:0])[27:8]    - (vig6h*vid6h[15:8])[27:8]
        voa1h    =    voa1h    + (vig5l*vid6l[15:8])[27:8]    + (vig6l*vid6l[7:0])[27:8]

        voa2l    =    voa2l    - (vig5h*vid5h[7:0])[27:8]    - (vig5h*vid5h[15:8])[27:8]
        voa2h    =    voa2h    + (vig5l*vid5l[15:8])[27:8]    + (vig5l*vid5l[7:0])[27:8]
```

```
            voa3l     =   voa3l     - (vig4h*vid4h[7:0])[27:8]    - (vig4h*vid4h[15:8])[27:8]
            voa3h     =   voa3h      + (vig4l*vid4l[15:8])[27:8]    + (vig4l*vid4l[7:0])[27:8]

            voa4l      =    0        - (vig3h*vid3h[7:0])[27:8]    - (vig3h*vid3h[15:8])[27:8]
            voa4h     =   0        + (vig3l*vid3l[15:8])[27:8]    + (vig3l*vid3l[7:0])[27:8]

            voa5l     =   0        - (vig6h*vid2h[7:0])[27:8]    - (vig2h*vid2h[15:8])[27:8]
            voa5h     =   0        + (vig6l*vid2l[15:8])[27:8]    + (vig2l*vid2l[7:0])[27:8]

            voa6l     =   0        - (vig1h*vid1h[7:0])[27:8]    - (vig1h*vid1h[15:8])[27:8]
            voa6h     =   0        + (vig1l*vid1l[15:8])[27:8]    + (vig1l*vid1l[7:0])[27:8]

            voa7l     =   0        - (vig3h*vid0h[7:0])[27:8]    - (vig0h*vid0h[15:8])[27:8]
            voa7h     =   0        + (vig3l*vid0l[15:8])[27:8]    + (vig0l*vid0l[7:0])[27:8]
]
```

↓Translated to

```
vmovi #0x89ABAB67, vie0
vmovi #0x6723CD67, vie1
vmovi #0x89ABCDEF, vie2
vmovi #0x01234567, vie3
vmovi #0x55555555, vie4
vmovi #0xFF, vie5
vgenmac32w {8op, cfge} vig, vid  ,voa0
```

8.5.11. vgenmpy

General Information

Single Line syntax:
```
vozN = +/- vixNp*viyMp  : 1-8  op's
vozN =     vixNp*viyMp  : 8-16 op's
vizN  =    vixNp*viyMp  : 1-16 op's
```

Lines for operation: 1

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number
- Destination must be the same for each 8 operations - but different between 8 operation sets (advance in cyclic order)

Encoding

```
vozN =   +/-        vixNp * viyMp   : 1-8  op's
vozN =              vixNp * viyMp   : 8-16 op's
vizN  =             vixNp * viyMp   : 1-16 op's
         (S(i))     (P(i))  (P(i+16))
```

Example 1

```
VGEN_MOV_OP (mpy, vie) [
        vie0  =    vib0l*vig7h
        vie1  =    vib0h*vig7l
        vie2  =    vib1l*vig6h
        vie3  =    vib1h*vig6l
        vie4  =    vib2l*vig5h
        vie5  =    vib2h*vig5l
        vie6  =    vib3l*vig4h
        vie7  =    vib3h*vig4
        vif0  =    vib4l*vig3h
        vif1  =    vib4h*vig3l
        vif2  =    vib5l*vig2h
        vif3  =    vib5h*vig2l
        vif4  =    vib6l*vig1h
        vif5  =    vib6h*vig1l
        vif6  =    vib7l*vig0h
        vif7  =    vib7h*vig0l
]
```

↓Translated to

```
vmovi #0x76543210, vie0
vmovi #0xFEDCBA98, vie1
vmovi #0x89ABCDEF, vie2
```

```
vmovi #0x01234567, vie3
vmovi #0x0, vie4
vmovi #0x0, vie5
vgenmpy {8op, vo2 ,cfge} vib, vig  ,vie0
```

Example 2

```
VGEN_MOV_OP (mpy, vie) [
        vie0  =    vib0l*vig7h
        vie1  =    vib0h*vig7l
        vie2  =    vib1l*vig6h
        vie3  =    vib1h*vig6l
]
```

↓Translated to

```
vmovi #0x00003210, vie0
vmovi #0x00000000, vie1
vmovi #0x0000CDEF, vie2
vmovi #0x00000000, vie3
vmovi #0x0, vie4
vmovi #0x0, vie5
vgenmpy {4op, cfge} vib, vig  ,vie0
```

Example 3

```
VGEN_MOV_OP (mpy, vie) [
        voa0  =    vib0l*vig7h
        voa1  =    vib0h*vig7l
        voa2  =    vib1l*vig6h
        voa3  =    vib1h*vig6l
        voa4  =    vib2l*vig5h
        voa5  =    vib2h*vig5l
        voa6  =    vib3l*vig4h
        voa7  =    vib3h*vig4l
        vob0  =    vib4l*vig3h
        vob1  =    vib4h*vig3l
        vob2  =    vib5l*vig2h
        vob3  =    vib5h*vig2l
        vob4  =    vib6l*vig1h
        vob5  =    vib6h*vig1l
        vob6  =    vib7l*vig0h
        vob7  =    vib7h*vig0l
]
```

↓Translated to

```
vmovi #0x76543210, vie0
vmovi #0xFEDCBA98, vie1
vmovi #0x89ABCDEF, vie2
vmovi #0x01234567, vie3
vmovi #0x0, vie4
```

vmovi #0x0, vie5
vgenmpy {8op, vo2 ,cfge} vib, vig ,voa0

8.5.12. vgensmacx

General Information

Single Operation syntax:

vozNp = [0/vozNp] +/- vix/yNp*vcWl +/- vix/yNp*vcWh

vozNp = [0/vozNp] +/- vix/yNp*vcWh +/- vix/yNp*vcWlLines for operation: 2

Limitations and Additional Information

- vix and viy registers used must be from the same register set in all operations used for this instruction construction. That is, the same 'x' for all vix registers and the same 'y' for all viy registers used
- Destination vector number required to match the operation number
- Destination part must start with '0l' in the first line, and advance in each line: '0h','1l', '1h', etc.
- Operation differ between operations that use vix, and operations which use viy

Encoding

vozNp = [0/vozNp] +/- vix/yNp * vcWp +/- vix/yNp * vcWp

vix: (A(2i+off)) (S(2i+off)) (P(4i+2off)) (S(2i+8+off)) (P(4i+1+2off))

viy: (A(2i+off)) (S(2i+off)) (P(4i+16+2off)) (S(2i+8+off)) (P(4i+17+2off))

Example

```
VGEN_MOV_OP (smacx, vih) [
        voa0l   =   voa0l   - vib0l*vc0l + vib0h*vc0h
        voa0h   =   voa0h   - vib1l*vc0l + vib1h*vc0h

        voa1l   =   voa1l   - vib2l*vc0l + vib2h*vc0h
        voa1h   =   voa1h   - vib3l*vc0l + vib3h*vc0h

        voa2l   =   voa2l   - vib4l*vc0l + vib4l*vc0h
        voa2h   =   voa2h   - vib5h*vc0l + vib5h*vc0h

        voa3l   =   0       - vid6l*vc1l + vid6l*vc1h
        voa3h   =   0       - vid7h*vc1l + vid7h*vc1h
]
```

↓Translated to

```
vmovi #0x76543210, vih0
vmovi #0xFFCCBB88, vih1
```

```
vmovi #0x00000000, vih2
vmovi #0x00000000, vih3
vmovi #0xFF, vih4
vmovi #0x53F, vih5
vgensmacx {cfgh} vib, vid ,vc0 ,vc1 ,voa0
```

8.5.13. vpermute2w

General Information

Single Operation syntax:

vozNp = vixNp

operation: 2

Limitations and Additional Information

- Destination vector number need to match the operation number
- Destination part must start with ‘0l’ in the first line, and advance in each line: ‘0h’,’1l’, ‘1h’, etc.

Encoding

vozNp = vixNp

(P(2i+2off))

Example

```
VEN_MOV_OP (vpermute2w, vig) [
   voa0l = via0l
   voa0h = via0h
   voa1l = via1l
   voa1h = via1h
   voa2l = via2l
   voa2h = via2h
   voa3l = via3l
   voa3h = via3h
   voa4l = via0l
   voa4h = via0h
   voa5l = via1l
   voa5h = via1h
   voa6l = via2l
   voa6h = via2h
   voa7l = via3l
   voa7h = via3h
]
```

↓Translated to

```
vmovi #0x76543210, vig0
vmovi #0x76543210, vig1
vmovi #0x00000000, vig2
vmovi #0x00000000, vig3
```

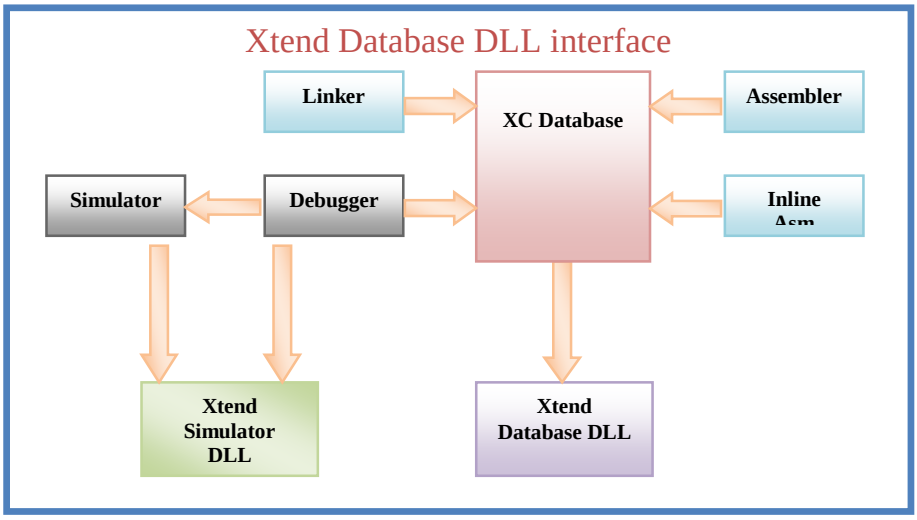
```
vmovi #0xFF, vig4
vmovi #0x53F, vig5
vpermute2w {8op, cfgg} via ,voa0
```

9. CEVA-Xtend Support

9.1 Introduction

CEVA-Xtend SDT enables the user to define new instructions which he can implement in the new CEVA-Xtend units. The user can define new switches and variables in addition to the basic encoding area that is different between instructions.
Using an automatic process, the user can define new instructions for the CEVA-Xtend units, create a Database and Simulator CEVA-Xtend DLLs, and by using them is able to:

- Use these instruction in assembly code
- View them in the Debugger disassembly window
- View and manually set the value of registers using CLI commands
- Simulate the behavior set for the new instructions in instruction set and cycle accurate modes



In order to enable automatic capability of generating the required DLLs, the following tools are provided:

- XML parser utility which generates C++ source files which have all the information regarding the CEVA-Xtend instructions
- CEVA-Xtend.bat batch file that runs XML parser utility and creates Database DLL and CEVA-Xtend Simulator DLL.

9.2 Limitations

- CEVA-Xtend unit comes instead of a regular unit in the instruction packet:
  - XH0 ←→ MS0
  - XH1 ←→ MS1
  - VXH ←→ VB
- For each VU configuration 16/32/64 there is a VU CEVA-Xtend unit is duplicated appropriately
- All CEVA-Xtend units can be used together in a single instruction packet, together with other CEVA-XC instructions
- Pipe stages and predicates are given only for sync instructions
- An immediate extension control word can be added to an CEVA-Xtend instruction
- Naming Conventions:
  - Variable and Switches names must comply with the following regular expression definition
    [_a-zA-Z] [_a-zA-Z0-9]+

- CEVA-Xtend instructions can use core resources by the following rule:
  - Instructions implemented in XH0 and XH1 units
    - Source 0, source 1, and destination
      - ***acX***
        - *a0-a7*
    - Predicate
      - ***prX***
        - *true / false*
  - Instructions implemented in VXH unit
    - Source 0
      - ***viX_vxi***
        - *via, vib, vic, vid, vie, vif, vig, vih*
    - Source 1
      - *viY_vx_up*
      - *vie, vif, vig, vih*
    - Destination
      - ***voZ0_vx***
        - *voa0, vob0*

- ***viZ0_vx***
  - *via0, vib0, vic0, vid0, vie0, vif0, vig0, vih0*

- Vector Predicate - 16 bit predication over writing to destination – per word or double word
  - ***vprX_vx***
    - *vpr0_w , vpr1_w , vpr2_w*
    - *vpr0_dw , vpr1_dw , vpr2_dw*
    - *True*

CEVA-Xtend Units:

Units must be provided by the user, in the following form:

- XH0
- XH1
- VXH

Syntax Limitations

- Switches and variables must comply with the legal variable name syntax
- A signed or unsigned immediate can be used as a variable

Encoding

- User instruction encoding is encoded first in the user-defined area
- Each variable and switch needs a specific definition of the beginning of their encoding bit, relatively to the last Hoffman bit, when counting only user-defined area bits

Immediate extension usage

- The user can use 10 or 26 bit immediate extension as part of his user-defined encoding area

9.3 Synchronous Instructions Syntax

XtendUnit.InstName [{usr_switches}] [usr_vars] [,src0] [usr_vars] [,src1] [usr_vars] [,dest] [usr_vars] [,?predicate]

Examples:

XH0.myInst1 { us0} ua0, ua2, a0, a1, a2, ?prg0
XH0.myInst2 a0, a1

9.4 Trigger (Asynchronous) Instructions Syntax

XtendUnit.InstName {[usr_switches]} [usr_vars] **[,src0]** [usr_vars] **[,src1]** [usr_vars]

Examples:
VXH.myInst1 { us0} ua0, ua2, via, vib
VXH.myInst2 vic

9.5 CEVA-Xtend Instruction Templates

Several template options are available – each defines a different usage of the CEVA-XC core resources. When defining a new CEVA-Xtend instruction a specific template needs to be assigned to it, defining its ability to read and write CEVA-XC variables, use predicates, and the stage CEVA-XC resource can be written.

For example, in the XH0 unit the second template (XH_1) is considered synchronous, so it must write to the CEVA-XC accumulator at stage E3. In addition, it has to use two CEVA-XC accumulators served as inputs, and has the option to use CEVA-XC predicate. In addition, it is possible that this instruction uses any number of user-defined CEVA-Xtend resources: switches, variables, and immediate values.

XH0/1

XH0/1	DPS (Stages)	Synchronous	Source 0 Valid	Source 1 Valid
XH_0	E2 (1 stage)	**1**	**1**	**1**
XH_1	E3 (2 stages)	**1**	**1**	**1**
XH_2	E2 (1 stage)	**1**	**1**	**0**
XH_3	E3 (2 stages)	**1**	**1**	**0**
XH_4	E2 (1 stage)	**1**	**0**	**0**
XH_5	E3 (2 stages)	**1**	**0**	**0**
XH_6	-	**0**	**1**	**1**
XH_7	-	**0**	**1**	**0**
XH_8	-	**0**	**0**	**0**

VXH

VXH	DPS (Stages)	Synchronous	Source 0 Valid	Source 1 Valid	Vector input Destination Valid	Vector Output Destination Valid
VXH_0	E4 (1 stage)	**1**	**1**	**1**	**0**	**1**
VXH_1	E5 (2 stages)	**1**	**1**	**1**	**1**	**0**
VXH_2	E4 (1 stage)	**1**	**1**	**0**	**0**	**1**
VXH_3	E5 (2 stages)	**1**	**1**	**0**	**1**	**0**
VXH_4	E4 (1 stage)	**1**	**0**	**0**	**0**	**1**
VXH_5	E5 (2 stages)	**1**	**0**	**0**	**1**	**0**
VXH_6	-	**0**	**1**	**1**	**0**	**0**
VXH_7	-	**0**	**1**	**0**	**0**	**0**
VXH_8	-	**0**	**0**	**0**	**0**	**0**

Description:

- **DPS** – CEVA-XC accumulator/vector destination pipe stage
- **Synchronous** – the instruction is coupled with the core pipeline and its execution is synchronized with it
- **Source 0 Valid** – Source accumulator or vector 0 is present in the instruction
- **Source 1 Valid** – Source accumulator or vector 1 is present in the instruction
- **Vector input Destination Valid –** There is a vector input register as a destination in the instruction
- **Vector Output Destination Valid –** There is a vector output register as a destination in the instruction

9.6 User-Defined Area

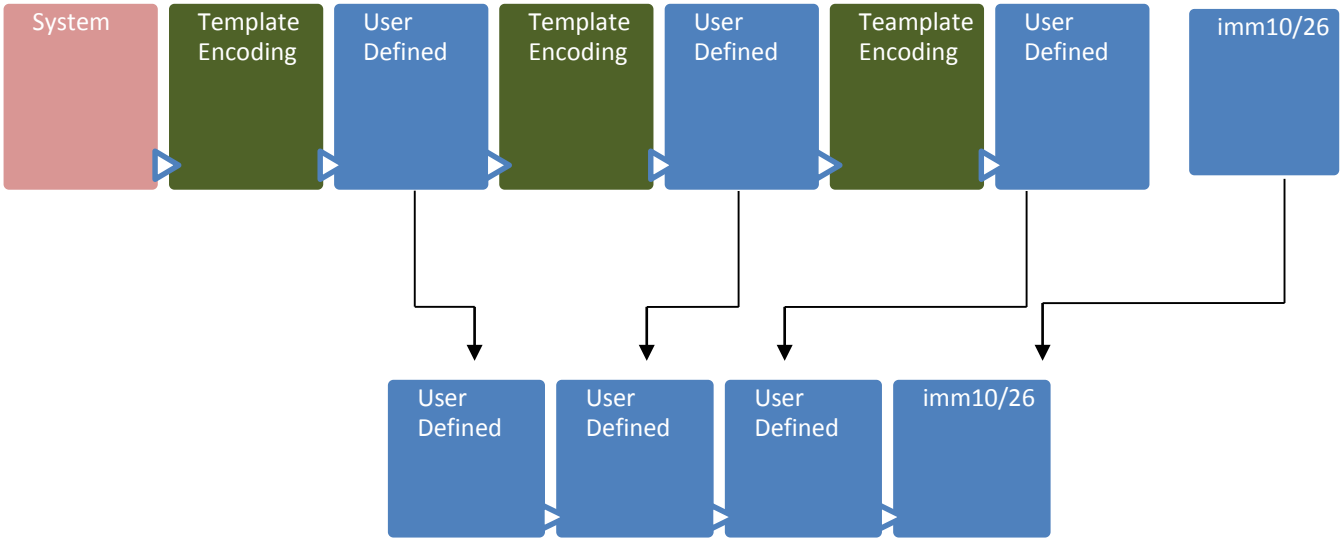


Figure 3: CEVA-Xtend user-defined Area

The user-defined area is used as a free encoding zone where the user can define the instruction encoding, new switches, new variables, and immediate values:

- **Instruction Encoding** can be used by the user to separate between different instructions he implemented in the same encoding template (the encoding templates already have different encoding patterns).
  The Huffman code needs to be encoded at the MSB of the user-defined area.
- **New switches** can be defined in the user-defined encoding area, right after the Huffman encoding of the instruction.
  New switch types need to be defined in the XML "CEVA-Xtend Types" structure, and list the switches included in this switch type together with their encoding. Switches may be defined as non-mandatory if one of the switch members is defined as default (only one switch member can be defined as default). If no default member is chosen then the switch is mandatory: the instruction syntax must include one of the switch members.
- **New variables** can be defined in the user-defined encoding area.
  New variable types need to be defined in the XML "CEVA-Xtend Types" structure, and list the variables included in the variable type together with their encoding. Variables may be defined as non-mandatory if one of the variable members is defined as default (only one variable member can be defined as default). If no default member is chosen then the variable is mandatory: the instruction syntax must include one of the variable members.
  Syntactically, the new variables are used before the mandatory variables of the specific template the instruction is implemented in.

9.7 XML File Description

The XML file needs to satisfy Database, Simulator and Debugger required information.

The XML file has 2 main parts:

1. CEVA-Xtend Types Definition
2. CEVA-Xtend Instruction Definition

Basic XML file structure

```
<CEVA-Xtend_Types>

        < Registers>
        </Registers>
         <Switches>
        </Switches>
        <Variables>
        </Variables>

</Types>

<Units>
         <Unit Name= "XH0">

                  <Instruction>
                           <Syntax                    />
                           <Simulation file           />
                  </Instruction>
                  …
        </Unit>

        <Unit Name= "XH1">
        …
        </Unit>

        <Unit Name= "VXH">
        …
        </Unit>

</Units>
```

9.8 CEVA-Xtend XML File Definition

1. **CEVA_XML:** A node marking the root of the xml tree

2. **Xtend_Types:** Holds a list of CEVA-Xtend Types

    a. **Registers:** Holds a list of CEVA-Xtend registers which can be accessed by the CEVA-Xtend units
        i. **Register:** Holds a list of the register parts, and contains the name of the CEVA-Xtend register, and its total length in bits
            1. **Register_Part:** Contains the part's name, its start bit in the register and it length in bits

    b. **Switches:** Holds a list of CEVA-Xtend Switch types
        i. **Switch:** Contains a type name, encoding length in bits, and holds a list of the member switches
            1. **Switch_member:** Contains the name of the switch, its encoding, and a flag which determines whether this member is the default member of this switch which is used if a switch member is not written in the instruction syntax

    c. **Variables:** Holds a list of CEVA-Xtend Variable types and CEVA-Xtend Immediate types
        i. **Variable:** Contains a type name, variable size in bits, encoding length in bits, and holds a list of the member variables
            1. **Variable_member:** Contains the name of the variables, and its encoding.
            The variable may be a register or a register part

3. **Units:** Holds a list of units
    a. **Unit:** Contains the unit name and holds a list of instructions included in this unit
    The unit name can be only one of the three CEVA-Xtend units available – and it must appear not more than once.
    The unit names are: XH0, XH1, VXH
        i. **Instruction:** Contains the instruction name, the Template it is based on, the instruction base Instruction encoding, the DPS value, and holds a list of the instruction attributes.
        The DPS value can be one of the following: E2, E3, E4, E5 – but they have to fit their template.
        The instruction name is unique in the unit: it must not appear more than once in the same unit.

1. **Syntax:** Contains a single string describing the order of the switches and variables.
   Immediate variables are described only here without being mentioned in the variable definition.

2. **Simulation:** Contains the file name of the C++ source file which holds the implementation of this instruction.

9.9 CEVA-Xtend Types Example

```
<Xtend_Types>

        < Registers>
                < Register     name=”uv0”   length=”20”>
                        < Register_Part      name=”uv0l”  start_bit=”0”   length=”8” />
                        < Register_Part      name=”uv0h”  start_bit=”8”   length=”8” />
                        < Register_Part      name=”uv0e”  start_bit=”16” length=”4” />
                </Register >
                < Register     name=”uv1”   length=”20”>
                        < Register_Part      name=”uv1l”  start_bit=”0”   length=”8” />
                        < Register_Part      name=”uv1h”  start_bit=”8”   length=”8” />
                        < Register_Part      name=”uv1e”  start_bit=”16” length=”4” />
                </Register >
                < Register     name=”uac0”  length=”128” />
                < Register     name=”uac1”  length=”128” />
                < Register     name=”uac2”  length=”128” />
        </Registers>

        <Switches>
                <Switch type=”usX” enc_length=”2”>
                        <Switch_member    name=”us0”   encoding =”00”      default=”yes” />
                        <Switch_member    name=”us1”   encoding=”01”       default=”no” />
                        <Switch_member    name=”us2”   encoding=”10”       default=”no” />
                </Switch>
        </Switches>

        <Variables>
                <Variable type=”uvXh” enc_length=”1”>
                        <Variable_member name=”uv0h”   encoding=”0” />
                        <Variable_member name=”uv1h”   encoding=”1” />
                </Variable>

                <Variable type=”uacX” enc_length=”2”>
                        <Variable_member name=”uac0”   encoding=”00” />
                        <Variable_member name=”uac1”   encoding=”01” />
                        <Variable_member name=”uac2”   encoding=”10” />
                </Variable>
        </Variables>

</Types>
```

9.10 Instruction Definition Example

```
<Units>

        <Unit Name= “XH0”>

                <Instruction Name=”userInst01” Template=”XH_0” Inst_Enc=”1101101”>
                        <syntax=”userInst01 {usX} acX, acY, uaXh, uimm16, uacX, acZ”>
                        <Simulation file=”userInst01.sim”/>
                </Instruction>

                <Instruction Name=”userInst02” Template=”VXH_0” Inst_Enc=”0011010”>
                …
                </Instruction>

                …

        </Unit>

        <Unit Name= “XH1”>
        …
        </Unit>

        <Unit Name= “VXH”>
        …
        </Unit>

        …

</Units>
```

9.11 CEVA-Xtend Simulation File

9.11.1. Simulation File Structure

The user defines CEVA-Xtend instructions that can be supported in the CEVA-XC simulator. In order to do so, the user should supply a unique simulation file that describes each instruction's functionality.

The simulation file should be written in C++ language, using specific statements (macros) described below. These special macros enable the user to set or get CEVA-Xtend registers values, access the XC registers, use the instruction's switches, etc.

The User's simulation file is translated into two execution functions – one for the simulator's instruction-set mode and the other for the cycle-accurate mode. In order to support both modes, the simulation file should detail the pipeline stage of each instruction (both synchronous and asynchronous). In the instruction-set mode, all stages are performed at once, while in the cycle-accurate mode each stage is performed in the proper cycle.

When receiving a value of particular source and using it on another pipeline stage, it should be saved in special structure by **M_SET_PIPELINE_VAL** macro. This value should be received by **M_GET_PIPELINE_VAL** macro when is used on another pipeline stage. The definition of these macros can be found on the next paragraph. The examples of using of this macro's can be found in examples.

The following example describes the basic structure of the simulation file:

```
M_PIPE_LINE_START
{

        M_PIPE_LINE_STAGE(M_D3)

                [User C++ Code Block]

        M_PIPE_LINE_STAGE(M_D4)

                [User C++ Code Block]

        M_PIPE_LINE_STAGE(M_E1)

                [User C++ Code Block]

        M_PIPE_LINE_STAGE(M_E2)

        [User C++ Code Block]

                …
}
```

9.11.2. Simulation File Macros

```
-------------------------------------------------------------------------
Purpose: Sets the register value in the pipeline.
Input:   index - Index of the rigister in the instruciton
         value – register value
Output:  None.
-------------------------------------------------------------------------
```

M_SET_PIPELINE_VAL(index, value)

```
-------------------------------------------------------------------------
Purpose: Get the register value from the pipeline.
Input:   index - Index of the rigister in the instruciton
Output:  Value of the reigister.
-------------------------------------------------------------------------
```

M_GET_PIPELINE_VAL(index)

```
-------------------------------------------------------------------------
Purpose: Marks the last stage of the instruction.
Input:   None.
Output:  None.
-------------------------------------------------------------------------
```

M_INST_FINISHED

```
-------------------------------------------------------------------------
Purpose: Returns ID of the register according to its index in the
         instruction Notice that the register index refers to the
         registers only, not to the total variable index which includes
         immediate values as well.
Input:   Index of the reigister in the instruction
Output:  ID of the register.
-------------------------------------------------------------------------
```

M_GET_INST_REG_ID(index)

```
-------------------------------------------------------------------------
Purpose: Returns the length of the register in bytes Notice that the
         register index refers to the registers only, not to the total
         variable index which includes immediate values as well.
Input:   ID of the register.
Output:  Length of the register in bytes ------------------------------
-------------------------------------
```

M_GET_REG_LEN_IN_BYTES(regId)

```
-------------------------------------------------------------------------
Purpose: Returns a register ID when getting the register name as input
Input:   Register name given as a string
Output:  Register ID
-------------------------------------------------------------------------
```

M_GET_REG_ID_BY_LEN(char*)

```
-------------------------------------------------------------------------
Purpose: Returns register value. Can return int, int64 or T_REG. Notice
         that the register index refers to the registers only, not to
         the total variable index which includes immediate values as
```

```
         well.
Input:   ID of the register.
Output:  Value.
-------------------------------------------------------------------------
```

M_GET_REG(regId)

```
-------------------------------------------------------------------------
Purpose: Sets the register value. Can get int, int64 or T_REG. Notice
         that the register index refers to the registers only, not to
         the total variable index which includes immediate values as
         well.
Input:   ID of the register, register value
Output:  None.
-------------------------------------------------------------------------
```

M_SET_REG(regId, val)

```
-------------------------------------------------------------------------
Purpose: Get the value of the predicate of the current instruction
Input:   None.
Output:  Value3
-------------------------------------------------------------------------
```

M_GET_PREDICATE_VAL

```
-------------------------------------------------------------------------
Purpose: Get the value of the vector predicate of the current
         instruction
Input:   None.
Output:  Value3
-------------------------------------------------------------------------
```

M_GET_V_PRDICATE

```
-------------------------------------------------------------------------
Purpose: Get the specific bit value in th vector predicate of the
         current instruction.
Input:   None.
Output:  Value.
-------------------------------------------------------------------------
```

M_GET_V_PRDICATE_VAL(bitNumber)

```
-------------------------------------------------------------------------
Purpose: Returns the instruction’s predicate value.
Input:   None.
Output:  bool – predicate’s value. If the instruction has no predicates
         it’ll return “true”.
-------------------------------------------------------------------------
```

M_GET_PREDICATE_VAL

```
-------------------------------------------------------------------------
Purpose: Get immidiate value in the current instruction. Notice that the
         immediate index refers to the immediates only, not to the
         variable index.
Input:   Index of the immidiate in the instruction.
Output:  Value.
-------------------------------------------------------------------------
```

M_GET_INST_IMM(index)

```
-------------------------------------------------------------------------
Purpose: Get unsigned immidiate value in the current instruction. Notice
         that the immediate index refers to the immediates only, not to
         the variable index.
Input:   Index of the immidiate in the instruction.
Output:  Value.
-------------------------------------------------------------------------
```

M_GET_INST_UIMM(index)

```
-------------------------------------------------------------------------
Purpose: Marks the start of the instruction stages description
Input:   None.
Output:  None.
-------------------------------------------------------------------------
```

M_PIPE_LINE_START

```
-------------------------------------------------------------------------
Purpose: Marks the start stage description
Input:   Stage name.
Output:  None.
-------------------------------------------------------------------------
```

M_PIPE_LINE_STAGE(stage)

```
-------------------------------------------------------------------------
Purpose: Marks the start of the default stage description
         This stage is executed when there is no other stage to
         execute (until M_INST_FINISHED is called).
Input:   None.
Output:  None.
-------------------------------------------------------------------------
```

M_PIPE_LINE_DEFAULT

```
-------------------------------------------------------------------------
Purpose: Marks the end of the stage description
Input:   None
Output:  None.
-------------------------------------------------------------------------
```

M_PIPE_LINE_STAGE_END

```
-------------------------------------------------------------------------
Purpose: Decides if a specific switch was used in the current
         instruction
Input:   switchStr – a string of the desierd switch.
Output:  bool – “true” if the switch is used in the current
         instruction, “false” otherwise.
-------------------------------------------------------------------------
```

M_IS_SWITCH_USED (switchStr)

9.11.3. Simulation File type

T_REG – Register value type. Its length in bits is not limited, however arithmetic manipulations over it need to be done using access to its parts. The access can be done using int or int64.

Example:

```
T_REG myReg;
…
int   var32 = myReg[i];
Int64 var64 = myReg[i];
```

9.11.4. Simulation File - Example 1

XH0.mov fX, acZ_xh

```
M_PIPE_LINE_START
{
        M_PIPE_LINE_STAGE(M_D3)
        {
                int         srcRegId        = M_GET_INST_REG_ID(0);
                T_REG  val                  = M_GET_REG(srcRegId);

                M_SET_PIPELINE_VAL(0, val);
                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_STAGE(M_D4)
        {
                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_STAGE(M_E1)
        {
                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_STAGE(M_E2)
        {
                if (M_GET_PREDICATE_VAL == false)
                {
                        M_INST_FINISHED
                        M_PIPE_LINE_STAGE_END
                }

                int         dstRegId        = M_GET_INST_REG_ID(1);
                T_REG   val                 = M_GET_PIPELINE_VAL(0);

                M_SET_REG(dstRegId, val);

                M_INST_FINISHED
                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_DEFAULT
        {
                M_PIPE_LINE_STAGE_END
        }
}
```

Description:

- Stage D3: receives the source register value and setting it in the pipeline
- Stage D4: empty stage
- Stage E1: empty stage
- Stage E2: Checking the predicate value. If it is 0 – the instruction is finished. Otherwise the destination value is set. Instruction is finished.

9.11.5. Simulation File – Example 2

XH0.add fX, fX, fX

```
M_PIPE_LINE_START
{
        M_PIPE_LINE_STAGE(M_D3)
        {
                int     srcRegId    = M_GET_INST_REG_ID(0);
                T_REG   val         = M_GET_REG(srcRegId);
                                      M_SET_PIPELINE_VAL(0, val);

                srcRegId        = M_GET_INST_REG_ID(1);
                val             = M_GET_REG(srcRegId);
                M_SET_PIPELINE_VAL(1, val);

                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_STAGE(M_D4)
        {
                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_STAGE(M_E1)
        {
                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_STAGE(M_E2)
        {
                M_PIPE_LINE_STAGE_END
        }
        M_PIPE_LINE_DEFAULT
        {
                int     dstRegId    = M_GET_INST_REG_ID(2);
                T_Reg currentValue = M_GET_REG(dstRegId);

                If (currentValue == 0x50)
                {
                        T_REG   val =  M_GET_PIPELINE_VAL(0) +
                                       M_GET_PIPELINE_VAL(1);

                        M_SET_REG(dstRegId, val);

                        M_INST_FINISHED
                }
                M_PIPE_LINE_STAGE_END
                }
        }
}
```

Description:

- Stage D3: receiving the source register values and setting them in the pipeline
- Stage D4: empty stage
- Stage E1: empty stage
- Stage E2: empty stage
- Default stage: Checking the destination value. If it is not 0x50 – do nothing. Otherwise, recives the source values from the pipeline, perform addition and set the result in the destination.
- This instruction is executed until the destination value is set to 0x50. When the destination is 0x50, the sum of the sources is written to the destination.

9.12 XML2DLL Utility

This utility converts the XML files and .sim files written by the user to C++ source and header files. In addition it generates an HTML listing file that describes the encoding of each CEVA-Xtend instruction.

9.13 Generate_Xtend

Generate_Xtend.exe runs XML parser utility and builds Database DLL and Xtend Simulator DLL using Microsoft Visual 2008 C++ development environment. As input it receives path of the XML file.

The utility requires the following:

1. Perl 5.10
2. Adding Perl\bin directory to the PATH
3. Microsoft Visual 2008 C++
4. Adding devenv (Microsoft Visual 2008 main interface) to the PATH

9.14 Microsoft Visual 2008 C++ Projects and Source Files

There are two Microsoft Visual 2008 C++ projects supplied, that can be used for debugging new CEVA-Xtend Database and new CEVA-Xtend Simulator.

The projects are:

- XtendDb – CEVA-Xtend Database
- XtendSimulator – CEVA-Xtend Simulator

There are two configurations that can be used:

- Debug – for debugging purposes
- Release – without debug information, optimized for minimal run time or minimal code size.

9.15 Listing File

The XML2DLL utility generates a listing file where it describes the CEVA-Xtend units defined by the user. It contains the following information:

- General
  - Register structure and bit fields
  - Switch types
  - Variable types
- Instruction specification
  - Instruction syntax
  - Instruction encoding

9.15.1. Listing File Example

Instruction 0 - mov

Syntax	mov fX, acZ_xh
Template	4
Unit	XH0
Length in words	2
Immediate extension type	none
Simulation file name	XML_SIM\mov_fX_acZ_xh.sim
Encoding Elements reference:	
System	= S
Instruction Specific Encoding	= E
Predicate	= P
Don't Care Bit	= X
fX	= V0
acZ_xh	= V1

Instruction encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
S	S	S	S	S	S	S	S	E(0)	E(0)	E(0)	V0_0	0	0	0	0
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	X	X	X	V1	V1	V1	V1	V0_1	X	X	X	X	X	X	X

Notes:

- Register and switch bits can be split. In this example the user-defined fX variable that is referred to as V0 in this example, is split to two bits – bit 7 and bit 20. The two parts are named V0_0 & V0_1
- The bit values of the Instruction Specific Encoding are displayed in the table in parentheses – E(Bit Value). For example, bit 23 has value 0.

9.16 General Description of using CEVA-Xtend Support

1. 'Generate_Xtend.bat <xmlFilePath>' should be run. This creates CEVA-Xtend Simulator DLL and CEVA-Xtend Database DLL. These libraries automatically are placed into CEVA-Xtend tools directory
2. XML file and Simulation files location is relative to the execution folder
3. Following generation the new DLL's process is completed, CEVA-XC tools support the new instructions.

9.17 Support in the Debugger

The debugger supports the newly created instructions. They can be displayed in the code memory window. All the standard debugger operations (such as step, setting BP etc.) can be performed by them.

9.17.1. CLI Commands

There are three CLI commands supported for the CEVA-Xtend in addition to the standard CLI commands:

- '?<register>' – getting the value of the user register
- '<register> = <RegisterValue>' – setting the value of the user register
- 'use xtend <XtendSimulatorDLLName>' – setting which CEVA-Xtend Simulator DLL to use in the debugger. Default name is "xtendSimulator.dll".

10. Programming Hints

This chapter includes the following topics:

- **Data Structures**
- **Safe Macros**
- **DIFF Equate**
- **Common Export**
- **Multiple Sections Definitions**
- **Direct Memory Addressing Support**
- **Fractional Arithmetic Support**
- **Download Support**

10.1 Data Structures

One-level deep data structure type definition macros can be created as follows:

```
.MACRO MyStruct

        Member1: DW ?
        Member2: DW ?,?
        Member3: DW ?

        .ENDM
```

The above macro can be used to define the same data structure in more than one section.

```
.DATA MySec1

MyStruct ; defines MySec1.Member1,
 ; MySec1.Member2, MySec1.Member3

.DATA MySec2
MyStruct ; defines MySec2.Member1,
 ; MySec2.Member2, MySec2.Member3
```

A section independent macro can be defined to operate on the data structure.

```
        .MACRO OperateOnMyStruct Section
       mov #Section.Member1, (r0)+
```

```
        mov #Section.Member2, (r1)-
        mov #Section.Member3, (r2)+s
        .ENDM
```

The macro can be used as follows:

```
OperateOnMyStruct MySec1
OperateOnMyStruct MySec2
```

10.1.1. Safe Macros Using .PUSHSEC and .POPSEC

The following macro moves the specified label to the specified register. If the label is unqualified, i.e. it contains no section prefix, section Sec1 is used as default. Even though the macro uses the **.USE** directive, and therefore modifies the current **USE** section, it is possible to save and restore the state of the current **USE** section of the caller using the **.PUSHSEC** and **.POPSEC** directives.

```
.MACRO DefaultCopy SrcLabel, TargetReg
 .PUSHSEC ; save USE section
 .USE Sec1 ; modify current USE section
 mov ##SrcLabel,TargetReg
 .POPSEC ; restore USE section
.ENDM
.CODE MyCodSec
Label:
 Nop
 DefaultCopy Member2, r0 ;
  ;expand to: mov ##Sec1.Member2, r0
 DefaultCopy Sec2.Member1,a0 ;
  ;expand to: mov ##Sec2.Member1, a0
 mov #Label, r1 ;
  ; expand to: mov #MyCodSec.Label,r1
```

10.2 DIFF Equate

Normally, only one label is allowed in an operand expression. The **IMMEDOFFSET** operator can be used to convert the offset of a label (with respect to the section in which it is defined) into an immediate numeric constant. Therefore, any number of labels may be used in an operand expression as long as at least all but one are converted into constants by the **IMMEDOFFSET** operator. Recall that the IMMEDOFFSET operator can only handle labels that have been previously defined within the module, i.e. the labels cannot be external or forward references.

A common use of two labels in an operand expression is to calculate the difference, i.e. relative offset, between the locations of the two labels:

```
.DATA MyDatSec
...
LblA: DW 5 DUP ?
...
LblB: DW ?

.CODE MyCodSec
...
mov #(IMMEDOFFSET MyDatSec.LblB) - (IMMEDOFFSET MyDatSec.LblA), r0
```

The code can be simplified with the following equate:

```
.EQU  DIFF2(Label2,Label1) ((IMMEDOFFSET Label2) - (IMMEDOFFSET
Label1))
```

The equate can be used as follows:

```
mov #DIFF2(MyDatSec.LblB,MyDatSec.LblA), r0
```

Normally, it would be very poor programming practice to calculate the difference between two labels that were not defined in the same section. In order to enforce this check, the DIFF equate could be modified as follows:

```
.EQU DIFF3(Sec, Label2, Label1)
  DIFF2(Sec.Label2, Sec.Label1)
```

The protected equate can be used as follows:

```
mov #DIFF3(MyDatSec,LblB,LblA), r0
```

The drawback of using the protected equate DIFF3 is that none of the labels can be **temporary**, since a temporary label cannot contain a section prefix. The unprotected equate, DIFF2, has no such drawback, since the (optional) section prefix for each of the two label arguments must be explicitly supplied. Normally, it would be expected that the difference operator would be used for data structures defined with permanent labels. **DIFF3** is preferred in more general cases. A better approach is to define a separate section for each significant piece of data or code, and use the **SIZEOF** operator for finding the size or length between 2 labels/variables.

Note: In SDT V8.3 and above it is possible to calculate label's difference directly:

```
mov     ##(label2-label1), r0
```

It is possible also for external and forward references labels.

10.3 Common Export / Import Include Files

Each assembly module should begin with a list of **include** files defining the module's **exports** and **imports**. The simplest way to organize a project is to break it down into modules (subject to Modular Programming principles). Each module should be defined as a separate **asm** file with its associated include files for example: MySec.asm and MySec.inc, where module and file base names are identical. The **.INCLUDE** file should contain a **.USE** directive followed by **.GLOBAL** directives enumerating all the public labels in the module. The **.asm** file should start by including common macros, followed by including module's **export** file, followed by including all of the module's **import** files, for example:

;FILE: Sec1.asm

; **common** project **macros**

.INCLUDE "PROJECT.MAC"

; module's **exports**

.INCLUDE "Sec1.inc"

; module's **imports**

.INCLUDE "Sec2.inc"

.INCLUDE "Sec3.inc"

; **local** equates and **macros** used by this module

; ...

; module's **body**

.CODE Sec1

; ...

;FILE: Sec2.asm

; **common** project **macros**

.INCLUDE "PROJECT.MAC"

; module's **exports**

.INCLUDE "Sec2.inc"

; module's **imports**

.INCLUDE "Sec3.inc"

.INCLUDE "Sec4.inc"

; **local equates** and **macros** used by this module

; ...

; module's **body**

.CODE Sec2

; ...

;FILE: Sec1.inc

; **equates** and **macros** exported by this module

; ...

; **labels** exported by this module

.USE Sec1

.GLOBAL Label1, Label2, Label3

;FILE: Sec2.inc

; **equates** and **macros** exported by this module

; ...

; **labels** exported by this module

```
.USE Sec2

.GLOBAL Label1, Label2, Label3
```

In order to have a common header file for both **exports** and **imports**, when using the .EXTERN and .PUBLIC directives substituting for the .GLOBAL directive, the following tip could be used:

```
;FILE: Sec1.asm
;
; common project macros
.INCLUDE "PROJECT.MAC"
;
; module's exports
.EQU DECLARE .PUBLIC
.INCLUDE "Sec1.INC"
.PURGE DECLARE
;
; module imports
.EQU DECLARE .EXTERN
.INCLUDE "Sec2.INC"
.INCLUDE "Sec3.INC"
;
; local equates and macros used by this module
; ...
; module's body
.CODE Sec1
```

```
; ...
```

The **.inc** files in this case, must use a DECLARE statement for each global variable, for example:

```
;FILE: Sec1.inc

; equates and macros exported by this module

; ...

; labels exported by this module

.USE Sec1

DECLARE Label1, Label2, Label3


;FILE: Sec2.inc

; equates and macros exported by this module

; ...

; labels exported by this module

.USE Sec2

DECLARE Label1, Label2, Label3
```

10.4 Multiple Section Definitions

It is possible to define a **section** in multiple parts in a **single** module as well as defining a **section** in **multiple** modules. In the first case the section's location counter continues from the last section defined regardless of whether the last section is different or other sections have been defined in the interim. In the second case, the Linker takes care of chaining all the sections defined by the same name into a single section. The order in this case is dictated by the order of the object files in the Linker script file.

Example:

```
.CODE Sec1 ; 1st instance of this section - location
  ;counter is 0
...

.DATA Sec2
...

.CODE Sec1 ; location counter automatically
  ; continues from where it
  ; ended in the previous instance of Sec1

...
```

Location counter is initialized to zero the first time a **section** is defined within a module. If a **section** is defined in more than one module, the Linker detects this and sequentially chains the sections so that they do not overlay each other.

Note: This is done in contrast to the Linker's older versions' algorithm. No **.ORG** directive is needed to prevent **implicit** overlays.
Using the **.ORG** directive results in an addressing **"hole"** between the last instance of this section name, and the current one. Therefore, when an **.ORG** directive is used to skip some addresses, a real hole is created and the **skipped** area cannot be filled by other sections. This is also different from other Linker's older versions (V7.2 and earlier), where the holes were filled with zeroes by using **.ORG** directive.

10.5 Direct Memory Addressing Support

When using **Direct** memory addressing mode in CEVA DSP**s**, a 16-bit address is generated by two sources. The instruction's opcode supplies the LSB (lower) 8 bits of the address, while the upper 8 bits of the address are supplied by the **page** bits located in a status register **st1**. Since it is very inconvenient to use absolute values (for the lower 8 bits of data addresses) inside an assembly program, one uses symbols defined in data sections. This way, when new symbols are added, deleted or moved, the Assembler and the Linker take care of generating the correct lower 8 bits of the address. Each new data section starts a new series of consecutive symbols, starting from temporary address 0 upwards. Each such data section at link time can be located anywhere in the data space and the symbols corresponding to these sections will be relocated accordingly.

The Assembler has an automatic modulo 256 operator for cutting the 8 lower bits of a 16 bit address (by the Linker) for purposes of Direct memory addressing using the link time **@** operator. In addition, the link time **OFFSET** operator can be used, provided that all data sections linked ought to be on **CEVA DSPs** page boundaries (i.e. 0x0000, 0x0100, 0x0200, etc.) and are not longer than 256 words. The **OFFSET** operator tells the Linker to put in the opcode of the instruction, the value corresponding to the offset of the symbol from the beginning of its data section; i.e., it subtracts the absolute (final) 16 bit address of the beginning of the section in which the symbol is defined, from the absolute (final) 16 bit address of the symbol.

As an **example**, suppose a program has the following code:

```
.DATA SecA
VarA: DW ?
VarB: DW ?
VarC: DW ?
 .CODE SecB
lpg # PG ( SecA.VarC )
mov OFFSET SecA.VarC, r1
```

The Linker is instructed to locate section SecA at 0x0100 (for example: by using the **at** directive). The Assembler assigns a temporary address 0 to SecA and SecA.VarA, a temporary address 1 to SecA.VarB and temporary address 2 to SecA.VarC.
Next, the Linker assigns the final address of 0x100, 0x100, 0x101 and 0x102 to SecA, SecA.VarA, SecA.VarB and SecA.VarC respectively. In the code section SecB, the Linker updates the opcode for the ***lpg*** instruction by calculating the value of shifting SecA.VarC address (i.e. 0x102) by 8 bits to the right. This results in putting the value 1 as the immediate operand of the first instruction.
For the second instruction, the Linker subtracts 0x100 (SecA) from 0x102 (the final value of SecA.VarC) so that the first operand of the ***mov*** instruction, (the Direct memory address offset) will be 2.

What happens when multiple data sections are involved? If data sections are not aligned on page boundaries, then the **OFFSET** operator will produce incorrect direct memory addresses. As an example, suppose that the same program is used as previously, but that the Linker locates SecA at 0x205 (not aligned on a page boundary) as a result of another section occupying memory space up to address 0x204. In this case, the Assembler will produce the same output, but the Linker will give the final (absolute) addresses of 0x205, 0x205, 0x206 and 0x207 to SecA, SecA.VarA, SecA.VarB and SecA.VarC respectively. In addition, in reference to SecB in the **code** section, the Linker will substitute value 2 for the immediate operand of the ***lpg*** instruction (0x205 >> 8 gives 2), and for the first operand of the ***mov*** instruction; it will **still** produce the value of 2, because SecA.VarC - SecA = 0x207 - 0x205 = 2. This however is not the correct Direct memory address offset, needed to access SecA.VarC which has address 0x207, i.e. has Direct memory offset of 7 in page 2.

The **@** operator, on the other hand overcomes this problem, since it tells the Linker to perform a modulo 256 operation on the absolute address, instead of the subtraction operation.

For example:

```
.DATA SecA
VarA: DW ?
VarB: DW ?
VarC: DW ?

.CODE SecB
lpg # PG ( SecA.VarC )
mov @ SecA.VarC, r1
```

Whether the Linker is instructed to locate the section SecA at address 0x100 or 0x205, the object of the second instruction will be correctly **relocated** since 0x102 modulo 256 is 2 and 0x207 modulo 256 is 7. It is, therefore, better practice to use the @ operator for all Direct memory addressing operands, and keep the **OFFSET** operator for use with data structures.

10.6 Fractional Arithmetic Support

Fractional arithmetic can be performed by a fixed point DSP (**CEVA DSPs** family) by allocating part of the 16 bits of a variable or memory location for the **sign**, the **integer** part and the **fractional** part. For example, let us define the **Qn** binary fractional notation, where n bits are allocated for the fraction and 15-n bits are kept for the integer part of the number. The MS bit is used for the sign bit. In binary fractional numbers, each bit to the left of the floating point has the usual weight of 2n, while the bits to the right of the floating point have a weight of 2-(n+1).

As an example, consider the 16 bit number 0x3500 in Q12 notation.

Bit number:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit value:

0	0	1	1.	0	1	0	1	0	0	0	0	0	0	0	0

The floating point value of this binary fractional number is: $2^1 + 2^0 + 2^{-2} + 2^{-4} = 3.3125$

The largest positive fraction in **Q15** notation is very close to 1.0, represented by 0x7fff, while 0.5 is represented by 0x4000. In **Q14** notation, on the other hand, 14 bits are used to represent the fractional part of the number, leaving 1 bit for the integer part. The largest positive number in **Q14** notation is therefore 1.99999 (0x7fff). Note that **Q15** fractions are more accurately represented than **Q14** fractions, however numbers larger than 1.0 cannot be represented by **Q15** fractions. When multiplying binary fractional numbers, one needs to align the floating point in the result just as in decimal floating point arithmetic. This is accomplished by shifting the product one bit to the left and taking the high part of the shifted **P** register. For example, assume one needs to calculate 0.5 * 0.5 = 0.25. Using **Q15** notations, 0x4000 * 0x4000 = 0x10000000. The 32 bit result has **Q30** notation (15+15 bits to the right of the floating point). By shifting the product 1 bit to the left, the high part of the product becomes 0x2000 which is 0.25 in Q15 notation. Shifting the product 1 bit to the left, corresponds to deleting 1 sign bit from the Q30

product that contains 2 sign bits (one from each multiplicand). As another example, suppose one multiplies a **Q15** number by a **Q12** number. The product will be a **Q27** number, i.e. it has 27 bits representing the fractional part and 3 bits for the integer part.

The Assembler built-in operator **FRACT** automatically converts floating point numbers to 16 bit binary fractional numbers with a variable, user specified number of bits for the fractional part of the number.

Examples:

```
mov ##FRACT (0.5, 15), x ; translates to mov ##0x4000, x

mov ##FRACT (0.015625, 15), r0 ; translates to mov ##0x200, r0

mov ##FRACT (1.25, 14), y ; translates to mov ##0x5000, y
```

For convenience purposes, one can define a **macro** to simplify the notation as follows:

```
.EQU Q15(num) FRACT (num, 15 )

.EQU Q14(num) FRACT (num, 14 )
```

so that the previous examples can be rewritten as:

```
mov ##Q15(0.5), x

mov ##Q15(0.015625), r0

mov ##Q14(1.25), y
```

with the same end result.

Note that the **CEVA DSPs** architectures have a built-in shifter for the product register, that is specifically convenient for Q15 * Q15 operations. If all numbers are assumed to be in Q15 notation, the programmer can set the shift mode of the P register to be 1 bit to the left (SP=2 in ST1) and all results will be correctly aligned. By moving the product register to the accumulator and using combinations of the *shr*, *shl*, *shr4* and *shl4* instructions, it is possible to use all fractional notations to obtain fractional arithmetic with variable accuracy.

10.7 Download Support

This topic describes how downloadable applications can be created. The use of **SIZEOF** and **INCODE** Assembler operators are demonstrated, as well as the Linker operator { } and the Linker **noload** attribute.

Suppose you want to alternate execution between two small programs in a fast (but small and expensive) program RAM, and that you store these two programs in the data space, say in slow (cheap) EPROM that need to be downloaded. The following assembly file could do this job of downloading and executing the different programs:

```
        ; file: downld.asm
        .CODE Downld
        ;--- download first program:
Start:
        mov ## First.Start, r1
        mov ## INCODE First.Start, r4
        rep # SIZEOF ( First )-1
        movd(r1)+,(r4)+
        ;--- execute first program:
        call First.Start
        ;
        ;--- download second program:
        ;
        mov## Second.Start, r1
        mov## INCODE Second.Start, r4
        rep# SIZEOF ( Second )-1
        movd(r1)+, (r4)+
```

```
        ;--- execute second program:
        call Second.Start
        brr Downld.Start
        ;--- the first downloaded program:
        .CODE First
Start:
        mov r0, r1
        ret
        ;--- the second downloaded program:
        .CODE Second
Start:
        clr a0
        ret
```

Note how the **SIZEOF** operator is used to download the complete First or Second section. In addition, it should be noted that the **INCODE** operator must be used in order to indicate the Linker to use the addresses of the **First.Start** and **Second.Start** labels from the program space (and not from their addresses in the data space). This is done for **r4** pointer register to be loaded with the destination address of the ***movd*** instruction. The **INCODE** operator must not be used when the **r1** pointer register is loaded, because by default, all **mov** and **ALU** instructions, when using a memory resident operand, refer to the data space. Finally, note that when the down-loaded programs are executed, by the ***call*** instruction, no **INCODE** operator is used, because by default all ***br(r)*** and ***call(r)*** instructions refer to labels in the code space.

Now, assume that the program RAM is located at address 0x8000 (in the code space) and that the EPROM is located at address 0x4000 (in the data space). Also assume that some

main.asm includes start-up code that executes the Dowlnd code above. The Linker script file could look like:

```
objects:
        main.o
        downld.o
code:
        Main
        Downld at 0x100
        {
        First at 0x8000 noload
        Second at First noload
        }
data:
        First at x4000
        Second
```

Alternatively, one can make use of **classes** to make sure the program sections (First and Second) fit the physical program RAM (PRAM) and data EPROM devices:

```
objects:
        main.o
        downld.o
classes:
        PRAM [c:8000,c:81ff]
        EPROM [d:4000,d:4fff]
code:
```

```
        Main
        Downld at 0x100
PRAM:
        {
                First noload
                Second at First noload
        }
EPROM:
        First
        Second
```

The end result will be the same for both Linker script files, except that in the second case the Linker will warn if, for example, either First or Second do not fit into the program RAM or if both First and Second do not fit together into the EPROM.

Note that the **noload** attribute provides the assurance the loader of the Debugger will not load these sections at load time.

11. Appendix

11.1 Internal Directives used by MPP & the Compilers

The following directives are used internally by the macro preprocessor MPP and the C/C++ Compiler, to pass important information for listing and debugging purposes. These directives are usually not used by the assembly programmer and are mentioned here so that the user shall be aware of certain reserved words that should not be used for symbol names (labels or variables) in any of his programs.

.ALIGN number [,value]

The .ALIGN directive is used by the C/++ Compiler to align certain C variables on certain addresses. The first parameter determines the number of data words that need to be skipped, by setting the next available address to be the closest multiple factor of the given number. The optional second parameter determines how to fill the skipped memory locations (defaults to 0).

.ASCII "string" [, "string" ...]

The .ASCII directive is used to allocate and initialize a memory buffer for the string parameter(s). Note that each character is allocated a 16-bit word in memory. The string is not ended with a NULL character.

.ASCIIZ "string" [, "string" ...]

The .ASCIIZ directive is used to allocate and initialize a memory buffer for a NULL terminated string. Note that each character is allocated a 16-bit word in memory.

.

BB

The **.BB** directive is used to describe debugging information - this directive indicates the beginning of a C block of statements marked within {}. It is given after .DEF and before .ENDEF. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.BF

The **.BF** directive is used to describe debugging information - this directive indicates the beginning of a C function. It is given after **.DEF** and before **.ENDEF**. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.BSS

The **.BSS** directive is used to define a data section that includes all uninitialized global / static variables.

Notes:

1. By default, this section is loaded with zeros but is NOT reinitialized in the startup code (CRT0.C).

2. . By default (for compatibility sake) the .bss section is bound to the .data section using meta section attribute (i.e. .data$.bss).

.BYTE number [, number ...]

The **.BYTE** directive is used to allocate and initialize a memory buffer for a list of bytes. Two bytes are concatenated and put in each 16 bit word of memory. If an odd number of bytes are specified, the last one is zero-padded from the left side.

.COMM symbol, length

The **.COMM** directive is used to declare and allocate data memory variables in a common **.BSS** section of initialized data. This directive is used by the C/C++ Compiler for the Assembler and the Linker.

.data

The **.data** directive is used by the C/C++ Compiler for all initialized global / static variables (uninitalized variables goes to '.bss', see next). The **.data** IS reinitialized in the startup code (CRT0.C) in order to ensure correct values after reset as well. This directive is conceptually equivalent to "**.DSECT** .data". That is, **.DSECT** section type, with the section name .data.

.DEF symbol

The **.DEF** directive is used to start the description of debugging information for

the specified function. The information is ended by the **.ENDEF** directive. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.DIM number [, number ...]
The **.DIM** directive is used to describe debugging information - this directive is used to pass on dimension information of a C array. Up to 4 dimensions are supported. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.DOUBLE number [, number ...]
The **.DOUBLE** directive is equivalent to the **.FLOAT** directive, and is used to allocate and initialize a memory buffer for a list of single precision floating point numbers using the IEEE 754 single precision floating point format. Note that each double (or float) number is represented by two words (specific to **CEVA DSPs**) of memory.

.EB
The **.EB** directive is used to describe debugging information – It indicates the ending of a C block of statements marked within {}. It is given after .DEF and before .ENDEF. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.EF
The **.EF** directive is used to describe debugging information - this directive indicates the ending of a C function. It is given after **.DEF** and before **.ENDEF**. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.ENDEF
The **.ENDEF** directive is used to end the description of debugging information for the function specified in the last **.DEF** directive. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.ERROR “FreeText”
The **.ERROR** directive is the equivalent of the C "#error" directive. It displays the free text (the given argument) as a message to "stdout" and exits. The preprocessor uses this directive and if necessary, can be used by the programmer.

.FILE "filename"
The **.FILE** directive is equivalent to the C "#file" directive. It is used to pass the name of the current source file for debugging purposes. This information is passed by the C/C++ Compiler to the Assembler and Linker for use by the Debugger, when the user wants to debug his programs at source level.

.FLOAT number [, number ...]
The **.FLOAT** directive is used to allocate and initialize a memory buffer for a list of single precision floating point numbers using the IEEE 754 single precision floating point format. Note that each float (or double) number is represented by two words (specific **CEVA DSPs**) of memory.

.no_init
The **.no_init** directive is used for data section that includes uninitialized global / static variables that are explicitly placed (in this section) by the '__ attribute__(section('.no_init'))'. The objective of this section is to guarantee that certain variables that are initialized by the Debugger will not be overwritten by any other initialization (locating these variables in .bss section is not an option since optionally .bss can be zeroed in the startup code - CRT0.C). **Note** that by default (for compatibility sake) the .no_init section is bound to the .data section using meta section attribute (i.e. .data$.no_init).

.INT number [, number ...]
The **.INT** directive is used to allocate and initialize a memory buffer for a list of short integers. Note that each integer (or word) number is represented by 16 bits (1 word) of memory.

.LINE number ["filename"]
The **.LINE** directive is used to control the line number and/or name of the current source file for the purpose of reporting errors. This directive is inserted by the preprocessor for the Assembler and is not for use by the programmer.

.LONG number [, number ...]
The **.LONG** directive is used to allocate and initialize a memory buffer for a list of long integers. Note that each long integer number is represented by two words (specific to the **CEVA DSPs**) of memory.

.NDATA
Same as .data

.SCL number
The **.SCL** directive is used to describe debugging information - this directive indicates the storage class of a C variable. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.SIZE number
The **.SIZE** directive is used to describe debugging information - it describes the size of a C variable. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.SPACE number [, value]
The **.SPACE** directive is used to skip a number of data memory words (size is **CEVA DSPs** specific). Optionally, the skipped memory words may be filled with the supplied value. This directive is always used in data sections.

.STRING "string"
The .STRING directive is used to declare a string of characters (no null termination is added).

.STRINGZ "string"
The .STRINGZ directive is used to declare a string of characters. The Assembler will add a null termination character to the input string.

.TAG symbol
The **.TAG** directive is used to describe debugging information - it gives a reference to a user-defined C data type (structure). This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.TEXT
The **.TEXT** directive is used to start a new code section. This directive is issued by the C/C++ Compiler for the Assembler and Linker. This directive is conceptually equivalent to "**.CSECT** .text". That is, the **.CSECT** section type, with the section name .text

.TYPE number
The **.TYPE** directive is used to describe debugging information regarding the type of a variable or a function. This directive is issued by the C/C++ Compiler when the -g option is used, to allow symbolic debugging using the Debugger.

.VAL symbol | number | .

The **.VAL** directive is used to describe debugging information regarding the value of a variable. This directive is issued by the C/C++ Compiler when the **-g** option is used, to allow symbolic debugging using the Debugger.

.WARNING Free text

.The **WARNING** directive displays the free text (the given argument) as a message. This directive is used by the preprocessor and can be used by the programmer as well.

.WORD number [, number ...]

The **.WORD** directive is used to allocate and initialize a memory buffer for a list of integer words. Note that each word (or int) number is represented by word (size is **CEVA DSPs** specific) of memory.

.X

The **.X** directive is used to signify each line but the first of a multiple line macro expansion in order to keep the source line counter synchronized. This directive is inserted by the preprocessor for the Assembler and is not for use by the programmer.

11.2 Glossary

- **Application Profiler** – Can be invoked 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 Includes:

  - 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 DSP Cores 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. (Not relevant for Toolbox version, using pdf format guides).

- **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 DSP Core 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 DSP Core 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 software development.

- **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 (Assembly) 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-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 in case of 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. (Not relevant for Toolbox version, using pdf format guides).

- **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 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 CEVA-Toolbox Debugger). Enables the programmer to create sophisticated and complex Debugger's scripts that may contain conditionals, loops, and file I/O.

12. Index