cādence®

Xtensa® System Software Reference Manual

Basic Runtime (XTOS) and HAL Reference Manual

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date:10/2018

RI-2018.0

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Tables

List of Figures

Preface

This document is written for Xtensa processor customers who are experienced in the programming and debugging of software in the UNIX or Windows operating-system environment.

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- `$` represents your shell prompt, in user-session examples.
- **literal_input** indicates literal command-line input.
- `variable` indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *... output ...* indicates unspecified program output.
- *[optional-*`variable`*]* indicates an optional parameter.
- **[**`variable`**]** indicates a parameter within literal square-braces.
- **{**`variable`**}** indicates a parameter within literal curly-braces.
- **(**`variable`**)** indicates a parameter within literal parentheses.
- | means *OR*.
- *(var1* | *var2)* indicates a required choice between one of multiple parameters.
- *[var1* | *var2]* indicates an optional choice between one of multiple parameters.
- `var1` *[,* `varn`*]** indicates a list of 1 or more parameters (0 or more repetitions).

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *flush* is deprecated due to potential ambiguity (it may mean *write-back* or *discard)*.
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- word means 4 bytes.

Changes from the Previous Version

1. Introduction to System Software on Xtensa Processors

The Xtensa Xplorer design environment and Xtensa Processor Generator (XPG) allow the designer to customize and extend the Xtensa processor core to best meet their computation needs. This extensibility can result in dramatically higher performance and lower power and area requirements, while preserving benefits of programmability. The core of this design process entails the writing and running of software.

Running any application beyond the most trivial typically involves the use of basic software services — such as exception handling, interrupt dispatching, startup — typically referred to as *runtime* or *system software*. Such software can, of course, be found in various operating systems, commercial and open source, targeting the Xtensa architecture; some of these are described in the *Xtensa OSKit Guide*. Alternatively, you can use the Xtensa XOS embedded kernel provided with the tools, or choose to create your own. As a starting point for writing programs on Xtensa processors, Cadence provides a basic runtime, XTOS, that supports single threaded applications and works seamlessly with Xtensa Tools.

System software, in a larger sense, also includes other software that is close to the processor in some way. Such is the case, at least in some respects, of the software components described in this document. Essentially, the *Xtensa System Software Reference Manual* describes those portions of system software that are provided by Cadence as part of a configured Xtensa processor's download package. They include (in both object and source form):

- A basic runtime (XTOS) that includes basic vector handlers and a C runtime, and is pulled into an application's executable image by most of the linker support packages (LSPs). In combination with other libraries (a C library, basic character I/O, *etc*.), it lets designers bring up and run single threaded applications on the Xtensa emulation boards and on the simulator.
- The Xtensa "HAL", a header file and API that describe an Xtensa processor's configuration options, and those aspects of its extensions (TIE) that are relevant to runtime code such as operating systems. The API portion (HAL library) includes functions for managing caches, register windows, and other functions whose operation varies among configured processors.
- The Xtensa Multiprocessor Environment (XMP), a library and methodology that takes special advantage of the Xtensa processors' multiprocessing capabilities to manage such things as shared and private data, data caches, program startup, and barrier and spinlock synchronization.
- The Xtensa XOS embedded kernel, which is not described here. Refer to the *Xtensa XOS Reference Manual* for more information.

Beyond the scope of this document, Cadence also provides source code and documentation to aid the developer who is producing system software. This source code is intended to provide both an example and a starting point for the developer who is porting a proprietary operating system, who is developing a new operating system for the Xtensa architecture, or who is porting a runtime to work on a new board. For more documentation on this topic, refer to the *Xtensa Microprocessor Programmer's Guide*. Basic processor architecture reference information is also available in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

1.1 Xtensa Tools Installation Directories

The guide and conventions for directories used to install Xtensa tools, as well as installation instructions, are described in detail in the *Xtensa Development Tools Installation Guide*. References are made throughout this document to pathnames under these installed directories, so conventions for referring to them are briefly repeated here.

The Xtensa development system consists of two components, which are installed separately on your system:

- Xtensa Tools, which are independent of a specific processor configuration.
  We refer to the directory containing Xtensa Tools as `<xtensa_tools_root>`.
- The core package(s), consisting of configuration information, software and possibly RTL for a specific processor configuration. This tree is downloaded for each configuration you build. We refer to the directory containing these processor-specific files as `<xtensa_root>.`

2. Basic Runtime and Handlers (XTOS)

With Xtensa processors, Cadence provides a single-threaded *runtime* (run-time execution environment) that offers designers a quick start on developing software for Xtensa processors. This runtime, sometimes referred to as XTOS, is a simple, single-threaded interrupt and vector management package that provides startup code, handlers for all the vectors, and a simple C interface to interrupts and exceptions (some of the routines to manipulate interrupts and exceptions are provided by the HAL, see Chapter 3). These handlers are not intended for use as a production system. They serve as a simple bootstrapping and/or example system for the Xtensa software developer.

Despite the name, XTOS is not an operating system. It is a simple runtime. Cadence also provides a multi-threaded kernel, XOS. Note that XOS is not a full operating system. Refer to the *Xtensa XOS Reference Manual* for details.

Note: Refer to Appendix A for a list of deprecated, renamed, and dropped features.

2.1 Relationship with Linker Support Packages (LSP)

This simple runtime is pulled in by various linker support packages (LSPs) along with other object files and libraries according to the specific requirements of each LSP. These object files and libraries are normally found under one of these two directories, and standard LSP directories are under the first of these two:

```
<xtensa_root>/xtensa-elf/lib/
<xtensa_root>/xtensa-elf/arch/lib/
```

Some of the items typically pulled in by Cadence-provided LSPs include:

- Anchors(`_vectors.o`), which reference vectors to pull them into the application.
- Reset vector (`libhandlers-*.a`), which initializes the processor state, unpacks RAM based vectors and sections as defined by the LSP, and jumps to `_start`.
- Exception and interrupt vectors, handlers, and runtime code (`libhandlers-*.a`), which includes the utility vector and interrupt management routines (XTOS).
- C run-time (`crt*.o`), which starts at `_start`, sets up the environment and initialization expected by C code, calls `main`, and handles exit from `main` appropriately. See Section 2.4 on page 17.
- C library (`libc.a`), from either *newlib*, or *xclib*, alternate implementations of the C standard library, often used in embedded systems. The C library expects a number of functions that are usually implemented in a system call or equivalent library (such as in the next item). See the *Red Hat newlib C Library Reference Manual* for details on `newlib`, and the *Xtensa C Library User's Guide* for details on `xclib`.

- System call library (`libgloss.a`, `libsim.a`, or `libgdbio.a`) is the OS-specific “system call” library layer (*e.g.*, `_open_r`, `_close_r`, etc.) as described in the *System Calls* section of the *Red Hat newlib C Library Reference Manual*. `libgloss` is used on board-specific LSPs and requires board-specific libraries to implement character I/O and other facilities. `libsim` is used for code running in the Xtensa Instruction Set Simulator (ISS) and uses the semi-hosted `SIMCALL` instruction interface to implement a minimal set of system calls. `libgdbio` uses a connected GDB debugger for access to host-side console and file I/O; it is used by the `gdbio` LSP.
- Board-specific libraries (`libminrt.a`, `libtinyrt.a`, `libxtav60.a`, `libxtav110.a`, and `libxtav200.a`), which provide board-specific functionality. In particular the libraries provide the `inbyte()` and `outbyte()` functions required by `libgloss.a`, as well as any board-specific functions.
- Compiler-specific libraries (`libgcc.a`), which implement functions that may be called by compiler-generated code. The `specs` file does not mention this file explicitly; the compiler driver (`xt-xcc`) links it automatically unless given `-nostdlib`.
- HAL (`libhal.a`), the processor configuration description and abstraction layer. See Chapter 3.
- Other libraries, such as the math and C++ libraries, are also included.

Certain LSPs combine the above to provide a complete single-threaded C runtime. This runtime is not for use in production systems without modification and validation.

Each LSP also includes linker scripts that map exception vectors, code, and data to their appropriate locations in the processor’s address space. Refer to the *Xtensa Linker Support Packages (LSPs) Reference Manual* for more details on LSPs.

2.2 Public Interfaces

The basic runtime (XTOS) is distributed as both source and object. It is linked in automatically with various LSPs, or can be expressly provided to the linker with LSPs that do not include a runtime (such as `nort`).

The handlers provide three sets of public interfaces. These are interrupt management, exception management and timer management. They are defined in a C header file that can be included as follows:

```
#include <xtensa/xtruntime.h>
```

Note: On processors with the relocatable vector option, XTOS does not provide any special interface for relocating vectors. The location of vectors is most easily controlled using a custom LSP, as described in the *Xtensa Linker Support Packages (LSPs) Reference Manual*. The XTOS reset vector picks up the selected vector base from a symbol defined by the custom LSP’s linker script.

2.2.1 Differences Between XEA2 and XEA3 Interrupt Architectures

While XTOS supports both Xtensa Exception Architecture version 3 (XEA3) and Xtensa Exception Architecture version 2 (XEA2), there are significant differences in interrupt properties between the two. Refer to the section on *Options for Interrupts and Exceptions* in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more information. Even though the same set of XTOS functions are provided for both architectures, there are some architecture differences. These differences are summarized in Table 2–1.

Table 2–1. Interrupt Differences Between XEA2 and XEA3

XEA2	XEA3
XEA2 supports a maximum of 32 interrupt inputs.	XEA3 supports a maximum of 256 interrupt inputs.
Interrupts are controlled by the INTERRUPT and INTENABLE registers.	Interrupts are controlled and managed by a separate interrupt controller.
XEA2 defines three types of interrupt priorities: level-one interrupts (also known as low-priority interrupts), medium-priority interrupts, and high-priority interrupts.	XEA3 defines interrupt priority levels but does not group interrupts into low/medium/high based on priority level.
Interrupt priority levels are fixed and cannot be changed.	If the configuration allows, interrupt priority levels can be changed by software.
Interrupt sensitivity (edge vs. level) is fixed and cannot be changed.	Interrupt sensitivity (edge vs. level) can be configured per interrupt for external interrupt inputs. Software, timer and internal interrupts are edge-triggered.
Only one NMI interrupt input is allowed. This is fixed in hardware and cannot be changed.	Multiple interrupts can be programmed as NMI. Any interrupt can be changed at runtime to NMI.
Individual interrupts can be enabled / disabled by setting / clearing the corresponding bit in the INTENABLE register.	Individual interrupts can be enabled / disabled by setting / clearing the enable bit in the interrupt controller.
Interrupts up to a specific priority level can be masked by writing the INTLEVEL field of the PS register.	Interrupts up to a specific priority level can be masked by writing a register in the interrupt controller. All interrupts (except NMI) can be masked by setting the DI bit of the PS register.
Handling high-priority interrupts in C requires more complex (and slower) save and restore sequences. XTOS cannot dispatch NMI to a C handler.	All interrupts can be efficiently handled in C, including NMI interrupts.

XEA3 allows NMI interrupts to be programmable. Multiple interrupts (including software and timer interrupts) can be configured to become NMI by setting their priority to the highest available priority level. These interrupts then become non-maskable but are still dispatched to handlers in the same manner. NMI properties are locked in by setting the `NMILock` bit in the Xtensa interrupt controller. Once this lock bit is set, no further changes can be made to NMI interrupts.

For additional information on the interrupt architecture, refer to the Architectural Options chapter in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

2.2.2 Interrupt Priorities using XEA2

Unlike XEA3, which handles all interrupt levels consistently, XEA2 has different priority classes: low (Level 1), medium (up to EXCMLEVEL), and high. XTOS supports nesting of level 1 interrupts. By default[1], all level 1 interrupts are prioritized relative to each other, with higher-numbered level 1 interrupts having higher priority over lower-numbered level 1 interrupts.

However, medium-priority and high-priority interrupts do not nest within the same interrupt priority level. For example, a level 3 interrupt will not interrupt another level 3 interrupt handler. If multiple interrupts having the same priority are pending at the same time, the lowest numbered interrupt is taken first.

In other words, only level 1 interrupts are prioritized in software. XTOS invokes level 1 handlers with the virtual interrupt mask set such that the corresponding level 1 interrupt and all level 1 interrupts of lower virtual priority are disabled. A level 1 interrupt handler must not lower the virtual interrupt priority mask so as to enable its own interrupt.

2.2.3 Writing Interrupt Handlers in C

Particular care should be taken writing an interrupt handler. Many C library functions are non-reentrant and can cause problems if called by the handler, directly or indirectly. For more details, see the C library reentrancy section in the *Xtensa Linker Support Packages (LSPs) Reference Manual*. In addition, C interrupt handlers are subject to the limitations described in Section 3.6.1 “Handling Interrupts”.

The amount of stack space available to the interrupt handler may be limited.

In XEA3, all interrupt handlers share the same stack area as the application.

In XEA2, level 1 and medium-priority interrupt handlers share the same stack area as the application. High-priority interrupt handlers use a dedicated stack for each priority level, in which only one handler runs at a time. XTOS currently allocates a default stack of 1 kB for each high-priority interrupt level, except for unused interrupt levels when using the `tiny` LSP (see above). This size is defined by `PRI_N_STACK_SIZE` in `int-highpri-dispatcher.S`.

1. This default may be reversed by modifying the `XTOS_SUBPRI_ORDER` macro in `xtos-params.h` and rebuilding XTOS as described in Section 2.7.3. Support for editing of such XTOS compile-time parameters is subject to change.

2.2.4 Interrupt Management

The following functions help designers manage interrupts. The interrupt to be operated upon is specified by its number. Interrupt numbers start from zero.

In XEA3 the number reflects the numbering assigned when configuring interrupt controller.

In XEA2 the number corresponds to the interrupt's bit position in the INTERRUPT and INTENABLE registers (interrupt number zero corresponds to bit zero). The external interrupt input signals do not map to INTERRUPT [0..n]. External interrupts may be assigned to (non-consecutive) interrupt bits. For XEA2 only, if you are using the `tiny` LSP or a derivative thereof, registering a handler is not effective unless you also call `_xtos_dispatch_level<N>_interrupts()`, where `<N>` is the priority level of the corresponding interrupt. See the description of the `tiny` LSP in the *Xtensa Linker Support Packages (LSPs) Reference Manual*, Using Linker Support Packages (LSPs) chapter, Standard LSPs section, TINY subsection.

Generally, a handler must be registered for an interrupt before enabling it. It may also be necessary to clear the source of the interrupt before enabling it to avoid a spurious interrupt that the handler might not expect.

Edge-triggered interrupts are automatically cleared by XTOS, except for timer interrupts in XEA2. These must be cleared in the handler by writing the corresponding CCOMPAREn register, using Xtensa HAL. Level-triggered interrupts are usually cleared externally in a device-specific manner.

Startup code sets the current interrupt priority level to level zero, *i.e.*, enabling all interrupt levels, which start at level 1. Individual interrupts start off disabled, and must be enabled and disabled under program control. A given interrupt is taken if it is pending, it has been enabled, and its priority level is higher than the current processor priority level.

XTOS provides higher-level functions for setting up interrupt handlers, and enabling or disabling interrupts. The HAL provides lower-level functions for querying and setting interrupt properties. See Section 3.7 “Interrupts” for details.

Note: For XEA2 configurations, the functions described below do not simply manipulate the `INTENABLE` special register directly, nor simply return the previous value of `INTENABLE`. This is because the `INTENABLE` register serves two purposes:

- Enabling and disabling specific interrupts under application control.
- Masking interrupts according to the current virtual interrupt level or priority.
  Such masking is used to implement finer-grained software prioritization among level-one interrupts.

Thus, the `INTENABLE` special register is always set as a function of two global variables: the set of interrupts currently enabled, and the current virtual interrupt priority. And of course, in XEA3 configurations the `INTENABLE` register does not exist at all.

When using XTOS, do not modify the `INTENABLE` register directly. Instead, use the provided functions to safely manipulate interrupts. This will allow your code to continue to work independent of the underlying interrupt architecture.

xtos_set_interrupt_handler

This function registers a C interrupt handler to be called for a particular interrupt. It returns zero to indicate success, and -1 to indicate failure. Pass `0` (NULL) as the handler function pointer `handler` to cease handling the specified interrupt. An argument for the handler is specified in `param`. Pass NULL if no argument is required. If the argument `pprev` is not NULL, then the location it points to is filled in with the address of the last handler registered for this interrupt, or NULL if no handler had been registered.

```
typedef void (*xtos_handler)(void *arg);
int32_t xtos_set_interrupt_handler(uint32_t intnum, xtos_handler
handler, void * param, xtos_handler * pprev);
```

This function will return -1 to indicate an error if:

- The interrupt number is invalid.
- The interrupt cannot be dispatched to a C handler (XEA2).
- The core was configured without the interrupt option.

For XEA3, all interrupts can be dispatched to C handlers.

For XEA2, XTOS can dispatch all types of interrupts to a C handler, except NMI. Refer to Section 2.2.3 for limitations of using C functions for high-level interrupts.

_xtos_set_interrupt_handler
_xtos_set_interrupt_handler_arg

Note: These older functions are retained for backward compatibility but will be deprecated in a future release. You are encouraged to use `xtos_set_interrupt_handler()`.

These functions register a C interrupt handler to be called for a particular interrupt. Both return the address of the last handler registered for the specified interrupt, or `0` (NULL) if none were registered. Pass `0` (NULL) as interrupt handler function pointer `f` to cease handling the specified interrupt.

If _xtos_set_interrupt_handler_arg() is used to register, the argument `arg` will be passed to the handler when it is invoked. Calling `_xtos_set_interrupt_handler()` will result in the interrupt number being provided as the handler argument.

```
typedef void (*_xtos_handler)(void);
_xtos_handler _xtos_set_interrupt_handler(int interrupt, _xtos_handler
f);
_xtos_handler _xtos_set_interrupt_handler_arg(unsigned int interrupt,
_xtos_handler f, void *arg);
```

xtos_interrupt_enable
xtos_interrupt_disable

```
int32_t  xtos_interrupt_enable(uint32_t intnum);
int32_t  xtos_interrupt_disable(uint32_t intnum);
```

The `xtos_interrupt_enable()` function enables (turns "on") the specified interrupt. It is typically used to enable an individual interrupt when the application is ready to handle it. Generally, an application must use `xtos_set_interrupt_handler()` to register a handler for an interrupt *before* enabling that interrupt. The function `xtos_interrupt_disable()` disables (turns "off") the specified interrupt.

Both functions return zero on success and -1 on error. They return an error if:

- The interrupt number is invalid.
- The core was configured without the interrupt option.

Either function can be called from C code. They can be called from the application, an interrupt handler (running at level `XTOS_LOCKLEVEL` or lower), or an exception handler registered using `xtos_set_exception_handler()`.

_xtos_interrupt_enable
_xtos_interrupt_disable

```
void  _xtos_interrupt_enable(unsigned int intnum);
void  _xtos_interrupt_disable(unsigned int intnum);
```

Note: These older functions are retained for backward compatibility but will be deprecated in a future release. Use the newer ones described above.

xtos_interrupt_enabled

```
int32_t  xtos_interrupt_enabled(uint32_t intnum);
```

This function reports the enabled status of a specified interrupt. It returns zero if the interrupt is disabled, 1 if the interrupt is enabled, and -1 on error. It returns an error if:

- The interrupt number is invalid.
- The core was configured without the interrupt option.

xtos_interrupt_trigger
xtos_interrupt_clear

```
int32_t  xtos_interrupt_trigger(uint32_t intnum);
int32_t  xtos_interrupt_clear(uint32_t intnum);
```

The `xtos_interrupt_trigger()` function triggers the specified interrupt, if possible. It is typically used to trigger a software interrupt. Not all interrupts can be triggered by this function. See the databook for your processor to determine which interrupts can be triggered this way. Generally, an application must use `xtos_set_interrupt_handler()` to register a handler for an interrupt *before* triggering that interrupt. The function `xtos_interrupt_clear()` clears the specified interrupt. Edge-triggered interrupts (except timers) are automatically cleared by XTOS.

Both functions return zero on success and -1 on error. They return an error if:

- The interrupt number is invalid.
- The core was configured without the interrupt option.

The functions cannot detect whether the specified interrupt can in fact be triggered or cleared by this method.

Either function can be called from C code. They can be called from the application, an interrupt handler (running at level `XTOS_LOCKLEVEL` or lower), or an exception handler registered using `xtos_set_exception_handler()`.

XTOS_SET_INTLEVEL
XTOS_SET_MIN_INTLEVEL
xtos_set_intlevel
xtos_set_min_intlevel

These macros and functions set the current interrupt level mask. This mask disables interrupts up to the specified interrupt level. Interrupt levels range from 0 through `XCHAL_NUM_INTLEVELS`.

```
uint32_t  XTOS_SET_INTLEVEL(uint32_t intlevel);
uint32_t  XTOS_SET_MIN_INTLEVEL(uint32_t intlevel);
uint32_t  xtos_set_intlevel(uint32_t intlevel);
uint32_t  xtos_set_min_intlevel(uint32_t intlevel);
```

`XTOS_SET_INTLEVEL()` and `XTOS_SET_MIN_INTLEVEL()` are macros. They require that `intlevel` be a constant. This allows them to expand to an RSIL instruction or other short inline assembly sequence on processors configured with XEA2. `xtos_set_intlevel()` and `xtos_set_min_intlevel()` are functions, where `intlevel` is a normal integer parameter. In XEA2, they are slower than the corresponding macros.

All four functions and macros can be called from C code. They can also be called from the application, from an interrupt handler running at XTOS_LOCKLEVEL or below, or from an exception handler registered using `xtos_set_exception_handler()`.

All of these functions and macros return a value that represents the previous interrupt level mask. This value can be passed to `XTOS_RESTORE_INTLEVEL()` or `xtos_restore_intlevel()` to restore the interrupt level mask to what it was before the interrupt level mask was set. **The format of this value must not be relied upon**.

`XTOS_SET_MIN_INTLEVEL()` and `xtos_set_min_intlevel()` are identical to the other two, except that they never lower the interrupt level. Instead, they ensure that the current interrupt level mask is at least as high as the specified `intlevel` parameter.

In XEA3, the interrupt controller must be programmed to select the desired interrupt priority level.

In XEA2, two hardware mechanisms can be used to mask interrupts up to a given level: the `PS.INTLEVEL` field and the `INTENABLE` special register (the latter also serves to enable or disable specific interrupts). Both `XTOS_SET_INTLEVEL()` and `xtos_set_intlevel()` use the former.

Startup code sets interrupt masking by interrupt level to level zero, *i.e.*, enabling all interrupts. Individual interrupts are all disabled at startup. A given interrupt is taken if it is pending, it has been enabled using `xtos_interrupt_enable()`, and its level is higher than what is currently masked by interrupt level. For low-level interrupts, the interrupt must also be a higher interrupt number than any currently executing low-level handler.[2]

Example:

To disable level-one interrupts during a critical section, assuming this code is never executed when the current interrupt level mask might be greater than one:

```
uint32_t rval = XTOS_SET_INTLEVEL(1);
... critical section code ...
XTOS_RESTORE_INTLEVEL(rval);
```

If the code can be executed when the current interrupt level mask might be greater than one, use `XTOS_SET_MIN_INTLEVEL()` rather than `XTOS_SET_INTLEVEL()`.

_xtos_set_intlevel
_xtos_set_min_intlevel

```
unsigned  _xtos_set_intlevel(int intlevel);
unsigned  _xtos_set_min_intlevel(int intlevel);
```

Note: These older functions are retained for backward compatibility but will be deprecated in a future release. Use the newer ones described above.

XTOS_RESTORE_INTLEVEL
xtos_restore_intlevel

This macro and equivalent function restore the current interrupt level mask according to a value returned by `XTOS_SET_INTLEVEL()`, `XTOS_SET_MIN_INTLEVEL()`, `xtos_set_intlevel()`, or `xtos_set_min_intlevel()`. For more details on the latter, refer to their description above.

```
void  XTOS_RESTORE_INTLEVEL(uint32_t restoreval);
void  xtos_restore_intlevel(uint32_t restoreval);
```

`XTOS_RESTORE_INTLEVEL()` is a macro. It may expand to a short inline assembly sequence on processors configured with XEA2, and thus be faster than the function. `xtos_restore_intlevel()` is a function.

2. The ordering of low-level priorities can optionally be reversed by recompiling XTOS.

Both function and macro can be called from C code. They can be called from the application, from a level-one interrupt handler, or from an exception handler registered using `xtos_set_exception_handler()`.

Note: On processors configured with XEA2, this macro or function may restore the entire PS register, not just `PS.INTLEVEL`.

_xtos_restore_intlevel

```
void  _xtos_restore_intlevel(unsigned restoreval);
```

Note: This older function is retained for backward compatibility but will be deprecated in a future release. Use the newer one described above.

2.2.5 Exception Management

The following function helps designers manage exception handling.

xtos_set_exception_handler

```
int32_t xtos_set_exception_handler(uint32_t excnum, xtos_handler
handler, xtos_handler * pprev);
```

This function registers a C handler for a given user vector mode exception cause. It returns zero on success and -1 on error. If the argument `pprev` is not NULL, the location pointed to is filled in with the address of the last handler that was registered for this cause, or 0 if there was none. An error is returned if:

- The exception number is invalid
- The core was configured without exceptions

The `excnum` argument specifies an exception cause whose occurrence is to trigger the specified handler. Exception causes and associated numbers are listed in the *Exception Option* section of the *Xtensa Instruction Set Architecture (ISA) Reference Manual*. The `<xtensa/corebits.h>` header file defines preprocessor constants for each of these causes.

The specified handler `handler` is a normal C function. It is invoked with interrupts enabled.

As with interrupt handlers, particular care should be taken writing an exception handler. Many C library functions are non-reentrant and can cause problems if invoked in this context.

XEA3 does not require a separate ALLOCA exception handler. Neither does it provide a default SYSCALL exception handler.

For XEA2 only, the current implementation pre-registers assembly-level handlers for the ALLOCA and SYSCALL exception causes. There is generally no need to override these two handlers.

If you write your own SYSCALL handler, refer to `exc-syscall-c-handler.c` for an example that properly handles SYSCALL instructions executed with a2 == 0. Such invocations of SYSCALL are defined by the ABI and must be handled correctly; they are used by some generic Xtensa code (for example, the C library) as a way of spilling live register windows to the stack.

For XEA2 only, the current implementation also pre-registers an assembly-level handler for the `LEVEL1INTERRUPT` exception cause. This handler dispatches level-one interrupts registered with the `xtos_set_interrupt_handler()` function. See Section 2.6.2 on page 21 for details.

For details on XEA2 and XEA3 exception causes, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* and the < xtensa/corebits.h> header file.

_xtos_set_exception_handler

```
typedef void (*_xtos_handler)(void);
_xtos_handler _xtos_set_exception_handler(int cause, _xtos_handler f);
```

Note: This older function is retained for backward compatibility but will be deprecated in a future release. Use the newer one described above.

2.2.6 Integrated Level-2 Cache and RAM Setup

The XTOS reset vector initializes the integrated L2 cache controller, if configured, using symbols set by the `XTHAL_L2_SETUP()` macro described in Section 3.11.2.

If this macro is not invoked by the application, XTOS by default partitions L2 memory as half L2RAM and half L2 cache. Note that this differs from the hardware reset default of assigning all L2 memory as L2RAM. To preserve this hardware default, the `XTHAL_L2_SETUP()` macro must be invoked accordingly, for example:

```
#include <xtensa/core-macros.h>

// This must be invoked outside any function:
XTHAL_L2_SETUP(XCHAL_L2RAM_RESET_PADDR, 16, 0);
```

2.2.7 Multi-Core Support

XTOS provides some basic synchronization support for single-image multi-core applications. This is provided through two classes of objects, barriers and mutexes. Both are available only on configurations that support multi-core operation. Specifically, the PRID register and the Exclusive Store options must be present. Barrier and mutex objects are grouped together in a special XTOS data section, and must be placed with care to ensure proper operation. For a complete description of multi-core support and requirements, see Chapter 7 Building Multi-core Executables with XTOS.

2.2.7.1 Barriers

The XTOS barrier objects allow all the cores in a multi-core cluster to synchronize execution, i.e. all the cores must reach the barrier before any one of them can proceed beyond it.The current design of the barrier imposes some requirements for simplicity of implementation. The PRIDs for an N-core cluster must be numbered from 0 to N-1, sequentially with no gaps. Also, all the cores must participate in the barrier.

The barrier objects must be placed in shared memory. They can be placed in cached memory only if hardware cache coherence is available. Otherwise they must be placed in uncached memory.

A barrier is initialized by calling `xtos_barrier_init()` with the count of the number of cores that will participate. Calling `xtos_barrier_wait()` will make the caller wait until all participating cores have arrived at the barrier. The last core to decrement the barrier value will be core 0, and when this happens all the cores are released past the barrier. The `xtos_barrier_sync()` function allows the barrier to be used for mutual exclusion. When a call to this function returns, the calling core 'owns' the barrier, and can execute code without interference. All the other cores are either waiting at the barrier or have not arrived there yet.

A barrier object must be defined like this:

```
xtos_barrier  bar0;
```

The variable must be declared in global or static scope since it has to be placed in a specific section. It cannot be allocated on the stack.

xtos_barrier_init

```
void xtos_barrier_init(xtos_barrier_p pbar, uint32_t num_cores);
```

This function initializes a barrier. A barrier must be initialized before any sync/wait calls are made on the barrier. Normally core0 would initialize any barriers before the other cores are released from reset. `num_cores` must equal the number of cores in the system.

xtos_barrier_sync

```
int32_t xtos_barrier_sync(xtos_barrier_p pbar);
```

This function allows the barrier to be used for mutual exclusion. When this function returns, the calling core ’owns’ the barrier and can execute code in a critical section. The function returns 0 on success and -1 on error. This function must be called before `xtos_barrier_wait()` is called on the barrier.

xtos_barrier_wait

```
int32_t xtos_barrier_wait(xtos_barrier_p pbar);
```

This function waits on the barrier and releases the barrier when all the cores have arrived and are waiting at the barrier. It returns 0 on success and -1 on error.

2.2.7.2 Mutexes

XTOS also provides mutex objects for multi-core use. Mutex objects must be placed in shared memory. They can be placed in cached memory only if hardware cache coherence is available. Otherwise they must be placed in uncached memory.

A mutex object must be defined like this:

```
xtos_mutex  mutex0;
```

The variable must be declared in global or static scope since it has to be placed in a specific section. It cannot be allocated on the stack.

xtos_mutex_init

```
void xtos_mutex_init(xtos_mutex_p pmtx);
```

Initializes the mutex. A mutex must be initialized before it is used. Normally core0 would initialize any mutexes before the other cores are released from reset.

xtos_mutex_lock

```
int32_t xtos_mutex_lock(xtos_mutex_p pmtx);
```

Locks the specified mutex. Returns 0 on success and -1 on error. This function will busy wait until the mutex is available. It can be called repeatedly to lock an already owned mutex. Note that the locking is not protected against interrupts. This is a potentially blocking function and should not be called from an interrupt handler anyway.

xtos_mutex_unlock

```
int32_t xtos_mutex_unlock(xtos_mutex_p pmtx);
```

Unlocks the specified mutex. Returns 0 on success and -1 on error. his function can be called repeatedly to unlock the same mutex. The mutex is only released when the lock count goes to zero.

2.3 Files and Templates

The Xtensa runtime is not a complete solution. In certain cases it can be used “as is”, but in general it will require modification before use. This section examines various files that comprise the bulk of the runtime’s implementation. They can serve as starting points or templates for a production-level implementation.

Source code for the basic runtime (XTOS) implementation can be found in:

`<`*`xtensa_tools_root`*`>/xtensa-elf/src/xtos/`

All filenames given in the following section names are under this directory. Only a subset of the files in this directory are described (namely, the exception vector handlers).

Many of these files make use of the compile-time HAL, described in Section 3.3 on page 46.

2.4 Common Files crt1-***.S

These files contain the C run-time initialization code. It begins at the `_start` label, sets up the environment and initialization expected by C code, calls `main`, and handles exit from `main` appropriately.

Initialization expected by C code includes setting up a stack, clearing address register `a0` to terminate call tracebacks by a debugger, initializing the PS register and other registers specific to the selected ABI, and clearing uninitialized (`.bss`) variables. Such initialization usually also includes initializing the C library and calling C++ constructors before calling `main`, calling `exit` following any return from `main`, and arranging for C++

destructors to be called on exit. However, `crt1-tiny.S` avoids these latter steps to minimize application footprint when using the `tiny` LSP — for details, see the description of the `tiny` LSP in the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

If the Interrupt Stack Limit option is configured, the stack limit will be set up in this code (XTOS uses the interrupt stack for application code as well as interrupt handlers). The limit is initially set to 256 bytes above the lower bound of the stack area. If the stack and heap share the same segment, then calls to `_sbrk()` will move the stack limit upwards when needed as long as there is room to move it.

Generally, the reset vector jumps to `_start` after completing basic initialization (see next section). Because the C run-time initialization code is separate from the reset vector, its location and size are more flexible. For example, it can be unpacked to RAM when the reset vector is located in ROM.

2.5 XEA3 Files

This section provides descriptions of XEA3 specific files for default interrupt and exception dispatch code, reset code, XTOS default exception handler, and XTOS interrupt handler table and default interrupt handler.

Note: The dispatch code for XEA3 is a carefully designed and orchestrated set of steps. Descriptions are informative only. Users are strongly advised not to modify the dispatch code.

2.5.1 xea3/dispatch-vector-v2.S

This file contains the default interrupt and exception dispatch code. Both interrupt and exception handling follow essentially the same code path. An exception frame is allocated on the stack and live registers are saved into it. In addition, EPC, PS and some other state (depending on configuration) are also saved into the exception frame. For windowed ABI, the register window is rotated forward to emulate a CALL8. Then, specialized instructions query the interrupt controller and select the address of the handler to call based on the interrupt number received. This late choice of what to handle allows minimal latency for interrupts. If an interrupt is to be serviced, execution branches to the appropriate handler. Exceptions fall through and then branch to the exception dispatch code (see `xea3/exc_dispatch.S`), which selects the handler based on exception type.

On return from an interrupt or exception, the code checks for any pending interrupt and handles it right away (without restoring the saved state). This allows any pending interrupt to be handled with minimal latency.

The window overflow and underflow exceptions are also handled by the same dispatch code. A subset of the save and restore sequences execute to handle the exception and save or restore one register window.

2.5.2 xea3/dispatch-vector-v1.S

This file contains the dispatch code for the first version of the XEA3 implementation. It differs in some details from the later version, and is intended for use only on hardware versions RH.0 and RH.1.

2.5.3 xea3/reset-vector-xea3.S

Reset code consists of two parts executed in a sequence: the reset vector, and the C code that sets up the runtime environment and calls `main()`. The file `dispatch-vector-v2.S` only contains the code that goes into the reset vector location, which is a jump to the actual handler. The actual reset handling code is in the file `reset-vector-xea3.S`.

The initialization tasks include the following:

- Initialize the minimum set of processor registers required to complete the following steps. The reset code tries to ensure it initializes as few registers as necessary for the specific processor configuration.
- Initialize and enable the caches. This can significantly speed up the unpacking step.
- Unpack vectors and sections from ROM to RAM (see below) if necessary.
- Synchronize writeback caches, if any configured, for correct execution of unpacked code. For example, C run-time initialization code may have been unpacked to RAM.
- Jump to the C run-time (`crt1`) initialization at `_start`.

For a step-by-step look at an example reset vector, see the *Xtensa Microprocessor Programmer’s Guide*.

When a program placed in ROM has code that is to be executed from RAM, some mechanism must exist to place the code into RAM. The default reset vector provides code for this purpose that unpacks various program sections at reset. Each LSP defines a table (possibly empty) that describes the location and content of data to be moved from ROM into RAM at initialization. More information about ROM unpacking can be found in the *Xtensa Linker Support Packages (LSPs) Reference Manual*, as well as the *Xtensa Microprocessor Programmer’s Guide*.

More information about C run-time initialization can be found in Section 2.4, and in the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

2.5.4 xea3/exc_dispatch.S

This file implements the exception dispatcher. This code uses the exception cause to look up the handler from the exception handler table and transfers control to it.

2.5.5 xea3/exc_default_handler.S

This file implements the XTOS default exception handler. This handler will simply trap to the simulator or debugger if present.

2.5.6 xea3/exc_table.c

Implements the XTOS exception handler table. At startup, every entry in the table will point to the default handler.

2.5.7 xea3/double-handler.S

This file implements the default double exception handler. This handler attempts to determine where the double exception occurred, saves enough state to allow it to call a user-provided handler function, and finally restores the state and attempts to return to the exception location.

2.5.8 xea3/double-handler-min.S

This is a very minimal implementation of a double exception handler. It simply halts the processor.

2.5.9 xea3/int_table.S

This file implements the XTOS interrupt handler table and the default interrupt handler. At startup, every entry in the table will point to the default handler. Handlers installed by the application will replace the default handler. The default interrupt handler simply traps to the simulator or debugger if present.

If the application installs no interrupt handlers, the interrupt table may be omitted from the final executable to save memory.

2.6 XEA2 Files

This section describes XEA2 files.

2.6.1 reset-vector-xea2.S

Reset handling consists of two parts executed in sequence: the reset vector, and the C run-time initialization that calls `main()`. This file contains the default reset vector. The initialization tasks include the following:

- Initialize the minimum set of processor registers required to complete the following steps. The reset code tries to ensure it initializes as few registers as necessary for the specific processor configuration.
- Initialize and enable the caches. This can significantly speed up the unpacking step.
- Unpack vectors and sections from ROM to RAM (see below) if necessary.
- Synchronize writeback caches, if any configured, for correct execution of unpacked code. For example, C run-time initialization code may have been unpacked to RAM.
- Jump to the C run-time (`crt1`) initialization at `_start`.

For a step-by-step look at an example reset vector, see the *Xtensa Microprocessor Programmer's Guide*.

When a program placed in ROM has code that is to be executed from RAM, some mechanism must exist to place code into RAM. The default reset vector provides code for this purpose that unpacks various program sections at reset. Each LSP defines a table (possibly empty) that describes the location and content of data to be moved from ROM into RAM at initialization. More information about ROM unpacking can be found in the *Xtensa Linker Support Packages (LSPs) Reference Manual*, as well as the *Xtensa Microprocessor Programmer's Guide*.

More information about C run-time initialization can be found in the previous section, and in the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

2.6.2 user-vector.S

This file contains the default vector and handler for user vector mode exceptions.

The user vector dispatches an assembly level handler according to EXCCAUSE.[3]
The exact calling conventions for assembly level handlers are subject to change, and are not described here. Users are expected to register C exception handlers using `_xtos_set_exception_handler()`, described in Section 2.2.4 on page 7.

3. In processors configured with XEA1, the space available at the user vector is not guaranteed big enough (28 bytes) to do the dispatch, so the user vector code jumps to a separate dispatch sequence.

The assembly level handlers for level-one interrupt, ALLOCA, and SYSCALL exception causes are in files `int-lowpri-dispatcher.S`, `exc-alloca-handler.S`, and `exc-syscall-handler.S` respectively. When a C level handler is registered using `_xtos_set_exception_handler()`, the latter function installs an assembly level handler that does the extra work needed to safely call the C function. This "wrapper" assembly level handler is located in `exc-c-wrapper-handler.S`. All other exception causes, *i.e.*, those without any specific handler, go to the default assembly level handler located in `exc-unhandled.S`.

The assembly level handler that dispatches level-one interrupts is highly optimized, due to the generally time-critical nature of interrupt handling. The level-one interrupt dispatcher sources include various alternative interrupt prioritization schemes. Providing a high degree of optimization for all alternatives leads to liberal use of conditional directives that can somewhat obscure the code’s readability. For this reason, it may be easier to read the disassembled compiled code than the sources.

A detailed look at an alternate user vector exception handler implementation is provided in the *Xtensa Microprocessor Programmer’s Guide*. Other than optimization, the major differences between this alternate user vector handler and XTOS relate to the ABI. Only XTOS implements the Call0 ABI. For the windowed ABI, they differ in the method of linking the stacks of the application and interrupt/exception handler. Upon taking an interrupt or exception, there are essentially three approaches to linking the stack for the purpose of executing windowed ABI code (*e.g.*, C code), in increasing order of performance:

1. Spill all live register windows to the stack, then execute the handler on a completely independent stack. This is slow but perhaps simpler to understand.
2. Move the interrupted function’s stack pointer to the handler’s stack (this is similar to what happens when the MOVSP instruction takes an `alloca` exception, and the exception handler moves the register save area under the new stack pointer). This involves copying 16 bytes of register save area from one stack pointer location to the other, both before and after executing the handler. This method is described in the *Xtensa Microprocessor Programmer’s Guide*.
3. Simulate a CALL4 by the interrupted function to the handler, or to a dispatch function that calls the handler. (One cannot use CALL8 or CALL12, because the interrupted function might not have allocated sufficient stack space for the overflow save area required by these calls.) This is the most efficient approach, and is implemented in current versions of XTOS. However, this implementation is not trivially modifiable to support multiple threads.

2.6.3 kernel-vector.S

This file contains the default vector for kernel vector mode exceptions. Kernel vector exceptions are general exceptions that occur when the processor is in kernel vector mode.

In XTOS, the processor normally runs in user vector mode (`PS.UM` bit set). In this mode, general exceptions become user vector exceptions, where the processor jumps to the user vector. Kernel vector exceptions are not normally expected.

Processors configured with XEA2 never modify `PS.UM` except by explicit write of `PS` by software. XTOS never clears `PS.UM`. So kernel vector exceptions should not occur. (General exceptions that occur in exception mode, *e.g.*, during critical interrupt and exception handling code, result in double exceptions; see Section 2.6.4.)

Processors configured with XEA1 automatically clear `PS.UM`, *i.e.*, switch to kernel vector mode, upon a user vector exception or high-priority interrupt. XTOS handlers switch back to user vector mode (*i.e.*, set `PS.UM`) before re-enabling interrupts or executing any C handler. Thus, kernel vector exceptions can only occur within assembly level interrupt and exception handlers and dispatchers, which in XTOS are designed not to cause any exception themselves; and custom (user-written) high-priority interrupt vector code should usually avoid causing exceptions. So any kernel exception is a sign of a bug.

The kernel vector code does very little. It takes a breakpoint if the debug option is enabled; then just spins in an infinite loop.

2.6.4 double-vector.S

This file contains the default vector for double exceptions. Double exceptions exist only in Xtensa Exception Architecture 2 (XEA2). They are general exceptions that occur when the processor is already in exception mode (`PS.EXCM` bit set).

This double exception vector is only a stub. A full handler is only required when exercising a full MMU (XTOS does not), where double exceptions may be a normal occurrence (*e.g.*, for page faults within the window exception handlers). Otherwise, double exceptions are generally a symptom of a bug, *i.e.*, of an exception occurring in an exception or interrupt handler's critical section(s). They can be a useful debugging tool.

2.6.5 debug-vector.S

This file contains the default vector for debug exceptions.

When assembled for the simulator, it lets the simulator (ISS) handle the exception.

When assembled otherwise, it just does an infinite loop. This version is used by default by LSPs targeting hardware, such as for Xtensa-related emulation boards. Note that when debugging applications using OCD, this debug vector is not invoked - the debug exception is handled by OCD software instead.

Running a debugger agent on the target requires a much more elaborate debug exception handler than the minimal one included in this file.

2.6.6 int-vector.S

This file contains template source code for all vectors of interrupts (other than debug interrupts) at levels greater than one. The Makefile assembles this template once for each interrupt level greater than one. Implementation is trivial: saving a register and jumping to the appropriate interrupt dispatcher.

2.6.7 int-handler.S

This file is simply a wrapper that selects between three methods of interrupt-handler dispatching for interrupts at levels greater than one. Three methods of handler dispatching are required for various types of interrupts, which are:

- medium-priority interrupts (always handled in C code)
- high-priority interrupt handled in C code
- high-priority interrupt handled in assembly language

(There is actually a fourth type of interrupt called level-one interrupt, or low-priority interrupt. The Xtensa ISA, however, classifies level-one interrupts as a general exception, so they are handled in `int-lowpri-dispatcher.S` and related files.)

File `int-handler.S` includes the appropriate source file that implements the proper interrupt-dispatching method. Method selection is based on settings of XTOS parameters, processor configuration, and the interrupt level for which the template is being assembled. The Makefile assembles this template once for each interrupt level greater than one.

2.6.8 int-medpri-dispatcher.S

This file contains template source code for dispatching handlers for medium-priority interrupts. Medium-priority interrupts are at an interrupt level greater than one and less than or equal to EXCM Level. The Makefile assembles this template once for each medium-priority interrupt level in the processor configuration.

2.6.9 int-highpri-dispatcher.S

This file contains template source code for dispatching handlers for high-priority interrupts written in C. A high-priority interrupt is an interrupt at a level greater than EXCM Level. The Makefile assembles this template once for each high-priority interrupt level in the processor configuration at `XTOS_LOCKLEVEL` and below.

High-priority interrupt handlers may be written in C for those high-priority interrupts at level `XTOS_LOCKLEVEL` and below. Those higher than `XTOS_LOCKLEVEL` must be written in assembly language. `XTOS_LOCKLEVEL` defaults to the highest interrupt level defined in the processor configuration (defined in *<xtensa_tools_root>*`/xtensa-elf/src/xtos/xtos-params.h`). To avoid unnecessary increased latency, designers should ensure that `XTOS_LOCKLEVEL` does not include those interrupt levels for which the handlers are written in assembly language.

Dispatching high-priority interrupts written in C imposes a significant performance cost. (This overhead is smaller with the call0 ABI.)

2.6.10 int-highpri-template.S

This file contains skeleton template code for handling high-priority interrupts in assembly language. Designers prefer assembly-language handlers when high-priority performance is critical. The Makefile assembles this template once for each high-priority interrupt level in the processor configuration above `XTOS_LOCKLEVEL`.

These handlers are incomplete skeleton handlers that must be modified to be useful. They never get invoked unless the associated interrupt(s) are explicitly enabled using `_xtos_ints_on()`, and triggered.

By default, the XTOS runtime disables all interrupts at startup (except NMI if present). Also by default, XTOS assumes that all high-level interrupts have handlers written in C. Designers can replace default XTOS interrupt-handling functionality by the methods described in Section 2.7 on page 31.

2.6.11 Vector Code vs. Handler Code

Separating handler code in two, with one small portion located at the vector and another in the general code (usually `.text`) section, allows more flexibility on the total size of the handler without affecting the number of bytes required at the vector itself. This latter number is often referred to as the "vector size". Configuration of medium-priority and high-priority interrupt vector sizes defaults to only 28 bytes, and on T1050 and earlier processors to merely 12 bytes (with 4 bytes of preceding literal space), just enough for a minimal code sequence that jumps to the actual handler elsewhere in memory. This is why there are two files for each medium-priority and high-priority interrupt: one for the tiny sequence at the vector, and one for the actual handler, providing more flexibility on the size of the handler in the default case. See Section 2.6.13 on page 27 for further discussion.

2.6.12 Symbols From Template Source Code

Medium-priority and high-priority interrupt vectors and dispatchers employ a number of template files to simplify implementation for the configurable Xtensa architecture. However, the resulting vector, handler, and library-module symbol names are not obvious. This section enumerates the names that these templates produce.

For each interrupt vector at level N, where N is 2 or greater, the `int-vector.S` template generates the symbols shown in Table 2–2.

Table 2–2. Names and Symbols From int-vector.S Template

Symbol	Description
`.Level<N>InterruptVector.text`	Vector code is located in this section name with the `.section` directive
`.Level<N>InterruptVector.literal`	Literal values associated with the vector code are located in this section (when using PC-relative L32R)
`_Level<N>Vector`	Symbol marking starting point of the vector code
`_Label<N>FromVector`	Symbol to which the vector code jumps; assumes outside module defines
`Level<N>InterruptVector.o`	Library archive module name and intermediate object name

The vector code in XTOS always jumps to a generic interrupt dispatcher, which can be viewed as preliminary interrupt handling. However, at interrupt levels two and above, three types of interrupt dispatching are handled by three separate template files (see Section 2.6.7 on page 24). While interrupt dispatching methods differ, the template files all generate the names and symbols, shown in Table 2–3.

Table 2–3. Names and Symbols From int-handler.S Templates

Symbol	Description
`_Label<N>FromVector`	Start of the interrupt-dispatching code to which the vector code jumps
`Level<N>InterruptVectorHandler.o`	Library archive module name and intermediate object name

2.6.13 Strategies for Handling High-Priority Interrupts in Assembly

Multiple interrupts may be configured at the same high-priority interrupt level. In this case, any one of these interrupts can cause the corresponding high-priority interrupt handler to be invoked. The handler can determine the set of interrupts that caused its invocation using the `INTERRUPT` register. If any of the interrupts is a level-triggered external interrupt, that set can be empty due to spurious interrupts—*i.e.*, depending on the external device asserting the interrupt, it is possible for the external interrupt to be de-asserted between the time the interrupt is taken and the time the `INTERRUPT` register is read.

Normally, the high-priority interrupt handler must clear the source of the interrupt(s) before returning. Otherwise, the interrupt(s) will be taken again immediately after the handler returns. When multiple interrupts are mapped to the same level, the handler may resolve and clear at least one interrupt and return immediately. When the interrupt is taken again, the next one can be handled, and so on.

The method of clearing the interrupt source depends on the type of interrupt. Edge-triggered external interrupts and software interrupts are cleared by writing to the `INTCLEAR` register. The skeleton interrupt handlers do this already as an example. However, other types of interrupts are cleared differently, and the skeleton handlers don't do anything for these cases. Timer interrupts are cleared by writing to the corresponding timer's `CCOMPARE`*n* register. Level-triggered external interrupts are cleared by signaling the device that is asserting the interrupt to stop asserting it—the method by which this is done is device-dependent (and/or system dependent).

From a system design perspective, high-priority interrupts handled in assembly languages should be used when interrupt response needs to be very fast. Reducing the interrupt-handler dispatching overhead as low as possible is the primary goal.

The structure of assembly-language high-priority interrupt handlers is very simple. The only work to be done with the address registers is saving and restoring the registers that the handler will use.

Because the Xtensa Processor Generator allows the positioning and sizing of the interrupt vectors, the designer can increase this default vector size to be large enough to contain the entire handler (or the latency-critical portion thereof).

This approach ensures that no jump is required from the vector entry to the handler itself. With an appropriately sized vector, the code to handle the high-priority interrupt can be put right in line at the vector location.

The overall structure of a high-priority interrupt handler is as follows:

```
Save the registers to be used
Perform interrupt operation
Restore the registers
Return
```

Saving registers generally requires at least one address register. High-priority interrupt handlers must assume all address registers are in use when a high-priority interrupt is taken. Thus, some space is required to save one address register to start the register-saving process. The `EXCSAVE`*n* registers serve this purpose. There is one such scratch register for each interrupt level (including `EXCSAVE1` for general use by user and kernel vector exceptions, which include level-one interrupts). The skeleton vectors and handlers provide an example of how to use these registers to save and restore the processor state.

2.6.14 memerror-vector.S

This is the memory error vector and handler. It is relevant if the Memory ECC/Parity Option is configured. This handler attempts to recover from any memory error encountered. Some cases are non-recoverable, in which case by default the handler will run in an infinite loop. This default behavior can be modified by adapting the code as needed (see Section 2.7).

Note: The memory error vector and handler code and data should be located in uncached memory. Putting them in cached memory increases the probability of an unrecoverable double memory error due to cache tag errors.

By default, memory error checking is disabled. It must be enabled before there can be any memory error exception. XTOS provides two functions for this purpose:

```
void _xtos_memep_initrams( void );
void _xtos_memep_enable( int flags );
```

Before enabling memory error checking, the state of relevant local memories and caches must be initialized. XTOS already initializes all caches at reset. However, ECC/parity bits of local memories (instruction and data RAMs) may need to be initialized if the entire RAMs have not been written to. One may call `_xtos_memep_initrams()` to initialize any ECC and/or parity bits of instruction and data RAMs. This function simply loads and stores every word of every instruction and data RAM. It is usually safe to call if the caller can ensure that no interrupt can occur, or that any interrupt handler avoids writing to local instruction or data RAMs. Otherwise, the repeated load-then-store sequence is not atomic, and any local memory modification by an interrupt handler may be lost.

Once ECC/parity bits of local memories are appropriately initialized, memory error checking for both data accesses and instruction fetch can be enabled by calling `_xtos_memep_enable()`. This simply writes a default value to the `MESR` register. The `flags` parameter is present for future additions only, and is currently ignored. The caller is expected to pass zero.

Thus, a typical sequence to enable memory error checking, possibly called very early in `main()` (before enabling any interrupts), is as follows:

```
    _xtos_memep_initrams();
    _xtos_memep_enable(0);
```

For more details on the Memory ECC/Parity Option, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

Table 2–4 describes how the XTOS handler deals with various error cases. In some cases, a user-defined hook function will be called if one is provided.

Table 2–4. Memory Error Cases

Where	Type Access	Locked	Dirty	Correctable	Writeback	Lockable	Result
I-Cache Data	fetch	FALSE		TRUE			I-cache line invalidated
I-Cache Data	fetch	FALSE		FALSE			I-cache line invalidated
I-Cache Data	fetch	TRUE		TRUE			Corrected cache line written
I-Cache Data	fetch	TRUE		FALSE			I-cache line invalidated and relocked
I-Cache Data	LICW						Not supported/ Undefined
I-Cache Tag		FALSE		TRUE			I-cache line invalidated
I-Cache Tag		TRUE		TRUE			Corrected cache line written
I-Cache Tag				FALSE			Invalidated/user hook relocks if needed User hook: _xtos_merr_hook_ icache_relock
D-Cache Data	Load or Castout	FALSE			TRUE		D-cache line writeback (of corrected value) and invalidated

Where	Type Access	Locked	Dirty	Correctable	Writeback	Lockable	Result
D-Cache Data	Load	TRUE		TRUE			D-cache line writeback (of corrected value) and invalidated then relocked
D-Cache Data	Load	FALSE	FALSE	FALSE			D-cache line invalidated
D-Cache Data	Load	TRUE	FALSE	FALSE			D-cache line invalidated and relocked
D-Cache Data	Load or Castout		TRUE	FALSE	TRUE		Fatal/infinite loop User hook: _xtos_merr_hook_ uncor_dirty
D-Cache Data	LDCW						Not supported/ Undefined
D-Cache Tag		FALSE		TRUE			D-cache line writeback (of corrected value) and invalidated
D-Cache Tag		TRUE		TRUE			Corrected cache tag written
D-Cache Tag				FALSE	TRUE		Fatal/infinite loop User hook: _xtos_merr_hook_ uncor_dtag
D-Cache Tag				FALSE	FALSE	TRUE	Invalidated/user hook relocks if needed User hook: _xtos_merr_hook_ dcache_relock
D-Cache Tag		ANY		FALSE	FALSE	FALSE	D-cache line invalidated
I-RAM		N/A	N/A	TRUE			Corrected word written to memory

Where	Type Access	Locked	Dirty	Correctable	Writeback	Lockable	Result
I-RAM		N/A	N/A	FALSE			Fatal/infinite loop User hook: _xtos_mem_err_hook_ uncorrectable_local
D-RAM		N/A	N/A	TRUE			Corrected word written to memory
D-RAM		N/A	N/A	FALSE			Fatal/infinite loop User hook: _xtos_mem_err_hook_ uncorrectable_local

Notes:

- The “Writeback” and “Lockable” columns are properties of the cache, and are fixed for the core.
- The “Locked” and “Dirty” columns are properties of the cache line.
- LDCW and LICW are cache test instructions and are not supported by the memory error handler. If executing one of these causes a memory error exception, the result is undefined and should be treated as fatal.

2.6.15 window-vectors.S

These are the register window overflow and underflow exception handlers. They are only built when the windowed ABI is selected, which requires a processor configured with register windows, that is, with 32 or 64 address registers. The code in these handlers is fixed according to the windowed ABI. There should be no need for this code to be modified. These implementations should serve for every set of Xtensa software that uses register windows. For a detailed discussion of the window exception vectors, refer to the *Window Exception Handlers* chapter of the *Xtensa Microprocessor Programmer’s Guide*.

2.7 Customizing XTOS

Cadence delivers XTOS as source code and as prebuilt binaries in the form of library archives, `libhandlers-*.a`. To use XTOS, designers link their application against the appropriate XTOS library archive by selecting a Linker Support Package (LSP) that supports it (*i.e.*, using the `-mlsp=<lsp>` argument). Refer to the *Xtensa Linker Support Packages (LSPs) Reference Manual* for more details on LSPs.

There are two ways to tailor XTOS: *link-time override*, and *rebuilding sources*. Each is described in detail in the following subsections. Either method preserves the core package installation, leaving the original XTOS source code and libraries unmodified.

Link-time override works well when modifying specific exception and interrupt vectors and handlers. For more extensive changes, note that Cadence periodically updates the sources to XTOS. If you later upgrade to a newer version of Xtensa Tools, you may need to either integrate Cadence's changes into your sources, or gather and maintain the older versions of the entire XTOS sources and headers with your modifications.

2.7.1 Link-Time Override

The GNU linker combines the object files listed on its command line, processing them from left to right. The `xt-xcc` compiler driver's `-mlsp=`<*lsp*> argument causes any libraries specified in that LSP to be added on at the end of the linker command line, *i.e.*, to the right-most position on the link line. The linker searches library archive files only to finalize any remaining unresolved symbol references while processing left to right.

Designers can take advantage of this linker behavior when customizing XTOS for their application. Instead of modifying and rebuilding XTOS, designers can simply copy XTOS source code into one of their application files and modify it there. If such an application file is listed on the link line before (*i.e.*, to the left of) library archive files, the linker will prefer the designer's versions of the XTOS functions. That is, the linker will find and resolve the XTOS symbols first from the designer's application, and the linker will not search the library archive files for those XTOS symbols.

Using this approach, designers can preserve the original XTOS source code and avoid rebuilding the XTOS library archives.

For example, assume that a designer has increased the size of the level 3 high-priority interrupt vector and wants to locate a lightning-fast, assembly-language handler entirely in the vector itself. He could copy the contents of file `int-vector.S` into his application and replace the template descriptions with hard-coded label definitions for the entry-point label and the assembler-directive arguments. Careful definition of label `_Level3Vector` (and preservation of assembler directives `.global`, `.begin`, `.section`, and `.align`) will cause the linker will prefer the application's definition of the level-3 interrupt vector. The linker will not look to resolve these symbols from the library archives specified later on the linker command line. The developer then can leave the XTOS libraries and build infrastructure unmodified.

Note: A library archive file is a single file holding a collection of other files. Each file in the archive is called a member file. When the linker is looking to resolve a symbol and finds it in a library archive, the entire member file (but not the entire archive) is pulled into the link. The implication is that when designers want to copy XTOS source code into their application, they must copy the entire file contents and not just a single function or subset of functions from the file. Failure to do so will cause the linker to complain about multiple definitions for a symbol.

2.7.2 Link-Time Overrides for Standard XTOS

The following XTOS functions and symbols can be overridden at link time to provide custom behavior while still using the original XTOS libraries.

User-Defined MPU Table

If the Xtensa configuration includes an MPU, then a user-defined table can be provided to set up the MPU, replacing the default MPU setup performed by the reset handler. Another user-defined value must specify the size of the table (i.e. the number of entries in the table). The number of entries must not exceed the maximum number supported by the MPU (XCHAL_MPU_ENTRIES).

`__xt_mpu_init_table`	Name of user-defined table of MPU entries
`__xt_mpu_init_table_size`	Number of entries in the user-defined table.The allowed values are from 1 up to XCHAL_MPU_ENTRIES

For ROMable images, the above data must reside in a section that does not need to be unpacked (such as .ResetHandler.text or .srom.text). This is required because the MPU is programmed before any ROM unpacking takes place.

An example user definition might look like this:

```
const unsigned int

__xt_mpu_init_table_size __attribute__((section(".ResetHandler.text")))
= 4;

const struct xthal_MPU_entry

__xt_mpu_init_table[4] __attribute__((section(".ResetHandler.text"))) =

{

  XTHAL_MPU_ENTRY(0x00000000, 1, XTHAL_AR_NONE, XTHAL_MEM_DEVICE),

  XTHAL_MPU_ENTRY(0x00010000, 1, XTHAL_AR_RWX,  XTHAL_MEM_WRITEBACK),
  XTHAL_MPU_ENTRY(0xA5A50000, 1, XTHAL_AR_RWX,  XTHAL_MEM_DEVICE),
  XTHAL_MPU_ENTRY(0xFFFE0000, 1, XTHAL_AR_RWX,  XTHAL_MEM_WRITEBACK)
};
```

User-Defined ROM Unpacking/Custom Initialization

The default ROM unpacking code in the reset handler may not be suitable or optimal for all possible cases. Also, the destination of the unpack may require some setup; for instance, a DDR controller may need to be initialized and enabled before anything is unpacked into external memory. To support such cases, XTOS allows a user-defined hook function to replace the default unpacking code. If a user-defined hook is present, then it is called and its return value checked. If the return value is nonzero, it indicates that code unpacking has been completed successfully. This hook function must be written in assembly and should make minimal assumptions about system state. The function should not expect to be called with a valid stack pointer, and cannot assume that any global or static variables have been initialized.

```
      __reset_user_init
```

The hook receives its return address in a0 and returns a result in a2.

User-Defined Stack Initialization

The default XTOS stack setup can be overridden by providing a custom function for stack setup. The function must be named as follows:

`__stack_init`	User-defined stack setup function

This code cannot be written as a C function, since there is no valid stack yet and the BSS section has not been zeroed out. The code should make minimal assumptions about the state of the system.

User-Defined Memory Map Initialization

As part of setting up the C runtime environment, the BSS segments are zeroed out. Any initialization that affects the memory map needs to happen before the BSS segments are zeroed, and a user hook is provided for that purpose. This hook works if there is no ROM unpacking involved, or where ROM unpacking is not affected by memory remapping. If ROM unpacking is affected, then memory remapping must be done before ROM unpack, in the reset handler (see Section 2.7.2.2). The function must be named as follows:

```
      void __memmap_init (void);
```

This hook can be a C function, however remember that BSS has not been initialized yet. In particular, this function cannot set BSS variables (uninitialized globals) nor can it assume that they have been initialized to zero.

User-Defined BSS Initialization

A user-defined hook function can be provided to override the default BSS initialization procedure. The function must be named as follows:

`__bss_init`	User-defined BSS initialization function

This can be a C function, but as above, must not rely on the state of the BSS contents. It is called with two parameters:

`_bss_table_start`	Start of the BSS segment table
`_bss_table_end`	End of the BSS segment table

Refer to the XTOS sources for the layout of the BSS segment table.

User-Defined MEMCTL Register Setup

Two user-defined symbols can be used to override the default initial values for the MEMCTL register. The symbols must be named as follows:

`__memctl_default`	User-defined MEMCTL register initial value
`__memctl_default_post`	User-defined MEMCTL register post value

Normally, `__memctl_default` initializes the caches and `__memctl_default_post` sets up snoop, loop buffer, and branch prediction behavior.

These must be defined as symbols and not variables, for example:

```
xt-xcc test.c -g -o test -Wl,--defsym=__memctl_default=<value>
```

Or in assembly:

```
.global     __memctl_default

.equ        __memctl_default     <value>
```

User-Defined PREFCTL Register Setup

A user-defined symbol can be used to override the default initial value for the PREFCTL register. The symbol must be named as follows:

`__prefctl_default`	User-defined PREFCTL register initial value

This must be defined as a symbol and not a variable (see examples above).

User-Defined L2 Cache/Memory Setup

A user-defined symbol can be used to override the default L2 cache/memory setup. The symbol must be named as follows:

_Xthal_L2_ctrl_init	User-defined value for L2 setup

To define your own custom value for this override, you can use the macro XTHAL_L2_SETUP. This must be defined as a symbol and not a variable (see examples above).

2.7.3 Rebuilding Sources

Link-time override is not the most convenient method for making widespread or global changes, such as modifying global XTOS parameters. Such changes are sometimes more easily done by editing a copy of the relevant source tree and rebuilding all of it.

This section describes how to rebuild distributed target libraries and objects from sources. This technique applies to XTOS, the HAL library, and simulator and board-specific target libraries. Sources for these components are located in Xtensa Tools, in subdirectories `xtos`, `hal`, `sim`, and `board`, of the following directory:

<xtensa_tools_root>`/xtensa-elf/src`

Rebuilding these sources requires separate build and installation directories where object files and libraries are built and installed. The build process does not modify source directories, as long as they are kept separate from build and install directories.

To build, first create a small file (typically named `Makefile.info`) that specifies all required parameters, in Makefile format:

```
XTENSA_TOOLS_ROOT = <xtensa_tools_root>
XTENSA_ROOT = <xtensa_root>
INSTLIBDIR = <install-dir>
INSTCLIBDIR = <clib-specific-install-dir>
```

where:

<*xtensa_tools_root*> is the location of Xtensa Tools;

<*xtensa_root*> is the location of the processor configuration's core package;

<*install-dir*> is the destination directory for resulting libraries and objects; and

<*clib-specific-install-dir*> is a similar install directory for libraries and objects whose sources included C library header files and thus have dependencies on the specific C library being used.

The two latter installation paths can point to the same directory. The simplest method of using these directories to build an application is to specify them (or one if they are the same) using the `-L` <*dir*> command-line parameter to `xt-xcc` or `xt-ld` when linking the application. An alternative is to create a custom LSP (derived from a standard one), point these install paths to the custom LSP's directory so that the build sequence installs generated objects and libraries in it, and use that custom LSP for linking. Refer to the *Xtensa Linker Support Packages (LSPs) Reference Manual* for details on using LSPs.

If your environment is set up correctly for the desired processor configuration and Xtensa Tools version, the first two lines in `Makefile.info` can be written generically like this:

```
XTENSA_TOOLS_ROOT = $(shell xt-xcc --show-config=xttools)
XTENSA_ROOT = $(shell xt-xcc --show-config=config)
```

This uses GNU make features to query Xtensa Tools for the needed directory paths. It assumes that the PATH environment variable points to Xtensa Tools binaries, and that the XTENSA_SYSTEM and XTENSA_CORE environment variables select the desired Xtensa processor configuration.

Second, create the build directory (<*build-dir*>) in a convenient location. In the build directory, create a small Makefile (named `<build-dir>/Makefile`) as follows:

```
MAKEFILE_INFO = <path>/Makefile.info
include $(MAKEFILE_INFO)
MAKEFILE_SRC = <path-to-src>/Makefile.src
include $(MAKEFILE_SRC)
```

where:

<*path*> is the location of the `Makefile.info` file described above; and

<*path-to-src*> is the location of the directory containing the relevant sources (XTOS, HAL, etc.) whether it be the original sources or your own copy.

Once the Makefile is created, just invoke the following to build and install:

```
xt-make -C <build-dir> all install
```

For example, if `Makefile.info` is located just above the build directory, the following `Makefile` in the build directory lets one rebuild XTOS from the original sources:

```
MAKEFILE_INFO = ../Makefile.info
include $(MAKEFILE_INFO)
MAKEFILE_SRC = $(XTENSA_TOOLS_ROOT)/xtensa-elf/src/xtos/Makefile.src
include $(MAKEFILE_SRC)
```

To rebuild modified XTOS sources, first copy the entire `xtos` subdirectory to a separate directory, and make appropriate changes there. Direct modifications of installed Xtensa Tools files is discouraged. For example, to modify the default global XTOS parameter `XTOS_LOCKLEVEL`, first copy the `xtos` directory to another location we'll call <myxtos>. Edit the `<`*`myxtos`*`>/xtos-params.h` file as needed. Modify the above example `<`*`build-dir`*`>/Makefile` to set MAKEFILE_SRC to `<`*`myxtos`*`>/Makefile.src`. Rebuild XTOS by invoking `xt-make` as described above. And rebuild your application by specifying `-L <`*`install-dir`*`>` on the link line so that this newly rebuilt XTOS library gets chosen in preference to the default one. Or alternatively, point `<`*`install-dir`*`>` to a custom LSP as suggested above.

Note: Modifying standard header files, such as those that get installed under `<`*`xtensa_tools_root`*`>/xtensa-elf/include`, may require modifying the target source Makefiles (`Makefile.src`) to direct the compiler to use the modified headers rather than the default ones.

3. Xtensa Processor Hardware Abstraction Layer (HAL)

The Xtensa Processor Hardware Abstraction Layer (HAL) describes, and to some extent abstracts, the configurable and extensible portions of the Xtensa core. The Xtensa HAL enables writing software that functions properly on many or all possible Xtensa processor configurations and extensions. Although very useful for application level code, the Xtensa HAL is particularly relevant for OS and system software development. Such run-time code is typically sensitive to a greater number of Xtensa processor configuration parameters and of properties of custom extensions. Using the HAL, an OS or application can potentially work on any Xtensa processor.

The Xtensa HAL comes in two forms. First is the *compile-time HAL*, a C header file described in Section 3.3 on page 46 that describes a processor configuration. Second is the *link-time HAL*, in the form of a configuration-specific library and a configuration independent header file defining the API. It provides processor configuration-specific information and functionality at run-time, and is described in most of the following sections.

Many operating systems have architecture and board-specific hardware abstraction layers also termed "HAL". The Xtensa HAL is not to be confused with these. The Xtensa HAL is OS-independent. It purely describes and abstracts features of the *processor* itself, whereas most operating system specific "HALs" abstract the features of a hardware system *outside* the processor.

Note: Refer to Appendix A for a list of deprecated, renamed, and dropped features.

3.1 Overview of the HAL

The Xtensa HAL provides comprehensive information about the Xtensa processor configuration and extensions. For example, it describes endianness, presence of optional instructions, interrupt map, size and location of any local memories, and many other configuration parameters relevant to software. This is particularly useful to operating system and system software, and to certain aspects of applications written to be portable across multiple Xtensa processors.

The link-time HAL also provides basic support functions that are highly dependent on the configurable portions of the Xtensa architecture. This especially includes functions that can much more efficiently be tuned to the particular configuration parameter(s) at compile-time than at run-time. It also provides functionality that is very particular to the Xtensa architecture, such as spilling register windows to the stack.

This makes Xtensa HAL an operating system and system software enabler. It provides information and functionality necessary to produce a configuration-independent runtime.

The Xtensa HAL is provided as a library and as a header file. It is not an environment. The existence of a base architecture gives the designer enough stability to provide a runtime framework outside the Xtensa HAL.

The Xtensa HAL provides enough information and abstraction of custom TIE instructions, states, etc. to write a single runtime system or kernel software that works fully regardless of what those custom extensions might be. This includes, for example, size and alignment requirements of custom state, and functions to save and restore that state to enable automated context switching. It does not provide total information about TIE constructs; for example, it does not provide detailed instruction encodings and semantics.

3.1.1 Using Xtensa HAL with TIE

It is important to note that in the present implementation, the HAL is generated by the Xtensa Processor Generator (XPG), but not by the TIE compiler (TC). Thus, if you are developing a set of custom processor extensions using TIE, these extensions are only reflected in the HAL once you've generated and downloaded a processor core package using the XPG from within Xtensa Xplorer. The HAL headers and libraries are not updated when generating a TDK using TC.

3.1.2 Providing Source Level Portability Across Xtensa Processor Variants

The most straightforward and code-efficient approach to providing portability across many or all Xtensa processor variants (processor configurations and extensions) is to do so at the source level. In this approach, one need only recompile and/or re-assemble all relevant code from sources to target each unique Xtensa processor configuration.

Where source code has a dependency on Xtensa processor variances, it can simply include the appropriate compile-time HAL header file and use the relevant configuration information accordingly. It can alternatively use the link-time HAL, where this is more convenient. For example, cache operations and windowed ABI stack spilling are typically more easily done using existing link-time HAL routines than by rewriting them oneself.

3.1.3 Providing Binary Portability Across Xtensa Processor Variants

Occasionally, code needs to be available in a binary (compiled and assembled) form that works on a wide range of Xtensa processor configurations. This is common, for example, with certain third party software libraries.

A number of elements are required to generate portable binaries for Xtensa processors. One of these is the existence of a *base ISA* in the Xtensa architecture. There is a minimal set of instructions guaranteed to be present in every configured Xtensa processor.[1] This minimal set is more than sufficient for C compiler support. However, you also need a set of software tools capable of restricting itself to this minimal set. Compiling and assembling code for this minimal set, that is, building the portable binary, generally requires a mythical processor configuration that implements only that set. In such a pseudo processor subset configuration, all options that add instructions or similar functionality are turned off, and no custom extensions are specified. It is possible for a customer with full access to the Xtensa processor generator to configure such a processor. Such a subset configuration is just one element, however; others are needed to achieve binary portability. For example, a subset is not sufficient to deal with caches; use of a link-time cache abstraction, such as provided by the link-time HAL discussed below, is necessary for that purpose. Also, dealing with ABI variants as discussed below may require multiple subset configurations.

Important Note: Contact Cadence support if you intend to construct portable binaries. This section does not describe how to configure an appropriate processor subset configuration, and may gloss over important points. Even if such description were provided, most Xtensa processor users and third parties do not have access to the full Xtensa Processor Generator, and thus cannot configure an appropriate subset processor configuration. The main purpose of this section is to provide appropriate context for certain uses of the link-time HAL, as well as for certain aspects of its design.

A portable binary is usually in the form of a library. It is compiled using the pseudo processor subset configuration. However, the final executable image must be built (linked) using the specific processor configuration's software. This ensures that the correct processor-specific libraries are pulled-in, and that various software tools correctly identify the specific targeted processor. Binary portability of a post link-time binary, that is, of a fully linked executable program image, is beyond the scope of this document.

Typically, a given library need not be binary compatible with *every* possible Xtensa processor configuration. Rather, some set of constraints or additional minimal requirements of the processor configuration may be appropriate. For example, libraries written for a specific set of processor extensions (such as the Xtensa HiFi2 engine) naturally expect those extensions to be present, and may add other requirements as well. In that case, potential users of this library need to ensure that their Xtensa processor configuration satisfies those requirements. This allows the library to take advantage of processor features that its target user base is expected to have available. Another example is an RTOS that might require processor features typical of processors designed to run an RTOS, such as interrupts, a timer, and related features. See the *Xtensa OSKit Guide* for examples of existing supported third party RTOS and their associated minimum processor configuration requirements.

1. It is possible for this minimum set to become smaller in a major release of the Xtensa processor, by making optional some components that previously were always present. Conversely, new options may also be added in a release of the Xtensa processor. Both can affect binary compatibility.

Another element that helps address binary portability is the link-time HAL. The link-time HAL library of a specific Xtensa processor configuration can be linked with portable library code to produce an executable image targeted to that specific processor configuration. The portable library can access link-time HAL routines and global variables to obtain information about the processor configuration and extensions.

The link-time HAL cannot help provide portability of a post link-time binary, *i.e.*, of a fully linked executable program image. Again, binary portability of such images is beyond the scope of this document.

Remember that Xtensa instructions are encoded differently in big- and little-endian configurations. It is thus not possible to provide binary portability across endianness. The typical solution, if both need to be supported, is to provide both big and little endian versions of a library (binary code).

Software conventions also affect binary portability. At present, two software options are in this category: ABI selection and access to literals. Because they affect binary compatibility, these particular options are selected *before* building and downloading the core package that configures Xtensa Tools to a specific Xtensa processor configuration. Again, unless it is extremely simple, a single binary cannot be portable across these software options.[2] The typical solution is to pick the most appropriate one. Alternatively, one can pre-compile a library for all possible option combinations. However, when including endianness, this results in a total of eight (8) possible combinations given the following three options:

- Big endian vs. little endian
- Windowed ABI vs. CALL0 ABI
- Absolute literals vs. PC-relative literals (Extended L32R Software Option)

There are other costs to binary compatibility: performance and code size. These are to be expected, but worth mentioning here.

Let's take the MUL32 option as an example. This option adds a MUL32 instruction that implements single-cycle 32-bit multiplication. Code targeting processors with and without that instruction can simply be compiled without MUL32 support. However, this means that when running on a processor that has this option, such code won't take advantage of the MUL32 instruction.[3] An alternative might be to detect at run-time (e.g.

2. It *is* possible for functions in both ABIs to co-exist in a single binary, as long as they are kept separate and don't call each other. For example, where interrupt handlers use the CALL0 ABI and the application uses the Windowed ABI, without shared code, and with appropriately customized interrupt dispatch code. It is also possible to mix absolute and PC-relative literals, as long as the `LITBASE` register is carefully updated when switching literal modes. In fact, existing reset vectors must do this when absolute literals (Extended L32R Software option) are in use. Nevertheless, because these options cannot arbitrarily mix and match, they affect binary portability in the general sense.

3. In the case of MUL32, some of the lost performance is regained by linking using the targeted processor's software. Without a multiplication instruction and when not multiplying by certain constants, the compiler emits a call to a multiplication function in the targeted processor's `libgcc` library.

using the link-time HAL) whether the MUL32 instruction is present, and use it if so. However, this is both bigger and slower than recompiling for the specific processor configuration, meaning bigger and slower than source portability.

In some cases, the cost of dealing with configurability at link-time or run-time may become undesirable compared to compile-time. For example, one might want to squeeze every cycle of performance out of some code sequence, such as interrupt dispatch. In this case, it may be appropriate for the library being distributed in binary form to have a small layer of code distributed in source form for performance reasons. This way, these sources can be designed to deal with configurability at compile-time. In a sense, the link-time HAL is one such layer designed to deal with configurability at compile-time. It provides generic functionality, as opposed to application or library specific functionality.

3.1.4 Calling Conventions

The link-time HAL provides functions in the configured register calling conventions (windowed ABI or CALL0 ABI). Also, a few functions are provided that use non-standard non-windowed register calling conventions useful in the context of exception or interrupt handling code.

Both the windowed ABI and the CALL0 ABI (calling conventions) are described in detail in the *Using the Xtensa Architecture* chapter of the *Xtensa Instruction Set Architecture (ISA) Reference Manual*. These calling conventions are summarized in Table 3–5 in terms of general register usage. The `CALL4`, `CALL8`, and `CALL12` columns correspond to the windowed ABI.

Note that XEA3 supports only `CALL8` (Fixed Window) for windowed ABI.

Table 3–5. Non-Windowed (CALL0) and Windowed (CALL4,8,12) Calling Conventions

Caller Registers				Callee Registers	Description
CALL0	CALL4	CALL8	CALL12		
—	`a0-a3`	`a0-a7`	`a0-a11`	—	Saved by the `CALL`*n*
`a0`	`a4`	`a8`	`a12`	`a0`	Return PC
`a1`	`a5`	`a9`	`a13`	`a1`	Stack pointer
`a2`	`a6`	`a10`	`a14`	`a2`	1st argument / return value
`a3`	`a7`	`a11`	`a15`	`a3`	2nd argument / return value
`a4-a7`	`a8-a11`	`a12-a15`	—	`a4-a7`	3rd thru 6th arguments
`a8-a11`	`a12-a15`	—	—	`a8-a11`	Caller saved
`a12-a15`				`a12-a15`	Callee saved (CALL0)
	—	—	—	`a12-a15`	Avail. to callee (windowed)

Alternative Entry Points

If your processor configuration's software targets the windowed ABI, the link-time HAL also provides CALL0 ABI versions of certain functions (with names suffixed with `_nw`), so that they are available to low-level code in the simpler CALL0 ABI. If your processor configuration's software targets the CALL0 ABI, the link-time HAL also provides both functions (with and without the `_nw` suffix), but as a single function implementation with two names (labels).

The windowed ABI has many implicit requirements that are not always practical in the context of low-level kernel code such as interrupt and exception handlers. In particular, certain `PS` or `WB` register fields must have specific values (for example, `WB.S` zero in XEA3, `WOE` set and `EXCM` clear in XEA2,or `INTLEVEL` clear in XEA1); and the stack pointer, `WB` in XEA3, `WindowBase` and `WindowStart` in XEA2 and XEA1, and all address registers, must be consistent with each other according to the windowed ABI. Applications and kernel code need to invoke the link-time HAL from both C code, where use of the windowed ABI may have been configured, and from assembly code in execution states that do not satisfy the requirements of the windowed ABI. It is for this reason that certain link-time HAL operations are given an alternate entry point that always adheres to the CALL0 ABI.

The first entry point is a normal C callable entry point using the configured ABI. Hence for example, with the windowed ABI, the first instruction at the entry point is an `ENTRY` instruction, and it is invoked with a `CALL`*N* or `CALLX`*N* instruction (where `N` is 8 for XEA3, and `N` is 4, 8 or 12 for XEA2/XEA1).

The second entry point always implements the CALL0 ABI. The CALL0 mechanism can easily be invoked in the context of an assembler-level interrupt or exception dispatcher. Therefore, most operations of the link-time HAL also have a CALL0 ABI (non-windowed) entry point, distinguished by an `_nw` appended to the label name (see Section 3.1.5). The entry point is to be invoked with the `CALL0` instruction and will return using the `RET` instruction.

Note: The `_nw` function variants are meant to be called from assembly code. Although safe to call from C when configured to use the CALL0 ABI, doing so just makes C code non-portable to the Windowed ABI. In C, simply call the normal functions (without `_nw`).

3.1.5 Naming Conventions

All entry points in the link-time HAL begin with the prefix `xthal_`. All entry points that support the non-window calling convention are suffixed with `_nw`. Data references in the link-time HAL are prefaced with `Xthal_`. As examples:

```
void xthal_function();          // safe to call from C.
void xthal_function_nw();       // callable from assembly, not from C
int32_t Xthal_data;             // some data reference.
```

Functions in the link-time HAL that accept arguments will accept as parameters all possible valid configuration values. For example, Xtensa processors can have up to eight coprocessors. The `xthal_save_cpregs(void *`*save_area*`, int `*the_cp*`)` function accepts values from zero through seven for parameter *the_cp*. It will accept these values independent of the number of coprocessors actually configured. Even if the designer has configured but one coprocessor with ID zero, the values one through seven will still be accepted. In the case that a possible option is not actually present, the link-time HAL will do nothing and return benign values.

3.2 Architectural Constants

The Xtensa HAL header file (see Section 3.4.1 on page 64) defines a number of constants set by the Xtensa Instruction Set Architecture (ISA). Table 3–6 lists these constants. Their values are independent of any particular Xtensa processor configuration. It is possible, although unlikely, for these constants to change across Xtensa product releases.

Note: A particular implementation, or microarchitecture, of the Xtensa processor does not necessarily support the full range of all these architectural limits. This is illustrated in Table 3–6, which indicates that the current Xtensa product release supports a lower maximum number of interrupt levels and timers than allowed by the ISA.

Table 3–6. HAL Architectural Constants

Preprocessor Macro Name	Value	Description
XTHAL_MAX_CPS	8	Maximum configurable number of coprocessors
XTHAL_MAX_INTERRUPTS	256/32	Maximum configurable number of interrupts For XEA2/XEA1: 32
XTHAL_MAX_INTLEVELS	16	Maximum configurable number of interrupt levels, including level zero For XEA2/XEA1: implementation max is 7
XTHAL_MAX_TIMERS	4	Maximum number of timers (implementation max is currently 3)

The Xtensa HAL header file also defines preprocessor constants that report the current Xtensa product release, listed in Table 3–7. These are also independent of any specific Xtensa processor configuration, but vary by definition from release to release.

Note: These constants reflect the release of Xtensa product software used to build (typically target code), not of the hardware targeted by the software. There is a distinction from the first point one does a software upgrade for a given processor hardware. Also, when compiling code to be distributed in binary form and later linked with the link-time HAL to obtain configuration information, the link-time HAL's version of these constants may differ from these compile-time constants.

Table 3–7. HAL Version Constants

Preprocessor Macro Name	Type	Description
XTHAL_RELEASE_MAJOR	numeric	Software release (e.g. 12000 for 12.0.3, 1040 for T1040.2)
XTHAL_RELEASE_MINOR	numeric	Software release (e.g. 3 for 12.0.3, 2 for T1040.2)
XTHAL_RELEASE_NAME	string	Software release name (e.g. "12.0.3", or “T1040.2”)
XTHAL_RELEASE_INTERNAL	string	(Cadence use only internal release name, defined only for internal releases; *e.g.*, "devel" for "12.0.3-devel")
XTHAL_MAJOR_REV	numeric	Same as XTHAL_RELEASE_MAJOR (for backward compatibility only)
XTHAL_MINOR_REV	numeric	Same as XTHAL_RELEASE_MINOR (for backward compatibility only)

3.3 Compile-Time HAL

The Compile-Time Hardware Abstraction Layer (or compile-time HAL) provides macro definitions that describe configuration parameters of a specific Xtensa processor core. In typical use, it describes the Xtensa processor for which code is being compiled.

The set of configuration information provided is similar to (though not exactly the same as) that provided by the link-time HAL. However, because the information is provided in the form of C pre-processor macros rather than as a library, it is available at compile-time rather than at run-time (or at link-time).

This distinction has a number of advantages, mainly performance. On the basis of information and functionality provided by the link-time HAL, software can make many system and architectural decisions at run-time. However, each decision that can be removed from execution and pushed back into the source code at compile time can free up a few cycles of execution time. The compile-time HAL exists to help remove run-time decisions and increase run-time efficiency. The compile-time HAL is also sometimes a cleaner interface for obtaining configuration information, and doesn't require linking the HAL library.

Like the link-time HAL, the compile-time HAL provides configuration information that designers can use to improve the portability of their source code across multiple Xtensa processor configurations. Source code that properly uses these macro definitions will readily adjust (i.e., after a recompile) to any configuration of an Xtensa processor.

When designing and writing source code, designers should first consult the compile-time (or link-time) HAL when faced with the need for configuration information rather than make assumptions about their processor configuration. They are likely to find a macro already defined and suitable for the specific task.

3.3.1 Processor Core Configuration Information

C source code typically includes the compile-time HAL header file as follows:

```
#include <xtensa/config/core.h>
```

Assembly source code can include this header file in a similar fashion, but must first define the `_ASMLANGUAGE` macro to avoid including C definitions. A simpler and more common approach for assembly code is to include the `coreasm.h` file instead, which automatically defines `_ASMLANGUAGE` and includes the compile-time HAL header file, in addition to defining a number of assembler macros useful for abstracting certain configuration options:

```
#include <xtensa/coreasm.h>
```

The contents of the compile-time HAL header file (`core.h`) are extensive. Interested readers can browse this header file. The core.h file itself is now fixed, and located in:

```
<xtensa_tools_root>/xtensa-elf/include/xtensa/config/core.h
```

which includes other headers generated for a given Xtensa configuration, mostly in:

```
<xtensa_root>/xtensa-elf/arch/include/xtensa/config/core-isa.h
<xtensa_root>/xtensa-elf/arch/include/xtensa/config/tie.h
```

The general way in which the compile-time HAL defines most configuration information is not likely to change, however certain details such as exact macro names for some configuration parameters might change in a future release.

Note: The compile-time HAL header file includes the link-time HAL's header file `<xtensa/hal.h>`. This is only natural, because the link-time HAL header file provides configuration-independent definitions, whereas the compile-time HAL header file extends that with configuration-specific definitions.

Much of the compile-time HAL header file definitions describe the details of the selected Xtensa architecture options. This includes boolean values indicating whether various options are configured, numerical values, addresses of vectors and local memories, and other configuration parameters, as well as values derived from configuration parameters provided for convenience. Addresses are given in hexadecimal using the `0x` prefix which is compatible with both C and assembly. The header file also offers several utility macros to help simplify the creation of configuration-independent source code.

Note: Boolean macros are defined with a value or `0` or `1`, rather than by defining or not defining the macro. Thus conditionalizing on these is generally done with `#if` rather than `#ifdef` (this is an easy mistake to make for those more accustomed to the use of `#ifdef`).

Compile-time HAL configuration constants are listed in Table 3–8 below. All preprocessor macro names are uppercase and prefixed with “`XCHAL_`”.

Compile-time HAL assembler macros are listed in Table 3–9 on page 58. All assembler macro names are lowercase and prefixed with “`xchal_`”.

Table 3–8. Compile-Time HAL Configuration Constants

Preprocessor Macro Name	Description
Coprocessors and Custom State	
XCHAL_HAVE_CP	Coprocessor option flag (CPENABLE register present only if set)
XCHAL_CP_NUM	Number of coprocessors
XCHAL_CP_MAX	Max coprocessor ID plus one (0 if none)
XCHAL_CP_MASK	Bit mask of configured coprocessors
XCHAL_CP_PORT_MASK	Coprocessors with TIE ports only (subset of XCHAL_CP_MASK).
XCHAL_NCP_SA_SIZE	Non-coprocessor optional and custom state save area size (bytes)
XCHAL_NCP_SA_ALIGN	Non-coprocessor optional and custom state save area minimum alignment (bytes)
XCHAL_NCP_SA_NUM	Number of registers in non-coprocessor save area, for debugging
XCHAL_NCP_SA_LIST(*s*)	List of registers in non-coprocessor save area, for debugging. User must define own XCHAL_SA_REG() macro to expand. See core-specific `tie.h` file for details.
XCHAL_NCP_NUM_ATMPS	Number of temporary addr. regs needed by xchal_ncp_{load,store}
XCHAL_CP*n*_NAME	Coprocessor name (quoted string) for coprocessors *n* = 0 .. 7
XCHAL_CP*n*_IDENT	Coprocessor name (unquoted) for coprocessors *n* = 0 .. 7
XCHAL_CP*n*_SA_SIZE XCHAL_CP_SA_SIZE(*n*)	Save area sizes (bytes) for coprocessors *n* = 0 .. 7

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	**Description**
XCHAL_CP*n*_SA_ALIGN XCHAL_CP_SA_ALIGN(*n*)	Save area min. alignment (bytes) for coprocessors *n* = 0 .. 7
XCHAL_CP*n*_SA_NUM	Number of registers in coprocessor *n* save area, for debugging
XCHAL_CP*n*_SA_LIST(*s*)	List of registers in coprocessor *n* save area, for debugging. User must define own XCHAL_SA_REG() macro to expand. See core-specific `tie.h` file for details.
XCHAL_CP*n*_NUM_ATMPS	Number of temporary addr. regs needed by xchal_cp*n*_{load,store}
XCHAL_TOTAL_SA_ALIGN	Combined required alignment for all coprocessor and non-coprocessor state save areas
XCHAL_TOTAL_SA_SIZE	Combined required size (bytes) for all coprocessor and non-coprocessor state save areas, assuming 16-byte alignment padding
XCHAL_SA_NUM_ATMPS	Max num of temp addr. regs needed by any state load/store macro
XCHAL_CP_ID_*name*	Coprocessor ID (0 .. 7) for given uppercase coprocessor *name*
Interrupts	
XCHAL_HAVE_INTERRUPTS	Interrupt option flag
XCHAL_HAVE_HIGHLEVEL_INTERRUPTS	High-priority interrupt option flag. Not meaningful for XEA3, always 0.
XCHAL_NUM_INTERRUPTS	Number of interrupts
XCHAL_NUM_INTLEVELS	Number of interrupt levels (not including level zero)
XCHAL_EXCM_LEVEL	Highest interrupt level masked when PS.EXCM is set (XEA2 only; always 1 in T10XX releases) Not meaningful for XEA3 since PS.EXCM does not exist. Defined as XCHAL_NUM_INTLEVELS.
XCHAL_INTLEVEL*n*_MASK XCHAL_INTLEVEL_MASK(*n*) XCHAL_INTLEVEL_MASKS	Masks of interrupts for levels *n* = 0 .. 15 (Mask for level zero is always zero.) Not meaningful for XEA3, undefined.
XCHAL_INTLEVEL*n*_ANDBELOW_MASK XCHAL_INTLEVEL_ANDBELOW_MASK(*n*) XCHAL_INTLEVEL_ANDBELOW_MASKS	Masks of interrupts in levels 0..*n* for *n* = 0 .. 15 (Mask for level zero is always zero.) Not meaningful for XEA3, undefined.
XCHAL_INTLEVEL*n*_NUM XCHAL_INTLEVEL_NUM(*n*)	If only one interrupt at level *n* = 0 .. 15, this is its number 0 .. 31; else XCHAL_INTLEVEL*n*_NUM is undefined Not meaningful for XEA3, undefined.
XCHAL_INT*n*_LEVEL XCHAL_INT_LEVEL(*n*) XCHAL_INT_LEVELS	XEA3: levels of all configured interrupts. Note that this is the reset value and may be changed by software. XEA2/XEA1: levels of interrupts *n* = 0 .. 31 (0 for unconfigured interrupts) (`XCHAL_NMILEVEL` for NMI)

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	Description
XCHAL_HAVE_NMI	Non-maskable interrupt option flag. Always 0 for XEA3 since there is no hardwired NMI.
XCHAL_NMILEVEL	XEA3: always undefined. XEA2/XEA1: NMI pseudo-level used for the `RFI n` instruction and {`EXCSAVE`,`EPS`,`EPC`}`_n` registers; equals `XCHAL_NUM_INTLEVELS+1` (if NMI configured)
XCHAL_NMI_INTERRUPT	XEA3: undefined, no hardwired non-maskable interrupt. XEA2/XEA1: non-maskable interrupt number
XCHAL_WRITE_ERROR_INTERRUPT	Write-error interrupt number
XCHAL_INT*n*_TYPE XCHAL_INT_TYPE(*n*) XCHAL_INT_TYPES	Types of interrupts XEA3: n = 0 .. `XCHAL_NUM_INTERRUPTS` XEA2/XEA1: n = 0 .. 31 (values are `XTHAL_INTTYPE_*` macros)
XCHAL_INTTYPE_MASK_UNCONFIGURED	Mask of unconfigured interrupts Not meaningful for XEA3, undefined.
XCHAL_INTTYPE_MASK_SOFTWARE	Mask of software interrupts Not meaningful for XEA3, undefined.
XCHAL_INTTYPE_MASK_EXTERN_EDGE	Mask of external edge-triggered interrupts Not meaningful for XEA3, undefined.
XCHAL_INTTYPE_MASK_EXTERN_LEVEL	Mask of external level-triggered interrupts Not meaningful for XEA3, undefined.
XCHAL_INTTYPE_MASK_TIMER	Mask of timer (`CCOMPAREn`) interrupts Not meaningful for XEA3, undefined.
XCHAL_INTTYPE_MASK_NMI	Mask of non-maskable interrupt Not meaningful for XEA3, undefined.
XCHAL_INTTYPE_MASKS	Interrupt mask per type Not meaningful for XEA3, undefined.
XCHAL_NUM_EXTINTERRUPTS	Number of external interrupts (level-triggered, edge-triggered, or NMI)
XCHAL_EXTINT*n*_NUM	Interrupt numbers for external interrupts *n* = 0 .. `XCHAL_NUM_EXTINTERRUPTS-1`
XCHAL_EXTINT*n*_LEVEL	Interrupt levels for external interrupts *n* = 0 .. `XCHAL_NUM_EXTINTERRUPTS-1`
Timers	
XCHAL_HAVE_CCOUNT	Timer option flag (`CCOUNT` register)
XCHAL_NUM_TIMERS	Number of timers (`CCOMPAREn` registers)

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	Description
XCHAL_TIMER*n*_INTERRUPT XCHAL_TIMER_INTERRUPT(*n*) XCHAL_TIMER_INTERRUPTS	Interrupt numbers for timers *n* = 0 .. 3
Windowed Address Registers	
XCHAL_HAVE_WINDOWED	Windowed address registers option flag (always 1)
XCHAL_NUM_AREGS_LOG2	Number of address registers log2
XCHAL_NUM_AREGS	Number of address registers
Caches	
XCHAL_ICACHE_LINEWIDTH	Size of inst. cache line in bytes log2
XCHAL_DCACHE_LINEWIDTH	Size of data cache line in bytes log2
XCHAL_ICACHE_LINESIZE	Size of instruction cache line in bytes
XCHAL_DCACHE_LINESIZE	Size of data cache line in bytes
XCHAL_CACHE_LINEWIDTH_MAX	Max(ICACHE_LINEWIDTH, DCACHE_LINEWIDTH)
XCHAL_CACHE_LINESIZE_MAX	Max(ICACHE_LINESIZE, DCACHE_LINESIZE)
XCHAL_ICACHE_SETWIDTH	Instruction cache sets (lines per way) log2
XCHAL_DCACHE_SETWIDTH	Data cache sets (lines per way) log2
XCHAL_CACHE_SETWIDTH_MAX	Max(ICACHE_SETWIDTH, DCACHE_SETWIDTH)
XCHAL_CACHE_SETSIZE_MAX	Max(ICACHE_SETSIZE, DCACHE_SETSIZE)
XCHAL_ICACHE_WAYS	Instruction cache set associativity
XCHAL_ICACHE_WAYS_LOG2	log2(XCHAL_ICACHE_WAYS) or 0 if no icache
XCHAL_DCACHE_WAYS	Data cache set associativity
XCHAL_DCACHE_WAYS_LOG2	log2(XCHAL_DCACHE_WAYS) or 0 if no dcache
XCHAL_L1SCACHE_WAYS	L1S cache set associativity
XCHAL_L1SCACHE_WAYS_LOG2	log2(XCHAL_L1SCACHE_WAYS) or 0 if no L1S cache
XCHAL_ICACHE_SIZE	Size of instruction cache in bytes
XCHAL_ICACHE_SIZE_LOG2	log2(XCHAL_ICACHE_SIZE) or 0 if no icache
XCHAL_DCACHE_SIZE	Size of data cache in bytes
XCHAL_DCACHE_SIZE_LOG2	log2(XCHAL_DCACHE_SIZE) or 0 if no dcache
XCHAL_L1SCACHE_SIZE	Size of L1S cache in bytes
XCHAL_L1SCACHE_SIZE_LOG2	log2(XCHAL_L1SCACHE_SIZE) or 0 if no L1S cache
XCHAL_DCACHE_IS_WRITEBACK	Data cache writeback option flag
XCHAL_DCACHE_IS_COHERENT	Data cache multiprocessor coherency option flag
XCHAL_HAVE_PREFETCH	Cache prefetch option flag

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	Description
XCHAL_HAVE_PREFETCH_L1	Prefetch to the L1 Cache
XCHAL_PREFETCH_CASTOUT_LINES	Prefetch cast-out buffer size
XCHAL_CACHE_PREFCTL_DEFAULT	Default cache prefetch control value
XCHAL_HAVE_CACHE_BLOCKOPS	Block prefetch option flag
XCHAL_HAVE_CME_DOWNGRADES	Cache management engine is configured for downgrade operations.
XCHAL_HAVE_ICACHE_DYN_WAYS	Instruction cache dynamic way support
XCHAL_HAVE_DCACHE_DYN_WAYS	Data cache dynamic way support
XCHAL_ICACHE_LINE_LOCKABLE	Inst. cache line locking option flag
XCHAL_DCACHE_LINE_LOCKABLE	Data cache line locking option flag
XCHAL_ICACHE_ECC_PARITY	Inst. cache ECC/parity option flag (0 or XTHAL_MEMEP_PARITY or XTHAL_MEMEP_ECC)
XCHAL_DCACHE_ECC_PARITY	Data cache ECC/parity option flag (0 or XTHAL_MEMEP_PARITY or XTHAL_MEMEP_ECC)
XCHAL_ICACHE_ACCESS_SIZE	Inst. cache access size (affects SICW instruction)
XCHAL_DCACHE_ACCESS_SIZE	Data cache access size
XCHAL_L1SCACHE_ACCESS_SIZE	L1S cache access size
XCHAL_L1SCACHE_BANKS	Number of L1S cache banks
XCHAL_HAVE_L2	Closely-coupled unified level-2 cache option flag
XCHAL_L2CACHE_LINESIZE	Size of L2 cache line in bytes
XCHAL_L2CACHE_LINEWIDTH	Size of L2 cache line in bytes log2
XCHAL_L2CACHE_WAYS	L2 cache set associativity
XCHAL_L2CACHE_LOCKABLE	L2 cache line locking option flag
XCHAL_L2CC_NUM_CORES	The number of NX cores connected to the L2 cache controller
XCHAL_CA_BITS	Number of bits in cache attribute encodings
XCHAL_FCA_LIST XCHAL_LCA_LIST XCHAL_SCA_LIST	Fetch access modes for attributes 0 .. $2^{cabits} - 1$ Load access modes for attributes 0 .. $2^{cabits} - 1$ Store access modes for attributes 0 .. $2^{cabits} - 1$
Debug	
XCHAL_HAVE_DEBUG	Debug option flag
XCHAL_HAVE_DEBUG_EXTERN_INT	External debug interrupt pin option flag
XCHAL_HAVE_OCD	On-chip debug (OCD) option flag
XCHAL_HAVE_OCD_DIR_ARRAY	Faster OCD option flag
XCHAL_DEBUGLEVEL	Debug interrupt level (only defined if the debug option is configured)

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	Description
XCHAL_NUM_IBREAK	Number of IBREAKA registers
XCHAL_NUM_DBREAK	Number of DBREAK{A,C} registers
XCHAL_BYTE0_FORMAT_LENGTHS	Instruction length for each value (0..255) of instruction first byte
XCHAL_OP0_FORMAT_LENGTHS	Instruction length for each op0 field value (0..15) (DEPRECATED)
Exceptions	
XCHAL_HAVE_EXCEPTIONS	Exception option flag (0 for Xtensa TX, 1 otherwise)
XCHAL_XEA_VERSION	Exception architecture version number (1 or 2 or 3, 0 for TX)
XCHAL_HAVE_XEA1	Set if XCHAL_XEA_VERSION is 1
XCHAL_HAVE_XEA2	Set if XCHAL_XEA_VERSION is 2
XCHAL_HAVE_XEA3	Set if XCHAL_XEA_VERSION is 3
XCHAL_HAVE_HALT	Set for HALT architecture option (Xtensa TX)
XCHAL_HAVE_BOOTLOADER	Set if hardware boot loader interface is configured (Xtensa TX)
XCHAL_HAVE_MEM_ECC_PARITY	Set if local memory ECC/parity option is configured
XCHAL_HAVE_VECTOR_SELECT	Set if static vector select pin present (for relocatable vectors)
XCHAL_HAVE_VECBASE	Set if VECBASE register present (for dynamic relocatable vectors)
XCHAL_VECBASE_RESET_VADDR	Reset value of VECBASE register (defined if VECBASE present)
XCHAL_RESET_VECTOR_VADDR	Address of reset vector (without relocatable vectors)
XCHAL_RESET_VECTOR_PADDR	Physical address of reset vector (without relocatable vectors)
XCHAL_RESET_VECTOR0_VADDR	Address of reset vector when vector select pin = 0
XCHAL_RESET_VECTOR0_PADDR	Physical address of reset vector when vector select pin = 0
XCHAL_RESET_VECTOR1_VADDR	Address of reset vector when vector select pin = 1
XCHAL_RESET_VECTOR1_PADDR	Physical address of reset vector when vector select pin = 1
Miscellaneous	
XCHAL_MEMORY_ORDER	Processor endianness (value is one of XTHAL_{BIG,LITTLE}ENDIAN)
XCHAL_HAVE_BE	Big-endian option flag
XCHAL_HAVE_LE	Little-endian option flag
XCHAL_HAVE_SPECULATION	Speculation option flag (always 0)
XCHAL_HAVE_DENSITY	Density (16-bit) instructions option flag
XCHAL_HAVE_BOOLEANS	Boolean option flag (for coprocessors)
XCHAL_HAVE_LOOPS	LOOP* instructions option flag
XCHAL_HAVE_NSA	NSA and NSAU instructions option flag
XCHAL_HAVE_MINMAX	MIN and MAX instructions option flag
XCHAL_HAVE_SEXT	SEXT instruction option flag

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	Description
XCHAL_HAVE_CLAMPS	CLAMPS instruction option flag
XCHAL_HAVE_MAC16	16-bit MAC (MAC16) option flag
XCHAL_HAVE_MUL16	16-bit multiply (MUL16) option flag
XCHAL_HAVE_MUL32	32-bit multiply (MUL32) option flag
XCHAL_HAVE_MUL32_HIGH	32-bit multiply MULUH, MULSH suboption flag
XCHAL_HAVE_DIV32	32-bit division (QUOS, QUOU, REMS, REMU) option flag
XCHAL_HAVE_WIDE_BRANCHES	B*.W18 or B*.W15 wide branch instructions present flag
XCHAL_HAVE_FP	Floating point option flag
XCHAL_HAVE_DFP	Double precision floating point option flag
XCHAL_HAVE_DFP_ACCEL	Double precision floating point acceleration option flag
XCHAL_HAVE_VECTRALX	Vectra LX option flag
XCHAL_HAVE_HIFI2	HiFi 2 Audio Engine option flag
XCHAL_HAVE_HIFIPRO	HiFiPro Audio Engine option flag
XCHAL_HAVE_CONNXD2	ConnX D2 package option flag
XCHAL_HAVE_FULL_RESET	Full (exhaustive, all-flops) reset option flag
XCHAL_HAVE_PRID	PRID (processor ID) register option flag
XCHAL_HAVE_RELEASE_SYNC	L32AI and S32RI (load acquire, store release) instructions option flag
XCHAL_HAVE_S32C1I	S32C1I (store conditions) instruction option flag
XCHAL_HAVE_ABSOLUTE_LITERALS	Extended L32R (absolute literals) option flag
XCHAL_HAVE_THREADPTR	THREADPTR register option flag
XCHAL_HAVE_EXTERN_REGS	WER and RER instructions option flag
XCHAL_NUM_MISC_REGS	Number of miscellaneous (scratch) special registers
XCHAL_NUM_WRITEBUFFER_ENTRIES	Number of write buffer entries
XCHAL_MAX_INSTRUCTION_SIZE	Maximum instruction size in bytes (3, 4, 8, 16, etc.)
XCHAL_INST_FETCH_WIDTH	Instruction fetch width in bytes (4, 8, 16, etc.)
XCHAL_DATA_WIDTH	Data (load/store) width in bytes (4, 8, 16, 32, 64, etc.)
XCHAL_UNALIGNED_LOAD_EXCEPTION	Unaligned loads cause exception
XCHAL_UNALIGNED_STORE_EXCEPTION	Unaligned stores cause exception
XCHAL_UNALIGNED_LOAD_HW	Unaligned loads done in hardware (special ones may cause exception)
XCHAL_UNALIGNED_STORE_HW	Unaligned stores done in hardware (special ones may cause exception)
XCHAL_HW_CONFIGID0	First (upper) 32 bits of 64-bit processor configuration ID

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	Description
XCHAL_HW_CONFIGID1	Second (lower) 32 of 64-bit processor configuration ID
XCHAL_HW_CONFIGID_RELIABLE	Flag set except for older targeted hardware releases
XCHAL_HW_VERSION_MAJOR	Targeted hardware version (*e.g.*, 1040 for T1040.2)
XCHAL_HW_VERSION_MINOR	Targeted hardware version (*e.g.*, 2 for T1040.2)
XCHAL_HW_VERSION	Targeted hardware version (major*100 + minor); matches the value of one of the XTENSA_HWVERSION_*xxx* macros in `<xtensa/xtensa-versions.h>`
XCHAL_HW_MIN_VERSION	Lower end of the range if targeting a range of hardware versions; else same as XCHAL_HW_VERSION
XCHAL_HW_MAX_VERSION	Upper end of the range if targeting a range of hardware versions; else same as XCHAL_HW_VERSION
XCHAL_HW_VERSION_NAME	Targeted hardware release name (*e.g.*, "RA-2004.1")
XCHAL_HW_RELEASE_INTERNAL	(Cadence use only internal portion of targeted hardware version, defined only if present; *e.g.*, "b" for "T1040.2b")
XCHAL_BUILD_UNIQUE_ID	22-bit ID unique per Xtensa processor software build
MMU	
XCHAL_HAVE_CACHEATTR	`CACHEATTR` register option flag (XEA1)
XCHAL_HAVE_TLBS	TLBs option flag (XEA2)
XCHAL_HAVE_SPANNING_WAY	Set if single way maps entire instr. & data virtual space *(Note: set to 0 for MMU v1 and v2, and to 1 for MMU v3)*
XCHAL_HAVE_IDENTITY_MAP	Set if virtual = physical (no translation) everywhere
XCHAL_HAVE_MIMIC_CACHEATTR	Region protection option flag (XEA2)
XCHAL_HAVE_XLT_CACHEATTR	Region protection with translation option flag (XEA2)
XCHAL_HAVE_PTP_MMU	Full MMU (with page table [autorefill] and protection) option flag
XCHAL_MMU_ASID_BITS	Number of address space ID (ASID) bits
XCHAL_MMU_ASID_INVALID	Invalid address space ID (always 0)
XCHAL_MMU_ASID_KERNEL	Kernel address space ID (always 1)
XCHAL_MMU_RINGS	Number of memory protection rings
XCHAL_MMU_RING_BITS	Number of rings log2 (rounded up)
XCHAL_MMU_CA_BITS	Number of cache attribute encoding bits
XCHAL_MMU_MAX_PTE_PAGE_SIZE	Maximum PTE page size (bytes log2, *e.g.*, 22 for 4 MB max autorefill page size) (defined only if `HAVE_TLBS`)
XCHAL_MMU_MIN_PTE_PAGE_SIZE	Minimum PTE page size (bytes log2, *e.g.*, 12 for 4 KB min. autorefill page size) (defined only if `HAVE_TLBS`)
XCHAL_ITLB_WAY_BITS	Number of instruction TLB ways log2 (up)
XCHAL_ITLB_WAYS	Number of instruction TLB ways

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	**Description**
XCHAL_ITLB_ARF_WAYS	Number of instruction TLB autorefill ways
XCHAL_DTLB_WAY_BITS	Number of data TLB ways log2 (round up)
XCHAL_DTLB_WAYS	Number of data TLB ways
XCHAL_DTLB_ARF_WAYS	Number of data TLB autorefill ways
XCHAL_KSEG_CACHED_VADDR	Virtual address of cached kernel RAM static map
XCHAL_KSEG_CACHED_PADDR	Physical address of `KSEG_CACHED`
XCHAL_KSEG_CACHED_SIZE	Size in bytes of `KSEG_CACHED`
XCHAL_KSEG_BYPASS_VADDR	Virtual address of bypass (uncached) kernel RAM static map
XCHAL_KSEG_BYPASS_PADDR	Physical address of `KSEG_BYPASS`
XCHAL_KSEG_BYPASS_SIZE	Size in bytes of `KSEG_BYPASS`
XCHAL_KIO_CACHED_VADDR	Virtual address of cached kernel I/O static map
XCHAL_KIO_CACHED_PADDR	Physical address of `KIO_CACHED`
XCHAL_KIO_CACHED_SIZE	Size in bytes of `KIO_CACHED`
XCHAL_KIO_BYPASS_VADDR	Virtual address of bypass (uncached) kernel I/O static map
XCHAL_KIO_BYPASS_PADDR	Physical address of `KIO_BYPASS`
XCHAL_KIO_BYPASS_SIZE	Size in bytes of `KIO_BYPASS`
XCHAL_SEG_MAPPABLE_VADDR	Start of largest dynamic mappable area
XCHAL_SEG_MAPPABLE_SIZE	Size in bytes of `SEG_MAPPABLE`
XCHAL_HAVE_MPU	MPU Option Flag
XCHAL_MPU_ENTRIES	Number of MPU foreground entries
XCHAL_MPU_BACKGROUND_ENTRIES	Number of MPU background entries
XCHAL_MPU_ALIGN_REQ	Foreground entries must be aligned to the background map
XCHAL_MPU_ALIGN_BITS	MPU entry address alignment in bits
XCHAL_MPU_ALIGN	MPU entry address alignment
Local Memories and Interfaces	
XCHAL_NUM_INSTROM	Number of local instruction ROM interfaces, *m* = 0 .. 1
XCHAL_INSTROM*n*_VADDR XCHAL_INSTROM*n*_PADDR XCHAL_INSTROM*n*_SIZE	Virtual address at reset of instruction ROM *n* = 0 .. *m*-1 Physical address of instruction ROM *n* = 0..*m*-1 Size in bytes of instruction ROM *n* = 0 .. *m*-1
XCHAL_NUM_INSTRAM	Number of local instruction RAM interfaces, *m* = 0 .. 2
XCHAL_INSTRAM*n*_VADDR XCHAL_INSTRAM*n*_PADDR XCHAL_INSTRAM*n*_SIZE XCHAL_INSTRAM*n*_ECC_PARITY	Virtual address at reset of instruction RAM *n* = 0 .. *m*-1 Physical address of instruction RAM *n* = 0..*m*-1 Size in bytes of instruction RAM *n* = 0 .. *m*-1 ECC or parity checking flag for this memory (0 or XTHAL_MEMEP_PARITY or XTHAL_MEMEP_ECC)

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	**Description**
XCHAL_NUM_DATAROM	Number of local data ROM interfaces, *m* = 0 .. 1
XCHAL_DATAROM*n*_VADDR	Virtual address at reset of data ROM *n* = 0 .. *m*-1
XCHAL_DATAROM*n*_PADDR	Physical address of data ROM *n* = 0 .. *m*-1
XCHAL_DATAROM*n*_SIZE	Size in bytes of data ROM *n* = 0 .. *m*-1
XCHAL_NUM_DATARAM	Number of local data RAM interfaces, *m* = 0 .. 2
XCHAL_DATARAM*n*_VADDR	Virtual address at reset of data RAM *n* = 0 .. *m*-1
XCHAL_DATARAM*n*_PADDR	Physical address of data RAM *n* = 0 .. *m*-1
XCHAL_DATARAM*n*_SIZE	Size in bytes of data RAM *n* = 0 .. *m*-1
XCHAL_DATARAM*n*_ECC_PARITY	ECC or parity checking flag for this memory (0 or XTHAL_MEMEP_PARITY or XTHAL_MEMEP_ECC)
XCHAL_NUM_URAM	Number of local unified RAM interfaces, *m* = 0 .. 1
XCHAL_URAM*n*_VADDR	Virtual address at reset of unified RAM *n* = 0 .. *m*-1
XCHAL_URAM*n*_PADDR	Physical address of unified RAM *n* = 0 .. *m*-1
XCHAL_URAM*n*_SIZE	Size in bytes of unified RAM *n* = 0 .. *m*-1
XCHAL_URAM*n*_ECC_PARITY	ECC or parity checking flag for this memory (0 or XTHAL_MEMEP_PARITY or XTHAL_MEMEP_ECC
XCHAL_NUM_XLMI	Number of XLMI interfaces or apertures *m* = 0 .. 1
XCHAL_XLMI*n*_VADDR	Virtual address of XLMI *n* = 0 .. *m*-1
XCHAL_XLMI*n*_PADDR	Physical address of XLMI *n* = 0 .. *m*-1
XCHAL_XLMI*n*_SIZE	Size in bytes of XLMI *n* = 0 .. *m*-1
Defined for XEA2/XEA1 only	
XCHAL_USER_VECTOR_VADDR	Address of user vector
XCHAL_USER_VECTOR_PADDR	Physical address of user vector, defined only if mapping is known at reset
XCHAL_USER_VECTOR_VECOFS	Offset from VECBASE register to user vector (relocatable vector option)
XCHAL_KERNEL_VECTOR_VADDR	Address of kernel vector
XCHAL_KERNEL_VECTOR_PADDR	Physical address of kernel vector, defined only if mapping is known at reset
XCHAL_KERNEL_VECTOR_VECOFS	Offset from VECBASE register to kernel vector (relocatable vector option)
XCHAL_DOUBLEEXC_VECTOR_VADDR	Address of double exception vector
XCHAL_DOUBLEEXC_VECTOR_PADDR	Physical address of double exception vector, defined only if mapping is known at reset
XCHAL_DOUBLEEXC_VECTOR_VECOFS	Offset from VECBASE register to double exc vector (relocatable vector option)
XCHAL_WINDOW_VECTORS_VADDR	Base address of window exception vectors

Table 3–8. Compile-Time HAL Configuration Constants (continued)

Preprocessor Macro Name	Description
XCHAL_WINDOW_VECTORS_PADDR	Base physical address of window exception vectors, defined only if mapping is known at reset
XCHAL_WINDOW_OF4_VECOFS	Offset from VECBASE to window overflow 4 vector (reloc. vector)
XCHAL_WINDOW_UF4_VECOFS	Offset from VECBASE to window underflow 4 vector (reloc. vector)
XCHAL_WINDOW_OF8_VECOFS	Offset from VECBASE to window overflow 8 vector (reloc. vector)
XCHAL_WINDOW_UF8_VECOFS	Offset from VECBASE to window underflow 8 vector (reloc. vector)
XCHAL_WINDOW_OF12_VECOFS	Offset from VECBASE to window overflow 12 vector (reloc. vector)
XCHAL_WINDOW_UF12_VECOFS	Offset from VECBASE to window underflow 12 vector (reloc vector)
XCHAL_INTLEVEL*n*_VECTOR_VADDR XCHAL_INTLEVEL_VECTOR_VADDR(n)	Addresses of interrupt vectors for levels *n* = 2 .. `XCHAL_NUM_INTLEVELS`
XCHAL_INTLEVEL*n*_VECTOR_PADDR	Physical addresses of interrupt vectors for levels *n* = 2 .. `XCHAL_NUM_INTLEVELS`, defined only if mapping is known at reset
XCHAL_INTLEVEL*n*_VECTOR_VECOFS	Offset from VECBASE register to interrupt level *n* vector (reloc. vector option)
XCHAL_DEBUG_VECTOR_VADDR	Address of debug exception vector (level `DEBUGLEVEL` interrupt vector)
XCHAL_DEBUG_VECTOR_PADDR	Physical address of debug exception vector, defined only if mapping is known at reset
XCHAL_DEBUG_VECTOR_VECOFS	Offset from VECBASE register to debug vector (reloc. vector option)
XCHAL_NMI_VECTOR_VADDR	Address of NMI vector
XCHAL_NMI_VECTOR_PADDR	Physical address of NMI vector, defined only if mapping is known at reset
XCHAL_NMI_VECTOR_VECOFS	Offset from VECBASE register to NMI vector (reloc. vector option)

Table 3–9. Compile-Time HAL Assembler Macros

Assembler Macro Name	Description
Coprocessors and Custom State	(Section 3.6 on page 73)
xchal_ncp_load	Load (restore) non-coprocessor custom+optional state
xchal_ncp_store	Store (save) non-coprocessor custom+optional state
xchal_extra_load_funcbody	Load (restore) non-coprocessor custom+optional state (specialized to be placed in C-callable function body)
xchal_extra_store_funcbody	Store (save) non-coprocessor custom+optional state (specialized to be placed in C-callable function body)

Table 3–9. Compile-Time HAL Assembler Macros (continued)

Assembler Macro Name	Description
xchal_cp*n*_load xchal_cp_*name*_load	Load (restore) state of coprocessor ID *n* named *name* (for defined coprocessors only)
xchal_cp*n*_store xchal_cp_*name*_store	Store (save) state of coprocessor ID *n* named *name* (for defined coprocessors only)
xchal_cp*n*_load_a2 xchal_cp_*name*_load_a2	Same as without `_a2` suffix, with implied parameters a2 and up
xchal_cp*n*_store_a2 xchal_cp_*name*_store_a2	Same as without `_a2` suffix, with implied parameters a2 and up
xchal_cpi_load_funcbody	Load (restore) state of coprocessor selected with reg. a3 (specialized to be placed in C-callable function body)
xchal_cpi_store_funcbody	Store (save) state of coprocessor selected with reg. a3 (specialized to be placed in C-callable function body)

3.3.2 Compile-Time C Macros

C source code typically includes the compile-time HAL C macros using the following header:

```
#include <xtensa/core-macros.h>
```

Interested readers can browse this header file. The `core-macros.h` file itself is now fixed, and located in:

```
<xtensa_tools_root>/xtensa-elf/include/xtensa/core-macros.h
```

Many operations that were previously implemented as C preprocessor macros are now implemented as inline functions. Using inline functions provides equivalent size and speed, but improved error checking and semantic consistency with C.

Compile-time HAL C macros and inline functions are listed in Table 3–10. All the C macro and inline function names are lowercase and prefixed with “`xthal_`”.

Note: Except for certain helper functions, most of the block operations are either “upgrade”, or block prefetch operations (they bring useful data into the cache), or “downgrade” operations as they evict data from the cache.

Table 3–10. Compile-Time HAL C Inline Functions

C Macro Name	Description
Instruction Cache	(Section 3.11 on page 117)

xthal_icache_line_invalidate()
xthal_icache_line_lock()
xthal_icache_line_unlock()
xthal_icache_sync()
	Exist as the functions in the link-time HAL as well. See Table 3–12 and Section 3.11 for details on these functions.

Data Cache	(Section 3.11 on page 117)

xthal_dcache_line_invalidate()
xthal_dcache_line_writeback()
xthal_dcache_line_writeback_inv()
xthal_dcache_line_lock()
xthal_dcache_line_unlock()
xthal_dcache_sync()
	Exist as the functions in link-time HAL as well. See Table 3–12 and Section 3.11 for details on each macro functionality.

xthal_dcache_line_prefetch_for_write()	Prefetch a cache line for future writes to cache.
xthal_dcache_line_prefetch_for_read()	Prefetch a cache line for future reads from cache.

xthal_dcache_block_invalidate()
xthal_dcache_block_writeback()
xthal_dcache_block_writeback_inv()
xthal_dcache_block_invalidate_max()
xthal_dcache_block_writeback_max()
xthal_dcache_block_writeback_inv_max()
	The cache Downgrade operations; invalidate, writeback or writeback then invalidate the cache block, respectively.
	The *_max* versions may execute faster if the block size is smaller than the cache size.

xthal_dcache_block_prefetch_for_read()	Prefetch the block in cache for future reads from cache.
xthal_dcache_block_prefetch_for_write()	Prefetch the block in cache for future writes to cache.
xthal_dcache_block_prefetch_modify()	Prefetch the block in cache and modifies the lines arbitrary in expectance of future writes. Actual memory accesses are not performed.
xthal_dcache_block_prefetch_read_write()	Prefetch the block in cache for future reads and writes to cache.

xthal_dcache_block_prefetch_for_read_grp()
xthal_dcache_block_prefetch_for_write_grp()
xthal_dcache_block_prefetch_modify_grp()
xthal_dcache_block_prefetch_read_write_grp()
	The *_grp* version of the macros above issues the prefetch request as a part of a new group.

xthal_dcache_block_wait()	Wait for all block operations to complete.
xthal_dcache_block_required_wait()	Wait for all required block operations to complete (downgrades).

Table 3–10. Compile-Time HAL C Inline Functions (continued)

C Macro Name	Description
xthal_dcache_block_abort()	End all block operations.
xthal_dcache_block_end()	End all optional block operations (prefetches).
xthal_dcache_block_newgrp()	Start new block operations group.

Note: Certain C inline functions share the same name and the functionally with the functions available in the link-time API (included with `<xtensa/hal.h>`). Those are not described in Table 3–10 because their description is given within the link-time API description. Those functions are marked in Table 3 as well in the link-time API description.

Note: When a C application includes both `<xtensa/hal.h>` and `<xtensa/core-macros.h>` the compile-time functions override the use of the link-time functions.

3.3.3 Example Definitions — Local Memories and XLMI

Whereas most definitions in the compile-time HAL are not described in detail, a specific example is often useful.

Consider the configuration of memory-mapped ports local to the processor (*i.e.*, local memories and XLMI, but not the caches). The compile-time HAL header file includes macro definitions that indicate how many of each type of port has been configured in the processor, as well as parameters describing each instance of each type of port. (The current implementation allows only two of each type.)

The following discussion describes portions of macro names as <*type*> and <n>. <*type*> stands for the type of port and is one of `INSTROM`, `INSTRAM`, `DATAROM`, `DATARAM`, or `XLMI`. <n> is a decimal integer instance count for each port type starting from zero.

The macros `XCHAL_NUM_`<*type*> define the number of instances configured for each respective port type. A value of zero indicates that no port of that type was configured. Furthermore, each instance has three associated parameters: physical address, default virtual address at reset (may differ from the physical address in configurations with an MMU), and size in bytes. These parameters are described by macros named `XCHAL_`<*type*><*n*>_<*parm*> where <*parm*> is one of `VADDR`, `PADDR`, and `SIZE`, respectively.

As an example, a configuration with one instruction ROM (`INSTROM`), two data RAMs (`DATARAM`) and no other local ports might be defined as follows:

```
#define XCHAL_NUM_INSTROM 1
#define XCHAL_NUM_INSTRAM 0
#define XCHAL_NUM_DATAROM 0
#define XCHAL_NUM_DATARAM 2
```

```
#define XCHAL_NUM_XLMI    0

#define XCHAL_INSTROM0_VADDR 0x3FFF0000
#define XCHAL_INSTROM0_PADDR 0x3FFF0000
#define XCHAL_INSTROM0_SIZE 65536

#define XCHAL_DATARAM0_VADDR 0x3FFE0000
#define XCHAL_DATARAM0_PADDR 0x3FFE0000
#define XCHAL_DATARAM0_SIZE  65536

#define XCHAL_DATARAM1_VADDR 0x50000000
#define XCHAL_DATARAM1_PADDR 0x50000000
#define XCHAL_DATARAM1_SIZE  4096
```

3.3.4 Example Uses

Here are some examples of how a designer might use the Compile-Time HAL in source code.

Selecting a Timer Interrupt

Operating systems usually require some kind of periodic timer to provide a heartbeat or tick interrupt. An OS can reasonably require a level-one timer interrupt of an Xtensa processor. The OS source code might have the following statements:

```
#include <xtensa/config/core.h>

#if XCHAL_NUM_TIMERS < 1
#error This OS requires at least one internal timer.
#endif

#if XCHAL_INT_LEVEL(XCHAL_TIMER0_INTERRUPT) != 1
#error The internal timer must be a level-one interrupt.
#endif
```

Declaring Optional State

The MAC16 and zero overhead loop options add more processor registers. An OS will typically context switch these extra registers when present. The compile-time HAL can be used to determine whether any such option has been configured. For example:

```
#include <xtensa/config/core.h>

typedef struct {
  unsigned pc;
  unsigned ps
  :
      :
    #if XCHAL_HAVE_MAC16
      unsigned acclo, acchi;
      unsigned mr[4];
    #endif
    #if XCHAL_HAVE_LOOPS
      unsigned lbeg, lend, lcount;
    #endif
      :
      :
    } ExceptionFrameType;
```

Managing Architecture Options

Source code can detect the presence of an architecture option. For increased run-time efficiency, source code can choose whether or not to include relevant functionality. Initialization code might contain the following:

```
#include <xtensa/config/core.h>

#if XCHAL_HAVE_CP

int init_coprocessors ()
{
# if XCHAL_CP_NUM == 0
    return 0;
# else
    /* initialize the coprocessors here */
# endif
}

#endif /* if XCHAL_HAVE_CP == 1 */
```

Detecting Availability of an Instruction

The Miscellaneous Operations Option includes instructions that are common and very useful in some applications. Source code could check for the existence of a specific instruction (*e.g.*, check `XCHAL_HAVE_NSA` for the `NSA` and `NSAU` instructions). If it exists, the code can select an optimal solution using the `NSA` or `NSAU` instruction; otherwise, it selects a less efficient, yet functionally correct, alternative.

3.4 Using the Link-Time HAL

3.4.1 C Code

To use the link-time HAL in C code, simply include the Xtensa HAL header file:

```
#include <xtensa/hal.h>
```

3.4.2 Assembler Code

Assembler code may also include the HAL header file, if needed. However, it must first define the `_ASMLANGUAGE` macro to instruct the header file to omit C definitions that would confuse the assembler:

```
#define _ASMLANGUAGE
#include <xtensa/hal.h>
```

Assembler code may call either the standard HAL function entry point in the configured ABI, or the function name with an `_nw` suffix using the (non-windowed) CALL0 ABI. Particular attention must be paid to the particular function's parameter passing conventions when using non-windowed entry points.

3.4.3 Linking with the HAL

The path to the HAL library for a given processor configuration is:

```
<xtensa_root>/xtensa-elf/arch/lib/libhal.a
```

Certain LSPs (see Chapter 2) automatically pull-in the HAL library, but some do not. You may need to specify the HAL library explicitly when invoking the linker using `xt-ld`, `xt-gcc`, or `xt-xcc`. This is usually done by specifying:

```
-lhal
```

to one of these commands, assuming the default linker paths.

If the default linker search path is overridden or somehow unspecified, you may need to provide the full path to the HAL library.

3.4.4 Link-Time HAL Sources

The source code to the link-time HAL is supplied in the following directory for each processor configuration:

```
<xtensa_tools_root>/xtensa-elf/src/hal/
```

See Section 2.7.3 on page 36 for instructions on rebuilding the HAL library from these sources.

The source for the link-time HAL is completely configuration independent. It uses the compile-time HAL to provide constants and functionality tailored to the particular Xtensa processor configuration. As such, it is a good example use of the compile-time HAL.

3.5 Link-Time HAL API

Table 3–11 and Table 3–12 provide a summary of link-time HAL configuration constants (read-only global variables) and functions, respectively.

Table 3–11. Link-Time HAL Configuration Constants

Global Variable	Type	Description
Software Release Info		See Section 3.2 on page 45
Xthal_release_major	uint32_t	HAL software release (e.g. 12000 for 12.0.3, or 1040 for T1040.2)
Xthal_release_minor	uint32_t	HAL software release (e.g. 3 for 12.0.3, or 2 for T1040.2)
Xthal_release_name	char *	HAL software release name (e.g. "12.0.3" or “T1040.2”)
Xthal_release_internal	char *	(Cadence use only internal release name)
Xthal_rev_no	uint32_t	HAL software release (for backward compatibility)
Coprocessors and Custom State		See Section 3.6 on page 73
Xthal_cp_num	uint8_t	Number of coprocessors
Xthal_cp_max	uint8_t	Max coprocessor ID plus one (0 if none)
Xthal_cp_mask	uint32_t	Bit mask of configured coprocessors
Xthal_extra_size	uint32_t	Extra state save area size
Xthal_extra_align	uint32_t	Extra state save area minimum alignment
Xthal_cpregs_size[]	uint32_t	Save area sizes per coprocessor
Xthal_cpregs_align[]	uint32_t	Save area minimum alignment per coprocessor
Interrupts		See Section 3.7 on page 93
Xthal_num_interrupts	uint8_t	Number of interrupts
Xthal_num_intlevels	uint8_t	Number of interrupt levels (not including level zero)

Table 3–11. Link-Time HAL Configuration Constants (continued)

Global Variable	Type	Description
Xthal_excm_level	uint8_t	Highest level of interrupts masked by PS.EXCM For XEA2 only
Xthal_intlevel_mask[]	uint32_t	Masks of interrupts per level For XEA2/XEA1 only
Xthal_intlevel_andbelow_mask[]	uint32_t	Masks of interrupts from zero to level, per level For XEA2/XEA1 only
Xthal_intlevel[]	uint8_t	Level per interrupt
Xthal_inttype[]	uint8_t	Type per interrupt
Xthal_inttype_mask[]	uint32_t	Interrupt mask per type For XEA2/XEA1 only
Timers		See Section 3.8 on page 109
Xthal_have_ccount	uint8_t	Timer option flag
Xthal_num_ccompare	uint8_t	Number of timers (`CCOMPAREn` regs)
Xthal_timer_interrupt[]	int32_t	Interrupt numbers for per timer
Windowed Address Registers		See Section 3.10 on page 115
Xthal_num_aregs_log2	uint8_t	Number of address registers log2
Xthal_num_aregs	uint8_t	Number of address registers
Caches		See Section 3.11 on page 117
Xthal_icache_linewidth	uint8_t	Size of instruction cache line in bytes log2
Xthal_dcache_linewidth	uint8_t	Size of data cache line in bytes log2
Xthal_icache_linesize	uint16_t	Size of instruction cache line in bytes
Xthal_dcache_linesize	uint16_t	Size of data cache line in bytes
Xthal_icache_setwidth	uint8_t	Instruction cache sets (lines per way) log2
Xthal_dcache_setwidth	uint8_t	Data cache sets (lines per way) log2
Xthal_icache_ways	uint32_t	Instruction cache set associativity
Xthal_dcache_ways	uint32_t	Data cache set associativity
Xthal_icache_size	uint32_t	Size of instruction cache in bytes
Xthal_dcache_size	uint32_t	Size of data cache in bytes
Xthal_L2cache_size	uint32_t	Size of L2 cache in bytes
Xthal_L2ram_size	uint32_t	Size of L2RAM in bytes
Xthal_dcache_is_writeback	uint8_t	Data cache writeback option flag
Xthal_icache_line_lockable	uint8_t	Instruction cache line locking option flag
Xthal_dcache_line_lockable	uint8_t	Data cache line locking option flag
Debug		See Section 3.12 on page 154

Table 3–11. Link-Time HAL Configuration Constants (continued)

Global Variable	Type	Description
Xthal_debug_configured	int32_t	Debug option flag
Xthal_num_ibreak	int32_t	Number of IBREAKA registers
Xthal_num_dbreak	int32_t	Number of DBREAK{A,C} registers
Xthal_byte0_format_lengths[]	uint8_t	Instruction length in bytes as a function of its first byte
Xthal_op0_format_lengths[]	uint8_t	Instruction length as function of op0 field (DEPRECATED)
Miscellaneous		
Xthal_memory_order	uint8_t	Processor endianness (XTHAL_{BIG,LITTLE}ENDIAN)
Xthal_have_windowed	uint8_t	Windowed address registers option flag (always 1)
Xthal_have_density	uint8_t	Density (16-bit) instructions option flag
Xthal_have_booleans	uint8_t	Boolean option flag (for coprocessors)
Xthal_have_loops	uint8_t	LOOP* (zero-overhead loop) instructions option flag
Xthal_have_nsa	uint8_t	NSA and NSAU instructions option flag
Xthal_have_minmax	uint8_t	MIN and MAX instructions option flag
Xthal_have_sext	uint8_t	SEXT instruction option flag
Xthal_have_clamps	uint8_t	CLAMPS instruction option flag
Xthal_have_mac16	uint8_t	16-bit MAC (MAC16) option flag
Xthal_have_mul16	uint8_t	16-bit multiply (MUL16) option flag
Xthal_have_fp	uint8_t	floating point option flag
Xthal_have_release_sync	uint8_t	Load acquire, store release (L32AI, S32RI instructions) option flag
Xthal_have_s32c1i	uint8_t	Conditional store (S32C1I instruction) option flag
Xthal_have_speculation	uint8_t	Speculation option flag (always 0)
Xthal_have_exceptions	uint8_t	Exception option flag (always 1)
Xthal_xea_version	uint8_t	Exception architecture (1 for XEA1, 2 for XEA2)
Xthal_have_interrupts	uint8_t	Interrupts option flag
Xthal_have_highlevel_interrupts	uint8_t	High-priority interrupt option flag
Xthal_have_nmi	uint8_t	Non-maskable interrupt option flag
Xthal_have_prid	uint8_t	PRID register option flag
Xthal_num_writebuffer_entries	uint16_t	Number of write buffer entries
Xthal_hw_configid0	uint32_t	First (upper) 32 of 64-bit processor configuration ID
Xthal_hw_configid1	uint32_t	Second 32 bits of 64-bit processor configuration ID
Xthal_hw_release_major	uint32_t	Targeted hardware release (*e.g.*, 1040 for T1040.2)
Xthal_hw_release_minor	uint32_t	Targeted hardware release (*e.g.*, 2 for T1040.2)
Xthal_hw_release_name	char *	Targeted hardware release name (*e.g.*, "T1040.2")

Table 3–11. Link-Time HAL Configuration Constants (continued)

Global Variable	Type	Description
Xthal_hw_release_internal	char *	(Cadence use only internal part of targeted h/w rel. name; NULL if not present;*e.g.*, "b" for "T1040.2b")
MMU		
Xthal_have_spanning_way	uint8_t	Set if single way maps entire instruction and data virtual space
Xthal_have_identity_map	uint8_t	Virtual = physical flag
Xthal_have_mimic_cacheattr	uint8_t	[Region protection flag]
Xthal_have_xlt_cacheattr	uint8_t	[Region protection with translation flag]
Xthal_have_cacheattr	uint8_t	CACHEATTR register option flag (XEA1)
Xthal_have_tlbs	uint8_t	TLBs option flag (XEA2)
Xthal_mmu_asid_bits	uint8_t	Number of ASID bits
Xthal_mmu_asid_kernel	uint8_t	kernel ASID (always 1)
Xthal_mmu_rings	uint8_t	Number of memory protection rings
Xthal_mmu_ring_bits	uint8_t	Number of rings log2 (round up)
Xthal_mmu_ca_bits	uint8_t	Number of cache attribute encoding bits
Xthal_mmu_max_pte_page_size	uint32_t	Maximum PTE page size (if have_tlbs is set)
Xthal_mmu_min_pte_page_size	uint32_t	Minimum PTE page size (4k) (if have_tlbs is set)
Xthal_itlb_way_bits	uint8_t	Number of instruction TLB ways log2 (up)
Xthal_itlb_ways	uint8_t	Number of instruction TLB ways
Xthal_itlb_arf_ways	uint8_t	Number of instruction TLB autorefill ways
Xthal_dtlb_way_bits	uint8_t	Number of data TLB ways log2 (round up)
Xthal_dtlb_ways	uint8_t	Number of data TLB ways
Xthal_dtlb_arf_ways	uint8_t	Number of data TLB autorefill ways
MPU		
Xthal_mpu_bgmap	custom*	MPU background map custom* Type: const xthal_MPU_entry[ ]
Local Memories and Interfaces		
Xthal_num_instrom Xthal_instrom_vaddr[] Xthal_instrom_paddr[] Xthal_instrom_size[]	uint8_t uint32_t uint32_t uint32_t	Number of instruction ROMs Virtual address at reset of each instruction ROM Physical address of each instruction ROM Size in bytes of each instruction ROM
Xthal_num_instram Xthal_instram_vaddr[] Xthal_instram_paddr[] Xthal_instram_size[]	uint8_t uint32_t uint32_t uint32_t	Number of instruction RAMs Virtual address at reset of each instruction RAM Physical address of each instruction RAM Size in bytes of each instruction RAM

Table 3–11. Link-Time HAL Configuration Constants (continued)

Global Variable	Type	Description
Xthal_num_datarom	uint8_t	Number of data ROMs
Xthal_datarom_vaddr[]	uint32_t	Virtual address at reset of each data ROM
Xthal_datarom_paddr[]	uint32_t	Physical address of each data ROM
Xthal_datarom_size[]	uint32_t	Size in bytes of each data ROM
Xthal_num_dataram	uint8_t	Number of data RAMs
Xthal_dataram_vaddr[]	uint32_t	Virtual address at reset of each data RAM
Xthal_dataram_paddr[]	uint32_t	Physical address of each data RAM
Xthal_dataram_size[]	uint32_t	Size in bytes of each data RAM
Xthal_num_xlmi	uint8_t	Number of XLMI interfaces or apertures
Xthal_xlmi_vaddr[]	uint32_t	Virtual address of each XLMI
Xthal_xlmi_paddr[]	uint32_t	Physical address of each XLMI
Xthal_xlmi_size[]	uint32_t	Size in bytes of each XLMI

As indicated in the *Type* column of Table 3–12, most link-time HAL functions are available, as everything else, in the selected ABI (Windowed ABI or CALL0 ABI): these are marked "std".

Selected functions are also available in an alternative ABI for use by the OS assembler code: these are marked “nw” (non-windowed). Each provides the exact same functionality as the corresponding standard function, and has the same name with an `_nw` suffix appended. Regardless of selected ABI, these “non-windowed” alternative functions are always called using CALL0 (or CALLx0). The calling convention used usually matches the CALL0 ABI, in which case `_nw` suffixed symbols point to the same code as the standard ones if CALL0 ABI is selected. For some functions, a special calling convention is used that is more adapted to OS routines and differs from both CALL0 ABI and Windowed ABI. Such special calling conventions are documented in the relevant function descriptions.

Table 3–12. Link-Time HAL Functions

Item	Type	Description
Coprocessors and Custom State		See Section 3.6 on page 73
xthal_save_extra()	std+nw	Save non-coprocessor (extra) state
xthal_restore_extra()	std+nw	Restore non-coprocessor (extra) state
xthal_save_cpregs()	std+nw	Save coprocessor state
xthal_restore_cpregs()	std+nw	Restore coprocessor state
xthal_validate_cp()	std+nw	Enable specified coprocessor
xthal_invalidate_cp()	std+nw	Disable specified coprocessor
xthal_get_cpenable()	std+nw	Read `CPENABLE` register

Table 3–12. Link-Time HAL Functions (continued)

Item	Type	Description
xthal_set_cpenable()	std+nw	Write `CPENABLE` register
xthal_init_mem_extra()	std	Initialize non-coprocessor (extra) state save area
xthal_init_mem_cp()	std	Initialize coprocessor save area
Interrupts		See Section 3.7 on page 93
xthal_disable_interrupts	std	Disable all interrupts. Individual interrupt enables are not affected.
xthal_enable_interrupts	std	Enable all interrupts. Individual interrupt enables are not affected.
xthal_restore_interrupts	std	Restore interrupt status. Individual interrupt enables are not affected.
xthal_intlevel_get	std	Get current interrupt priority level.
xthal_intlevel_set	std	Set current interrupt priority level.
xthal_intlevel_set_min	std	Set interrupt priority level only if new level is higher than existing level.
xthal_interrupt_pri_get	std	Get specified interrupt's priority level. For XEA2/XEA1 the priority level is constant.
xthal_interrupt_pri_set	std	Set specified interrupt's priority level. Does nothing for XEA2/XEA1.
xthal_interrupt_sens_get	std	Get specified interrupt's sensitivity (edge/level). For XEA3 only.
xthal_interrupt_sens_set	std	Set specified interrupt's sensitivity (edge/level). For XEA3 only.
xthal_interrupt_type	std	Get specified interrupt's type.
xthal_interrupt_enabled	std	Check if specified interrupt is enabled.
xthal_interrupt_pending	std	Check if specified interrupt is pending.
xthal_interrupt_active	std	Check if specified interrupt is active. For XEA3 only.
xthal_interrupt_enable	std	Enable specified interrupt
xthal_interrupt_disable	std	Disable specified interrupt
xthal_interrupt_trigger	std	Trigger specified interrupt
xthal_interrupt_clear	std	Clear specified interrupt
Timers		See Section 3.8 on page 109
xthal_get_ccount()	std	Read `CCOUNT` register
xthal_set_ccompare()	std	Write `CCOMPARE`*n* register
xthal_get_ccompare()	std	Read `CCOMPARE`*n* register

Table 3–12. Link-Time HAL Functions (continued)

Item	Type	Description
MMU		See Section 3.9 on page 111
xthal_static_v2p()	std	Convert virtual to physical addresses across static maps
xthal_static_p2v()	std	Convert physical to virtual addresses across static maps
xthal_v2p()	std	Converts virtual to physical addresses dynamically
xthal_invalidate_region()	std	Invalidates the address translation for region
xthal_set_region_translation_raw()	std	Sets the translation for a region
xthal_set_region_translation()	std	Sets the translation for a region
Windowed Address Registers		See Section 3.10 on page 115
xthal_window_spill()	std+nw	Spill register windows to stack. Note: The non-windowed version of this function does not follow the standard non-windowed ABI. Refer to the function description for details on parameter passing.
Caches		See Section 3.11 on page 117
xthal_get_cacheattr()	std+nw	Read CACHEATTR register or equivalent
xthal_set_cacheattr()	std+nw	Write CACHEATTR register or equivalent
xthal_get_icacheattr()	std+nw	Read instruction cache attributes
xthal_set_icacheattr()	std+nw	Write instruction cache attributes
xthal_get_dcacheattr()	std+nw	Read data cache attributes
xthal_set_dcacheattr()	std+nw	Write data cache attributes
xthal_icache_get_ways()	std+nw	Read instruction cache number of enabled ways
xthal_icache_set_ways()	std+nw	Write instruction cache number of enabled ways
xthal_dcache_get_ways()	std+nw	Read data cache number of enabled ways
xthal_dcache_set_ways()	std+nw	Write data cache number of enabled ways
xthal_icache_all_invalidate()	std+nw	Invalidate entire instruction cache
xthal_dcache_all_invalidate()	std+nw	Invalidate entire data cache (and L2 cache)
xthal_dcache_all_writeback()	std+nw	Writeback entire data cache (and L2 cache)
xthal_dcache_all_writeback_inv()	std+nw	Writeback and invalidate dcache (and L2 cache)
xthal_set_region_attribute()	std	Set cache attributes on a region of memory
xthal_icache_region_invalidate()	std+nw	Invalidate range of addresses from instruction cache
xthal_dcache_region_invalidate()	std+nw	Invalidate range of addresses from data cache (and L2)
xthal_dcache_region_writeback()	std+nw	Writeback range of addr. from data cache (and L2)
xthal_dcache_region_writeback_inv()	std+nw	Writeback and invalidate range of addr. from data cache (and L2 cache)
xthal_icache_all_unlock()	std+nw	Unlock all locked instruction cache lines
xthal_dcache_all_unlock()	std+nw	Unlock all locked data cache lines

Table 3–12. Link-Time HAL Functions (continued)

Item	Type	Description
xthal_icache_region_lock()	std+nw	Prefetch and lock range of bytes into instruction cache
xthal_icache_region_unlock()	std+nw	Unlock range of bytes locked into instruction cache
xthal_dcache_region_lock()	std+nw	Prefetch and lock range of bytes into data cache
xthal_dcache_region_unlock()	std+nw	Unlock range of bytes locked into data cache
xthal_L2_line_lock()	std+nw	Prefetch and lock an L2 cache line
xthal_L2_line_unlock()	std+nw	Unlock an L2 cache line
xthal_L2_region_lock()	std+nw	Prefetch and lock range of bytes into L2 cache
xthal_L2_region_unlock()	std+nw	Unlock range of bytes from L2 cache
xthal_L2_all_unlock()	std+nw	Unlock all locked L2 cache lines
The following functions exist as C macros as well. See Section 3.3.2 for details.		
xthal_icache_line_invalidate()	std+nw	Invalidate instruction cache line
xthal_dcache_line_invalidate()	std+nw	Invalidate data cache (and L2 cache) line
xthal_icache_line_lock()	std+nw	Prefetch and lock an instruction cache line
xthal_icache_line_unlock()	std+nw	Unlock an instruction cache line
xthal_dcache_line_lock()	std+nw	Prefetch and lock a data cache line
xthal_dcache_line_unlock()	std+nw	Unlock a data cache line
xthal_dcache_line_writeback()	std+nw	Writeback data cache (and L2 cache) line
xthal_dcache_line_writeback_inv()	std+nw	Writeback and invalidate data cache (and L2 cache) line
xthal_icache_sync()	std+nw	Await completion of icache operations
xthal_dcache_sync()	std+nw	Await completion of dcache operations
xthal_is_kernel_readable()	std	Returns 1 if the access rights allow read by kernel
xthal_is_kernel_writeable()	std	Returns 1 if the access rights allow write by kernel
xthal_is_kernel_executable()	std	Returns 1 if the access rights allow execute by kernel
xthal_is_user_readable()	std	Returns 1 if the access rights allow read by user
xthal_is_user_writeable()	std	Returns 1 if the access rights allow write by user
xthal_is_user_executable()	std	Returns 1 if the access rights allow execute by user
xthal_encode_memory_type()	std	Encodes the memory type flags into a 9-bit memory type
xthal_is_cached()	std	Returns 1 if memory type is cached
xthal_is_writeback()	std	Returns 1 if memory type is writeback cached
xthal_is_device()	std	Returns 1 if memory type is a device
xthal_read_map()	std	Reads and returns the MPU map
xthal_write_map()	std	Writes the MPU map
xthal_write_map_raw()	std+nw	Writes the MPU map without cache flush
xthal_check_map()	std	Checks the MPU map for correctness

Table 3–12. Link-Time HAL Functions (continued)

Item	Type	Description
xthal_get_entry_for_address()	std	Returns the MPU entry for an address
xthal_mpu_set_region_attribute()	std	Sets MPU attributes for a region of memory
xthal_mpu_set_entry()	std	Sets a single entry of the MPU
Debug		See Section 3.12 on page 154
xthal_set_soft_break()	std	Plant a breakpoint
xthal_remove_soft_break()	std	Remove planted breakpoint
xthal_debugexc_defhndlr_nw	nw	Default debug exception handler
Miscellaneous		
xthal_memcpy()	std	`memcpy()` variant, only uses aligned 32-bit accesses
xthal_clear_regcached_code()	std	Clear code or PCs cached in registers.
Multiprocessing		
xthal_compare_and_set()	std	Multiprocessor synchronization
xthal_get_prid()	std	Read `PRID` register
Valid for XEA2/XEA1 only		
xthal_get_intenable()	std	Read `INTENABLE` register
xthal_set_intenable()	std	Write `INTENABLE` register
xthal_get_interrupt() (was xthal_get_intread())	std	Read `INTERRUPT` register
xthal_set_intset()	std	Set bits of `INTERRUPT` register
xthal_set_intclear()	std	Clear bits of `INTERRUPT` register

3.6 Coprocessor and Custom State

This section describes HAL interfaces that support context-switching custom and optional processor registers. Separate interfaces are provided for registers assigned to coprocessors, and those that are not, because they are typically context-switched differently.

An Xtensa architecture *coprocessor* is simply a set of states whose access is controlled by a bit assigned to that coprocessor in the `CPENABLE` register. When a coprocessor's `CPENABLE` bit is clear (zero), any attempt to execute an instruction that accesses any of that coprocessor's state results in a corresponding coprocessor exception. Operating systems use this mechanism to efficiently implement lazy context switching of coprocessor (grouped) state. The HAL API provided here allows a single operating system port to implement lazy context switching for arbitrary coprocessors and other custom state.

Note: Even though derived from a designer's TIE, the HAL is generated by the Xtensa Processor Generator, not by the TIE compiler. The TIE compiler does not update the HAL header files and libraries. The following parameters only reflect the processor options and extensions provided to the Xtensa Processor Generator, not local changes made using the TIE compiler. If you use these HAL interfaces (for example, in an RTOS) you must rebuild the processor using the Xtensa Processor Generator after any relevant TIE changes in order to get an updated HAL that correctly reflects these changes. Relevant changes are any that affect the existence and properties of registers, or of the instructions used to access them.

The HAL context save and restore functions described in this section are designed to ease the porting of an operating system or runtime (OS) to the fully configurable and extensible Xtensa architecture. This design makes certain assumptions as to which states are handled by these functions, and which ones are the responsibility of the OS. These assignments of responsibility are described in the *Saved By* column of Table 3–13, which classifies all programmer-visible processor states (other than caches and TLBs).

Table 3–13. Processor State Classifications for Saving and Restoring

Processor Registers	Saved By	Used by XCC by Default†	Configuration Option
Address registers			
Current window (`a0` .. `a15`)	OS	Y	—
Outside the current window	OS	Y	Windowed ABI
Non-privileged special registers (0-63)			
`SAR`	OS	Y	—
Loop registers (`LBEGIN`, `LEND`, `LCOUNT`)	OS	Y	Zero-overhead loops
MAC16 registers (`ACCHI`, `ACCLO`)	HAL extra	Y	MAC16
MAC16 registers (`M0`..`M3`)	HAL extra	N	MAC16
Boolean registers (`BR`)	HAL extra	N‡	Boolean register
`SCOMPARE1`	HAL extra	N	Conditional store
`PREFCTL` (usually constant in a program)	none	implicitly	Cache prefetch
`LITBASE` (constant for a given program)	none	Y	Extended L32R
Privileged special registers (64-255)			
`PS.CALLINC` (implicit in `CALL`*n*, `ENTRY`)	OS	Y	Windowed ABI
other fields of `PS`	OS	N	—
`PC` (accessed by OS via `EPC`*n* or `DEPC`)	OS	Y	—
`WB / WINDOWBASE` and `WINDOWSTART`	OS§	Y	Windowed ABI
Other special registers	OS or none††	N	(multiple options)
Custom registers not in any coprocessor			
From user or 3rd party TIE	HAL extra	N	TIE extensions
`THREADPTR`	HAL extra	N	Thread pointer
Custom registers grouped in coprocessors			

Table 3–13. Processor State Classifications for Saving and Restoring (continued)

Processor Registers	Saved By	Used by XCC by Default[†]	Configuration Option
From user or 3rd party TIE	HAL cp	N	TIE extensions
Floating point unit (`F0`..`F15`, `FCR`, `FSR`)	HAL cp	N[‡‡]	FPU
Other TIE packages/coprocessors, such as: Audio engine registers ConnX DSP engine registers ConnX Baseband engine registers	HAL cp	N	 Xtensa HiFi options ConnX D2, *etc.* BBE16, *etc.*

†. More registers may be used when using the `-LNO:simd` flag of XCC with SIMD compatible TIE packages, or if the C code uses TIE intrinsics.

‡. XCC uses these registers for C code that explicitly uses floating point types, when the FPU option is configured.

§. Depending on context and OS design, it is possible to avoid saving and restoring these registers (such as, by spilling register windows to the stack).

††. These are typically either saved/restored by the OS, global in scope, dedicated to a specific context (such as exceptions and interrupts), or unused.

‡‡. XCC uses these registers for C code that explicitly uses floating point types.

Typically, most registers listed in Table 3–13 are context-switched and thus available for use and modification by individual threads (tasks or processes) in a multi-threading environment, except for those listed as *none* in the *Saved By* column.

The HAL functions do not save and restore zero-overhead loop registers, because the OS may need to save them separately so as to clear the `LCOUNT` register early for reasons of protection. They also avoid saving and restoring the `LITBASE` register, which is assumed to have a constant value throughout a program's execution by existing software tools. They also avoid saving and restoring the `SAR` register, because doing so would not hide any configuration option (the `SAR` register is always present), and because `SAR` is often useful early in exception processing such as interrupt dispatch.

In Table 3–13, *custom registers* include TIE user registers (created in TIE using the `user_register` statement) and TIE register files (created using the `regfile` statement). TIE states (created using the `state` statement) are only available for context-switching if mapped to user registers, so they are not considered separately here. Also note that TIE *export states* are never context-switched by HAL routines, because doing so may have side-effects.

Only custom registers can be grouped in coprocessors. Address and special registers are never assigned to a coprocessor.

3.6.1 Handling Interrupts

Interrupt dispatch tends to be highly optimized. Which means saving as few registers as truly necessary. To this end, interrupt handlers are typically restricted as follows:

- Avoid using floating point types (only relevant here if the FPU option is configured).
- Avoid using XCC vectorization options, such as `-LNO:simd`.
- Avoid using the XCC option `-mcoproc`.
- Avoid using intrinsics that might access custom TIE registers and states.

The restrictions generally apply to exception handlers as well. In some cases, the restriction for exception handlers may be relaxed depending upon the specific OS being run.

The restrictions apply both to the handler function, and also to any functions directly or indirectly called by it. If it is necessary for a handler to use floating point, or other coprocessors, then the handler must restore any coprocessor state it alters. The HAL functions described in Section 3.6.4 “Save and Restore Functions” may be used to save and restore the coprocessor state.

In an OS that imposes such restrictions and allows interrupt handlers to be written in C, interrupt dispatch need only save and restore registers that the compiler uses by default. These registers are identified by a Y in the *Used by XCC by Default* column of Table 3–13.

By default, for standard C code (not using floating point types), the compiler never uses coprocessor registers, nor does it use any custom states and registers. These two properties are significant. The first allows an efficient OS implementation to dispatch interrupts without having to either save and restore entire coprocessors, or lazily context-switch such coprocessors in interrupt context (which requires more overhead than doing so in thread context). The second reduces the amount of state that needs to be saved and restored before calling generic C code, such as a simple interrupt handler.

3.6.2 Architectural Constants

XTHAL_MAX_CPS

The Xtensa ISA supports up to eight coprocessors.

```
#define XTHAL_MAX_CPS 8
```

3.6.3 Core-Specific Constants — Presence of Coprocessors

The following values provide summary information on what coprocessors exist (*i.e.*, were configured).

At configuration time, each Xtensa coprocessor is given a unique ID in the range 0 through `XTHAL_MAX_CPS-1` (i.e., 0 to 7). Coprocessor IDs need not be assigned sequentially or contiguously. Thus, a given processor configuration can have a coprocessor 0 and a coprocessor 5 with no intervening coprocessors. In this case, as described below, the number of coprocessors (`XCHAL_CP_NUM` or `Xthal_cp_num`) is 2, the maximum ID boundary (`XCHAL_CP_MAX` or `Xthal_cp_max`) is 6, and the bit mask of configured coprocessors (`XCHAL_CP_MASK` or `Xthal_cp_mask`) is `0x21`.

XCHAL_CP_NUM

This parameter reports the actual number of coprocessors configured.

```
extern const uint8_t Xthal_cp_num;// in libhal.a
```

XCHAL_CP_MAX

This parameter reports the maximum coprocessor ID boundary. This is the greatest coprocessor ID plus one, or zero if there are no coprocessors.

This value may be used to cut down the number of iterations in a loop.

```
extern const uint8_t Xthal_cp_max;
```

XCHAL_CP_MASK

This parameter reports a bit mask of coprocessors that are configured on this processor. Bits zero through seven correspond to coprocessor IDs zero through seven respectively. Each bit is set if a coprocessor with that ID exists (is configured for that processor) and is clear if not.

```
extern const uint32_t Xthal_cp_mask;
```

Example Usage

The following code uses the HAL macro `XCHAL_CP_MAX` to potentially lower the number of iterations of an initialization loop.

```
void init_cps(TCB *tcb)
{
        int i;

        for(i = 0; i < XCHAL_CP_MAX; i++)
        {
             xthal_invalidate_cp(i);
             xthal_init_mem_cp(TCB_CP_SAVE(tcb, i), i);
        }
   }
```

3.6.4 Save and Restore Functions

The way an Xtensa processor's extended and optional state are managed depends largely on whether such state is assigned to a coprocessor. State (registers, *etc.*) that is not in a coprocessor must be saved and restored on each context switch. A collection of state grouped in a coprocessor can be saved and restored lazily by using the `CPENABLE` register (or on every context switch, as determined by the OS).

The following functions save and restore all state that is not in a coprocessor, and state grouped in each coprocessor. They allow an OS to context-switch such state without *a priori* knowledge of the specific state extensions and options configured in a processor.

xthal_save_extra

These routines save most optional and custom processor states not in a coprocessor. Specifically, they save registers listed in Table 3–13 as *HAL extra* in the *Saved By* column.

These registers, named *extra state* for historical reasons, are usually saved and restored on each context switch.

The following routines save this extra state in the area pointed to by `base`. This save area must be aligned to either a 16-byte boundary (always sufficient in existing implementations), or per `XCHAL_NCP_SA_ALIGN` (see Section 3.6.6). And it must consist of at least `XCHAL_NCP_SA_SIZE` bytes.

```
   void xthal_save_extra(void *base);
   void xthal_save_extra_nw(void *base);
```

Depending on implementation, the memory for storage for this data can come from a variety of different places. One obvious candidate is past the end of the current stack pointer. While this is simple for the extra state, this is not an ideal place to put all additional state because of the laziness of coprocessor context switching. The base of the stack, or the task or thread control block (TCB), are probably better selections.

If the caller is using the current stack pointer, the callee must decrement the stack pointer past the window save area in addition to offsetting for the actual space required for the save. Again, using the current stack pointer is not recommended.

The save areas produced by this call do not contain pointers into the area. As a result, the contents of this memory may be safely moved to another memory location as long as that memory also obeys the required alignment restrictions.

Note: While these routines save the state of the extra registers to memory, the values of the registers themselves may be changed by the save operation. The memory is an accurate representation of the registers before the call, but the registers themselves may be changed by the call. This can happen, for example, when there are dependencies between TIE register files. Such a dependency can occur if the only instruction that accesses a TIE register file moves its registers to another register file rather than directly to memory or an address register.

This does not pose a problem during a context switch because the registers are being saved so that a different set of values can promptly be loaded into them. In the case where the designer wishes to save the extra register state to memory and also leave the contents of the registers intact, the designer must follow the save call with the appropriate `xthal_restore_extra` call.

xthal_restore_extra

These routines restore the extra processor state from memory. The layout of this memory is configuration specific and must have been initialized either with the `xthal_save_extra` or `xthal_init_mem_extra` function calls. The base pointer must follow the same alignment restrictions as the base pointer in the `xthal_save_extra` calls.

```
void xthal_restore_extra(void *base);
void xthal_restore_extra_nw(void *base);
```

xthal_save_cpregs

These routines save optional and custom processor states assigned to coprocessors. Specifically, they save registers listed in Table 3–13 as *HAL cp* in the *Saved By* column.

The interface to the routines is almost identical to those for extra state except for the addition of a parameter that specifies the coprocessor to be saved. These functions have no effect if called for a non-existent coprocessor. The save area pointed to by `base` must be aligned per `XCHAL_CP`*n*`_SA_ALIGN` (or `Xthal_cpregs_align[cp]`; see Section 3.6.6), and contain at least `XCHAL_CP`*n*`_SA_SIZE (or Xthal_cpregs_size[cp])` bytes.

```
    void xthal_save_cpregs(void *base, int32_t cp);
    void xthal_save_cpregs_nw(void *base, int32_t cp);
```

Allocation of this memory is more complex. Coprocessors tied to `CPENABLE` can be saved and restored lazily. It is recommended that this memory be allocated either in the TCB, the heap or at the base of the stack. It should not be allocated below the most current stack frame.

Like the `xthal_save_extra` calls, the contents of this memory can be moved as long as the new location still obeys the alignment restrictions.

Note that `CPENABLE` must be set to allow access to the coprocessor for the save and restore functions to work without posting an exception.

Like the extra save routines, the coprocessor save routines may change the contents of the registers as a side effect of saving. The register state stored in memory accurately reflects the state of the coprocessor registers before the save call. However, after the save call, the state of the registers themselves may differ from the state of the registers before the save call. Following the save operation, the designer must issue the appropriate `xthal_restore_cpregs` call if the designer must ensure that contents of the register file are not affected.

xthal_restore_cpregs

Analogous to the `xthal_restore_extra` call, these functions restore the coprocessor state from memory. This layout of this memory is configuration specific and must have been initialized either with the `xthal_save_cpregs` or `xthal_init_mem_cp` calls.

```
    void xthal_restore_cpregs(void *base, int32_t cp);
    void xthal_restore_cpregs_nw(void *base, int32_t cp);
```

Note that `CPENABLE` must be set to allow access to the coprocessor for the restore function to work without posting an exception.

3.6.5 Save and Restore Assembler Macros

xchal_ncp_load

xchal_ncp_store

xchal_cp*n*_load or xchal_cp_*NAME*_load

xchal_cp*n*_store or xchal_cp_*NAME*_store

The use of these macros require including the header:

```
        #include <xtensa/core/tie-asm.h>
```

Assembler macros to save (`xchal_*_store`) or restore (`xchal_*_load`) custom TIE and optional state. Such state is categorized and selectable in several ways. Primarily, by coprocessor:

- Using separate `xchal_cp*` macros for each coprocessor, by number and by name. In Table 3–13 on page 74, states assigned to coprocessors are identified as *HAL cp* in the *Saved By* column.
- Using `xchal_ncp_*` macros for non-coprocessor (extra) state. In Table 3–13 on page 74, such states are identified as *HAL extra* in the *Saved By* column.

Also, each macro supports an optional *`select`* parameter to restrict save or restore to a subset of the registers in the coprocessor (or non-coprocessor) save area. It is specified in one of two ways. Passing `XTHAL_SAS_ALL` selects all registers in the save area, the default if *`select`* is not specified. Passing the `XTHAL_SAS3(optie,cc,abi)` macro selects a subset of the registers along three dimensions, as follows:

1. The `optie` argument selects registers according to whether they are standard options vs. other extensions implemented using TIE. This distinction is only relevant to non-coprocessor save and restore macros. It is an OR combination of the following, or `-1` to include all registers along this dimension:

▪ `XTHAL_SAS_OPT`	Registers that are part of a standard option and not in a coprocessor; includes `BR`, `SCOMPARE1`, `THREADPTR`, and MAC16 registers.
▪ `XTHAL_SAS_TIE`	Other registers (custom extensions, or in a coprocessor).

2. The `ccuse` argument selects registers according to whether they are used automatically by the compiler without any explicit compiler flag or use of the feature in code. (As of this writing, the compiler never uses coprocessor registers by default, but this is expected to change in the future.) It is an OR combination of the following, or `-1` to include all registers along this dimension:

▪ `XTHAL_SAS_CC`	Registers used by default by compiler.
▪ `XTHAL_SAS_NOCC`	Registers not used by default by compiler.

3. The `abi` argument selects registers according to their ABI handling across function calls. It is an OR combination of the following, or `-1` to include all registers along this dimension:

▪ `XTHAL_SAS_CALR`	Caller-saved registers.
▪ `XTHAL_SAS_CALE`	Callee-saved registers.
▪ `XTHAL_SAS_GLOB`	Global across function calls (in thread).

The selected set is the intersection of selections along each of these three dimensions.

Thus, passing `XTHAL_SAS3(-1,-1,-1)` is equivalent to passing `XTHAL_SAS_ALL`, or not passing the *`select`* parameter: all registers in the save area are saved or restored. Note that passing zero to any of the `XTHAL_SAS3()` arguments results in an empty set: nothing gets saved or restored.

WARNING: The `xchal_*_store` macros may clobber the registers after storing them. This happens, in particular, when storing a certain type of register requires a temporary register of another type (both types necessarily in the same coprocessor or both in a non-coprocessor state). Such registers can be defined using the TIE `regfile` and `proto` constructs, described in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. In this case, registers of the type used as temporaries get stored first; then, those of the earlier type get stored, clobbering the temporary registers.

Synopsis:

```
xchal_*_load  ptr at1 at2 at3 at4 [continue] [ofs] [select] [alloc]
xchal_*_store ptr at1 at2 at3 at4 [continue] [ofs] [select] [alloc]
```

Parameters:

▪ *`ptr`*	Address register that points to the save area to load from or store to. Required parameter. This register is clobbered by the macro; however, it can be restored using the `xchal_sa_ptr_restore` macro (see below). **Note:** the register must contain an address aligned according to the coprocessor requirements:

XCHAL_NCP_SA_ALIGN byte aligned for a non-coprocessor state, or XCHAL_CP<*n*>_SA_ALIGN byte aligned for coprocessor *n*.

- *at1*, *at2*, *at3*, *at4* — Four temporary address registers (first XCHAL_NCP_NUM_ATMPS registers are clobbered, the remaining are unused). Required parameters.
- *continue* — If the macro is invoked as part of a larger load or store sequence, set to 1 if this is not the first in the sequence. Defaults to 0.
  Note: Assembler symbols are used to keep track of how much was written to the save area, and how much the original *ptr* has advanced; they are reset if *continue* is 0.
- *ofs* — Offset from the start of larger sequence (from value of first *ptr* in sequence) at which to load or store. Defaults to next available space (to 0 if *continue* is 0).
- *select* — Select what category(ies) of registers to load or store, as a bitmask (see above). Defaults to all registers (XTHAL_SAS_ALL).
- *alloc* — Select what category(ies) of registers to allocate. If any category is allocated here that is not in *select*, space for the corresponding registers is skipped without doing any access (load or store) to that space, rather than packing subsequent selected registers into that space.

Examples:

Save the state of coprocessor 3 to save area at a2, using a4-a7 as temporaries:

```
xchal_cp3_store a2, a4,a5,a6,a7
```

Save the state of the HiFi2 DSP coprocessor, in a similar manner:

```
xchal_cp_AudioEngineLX_store a2, a4,a5,a6,a7
```

Save all non-coprocessor and coprocessor states to a contiguous save area at *a2*:

```
xchal_cp0_store a2, a4,a5,a6,a7, continue=0
xchal_cp1_store a2, a4,a5,a6,a7, continue=1
xchal_cp2_store a2, a4,a5,a6,a7, continue=1
xchal_cp3_store a2, a4,a5,a6,a7, continue=1
xchal_cp4_store a2, a4,a5,a6,a7, continue=1
xchal_cp5_store a2, a4,a5,a6,a7, continue=1
xchal_cp6_store a2, a4,a5,a6,a7, continue=1
xchal_cp7_store a2, a4,a5,a6,a7, continue=1
```

```
    xchal_ncp_store a2, a4,a5,a6,a7, continue=1
```

Restore only callee-saved non-coprocessor states, from a3:

```
    xchal_ncp_load a3, a6,a7,a8,a9 select=XTHAL_SAS3(-1,-1,XTHAL_SAS_CALE)
```

Save only caller-saved non-coprocessor states used by default by the compiler, allocating space for both caller-saved and callee-saved (but not global) registers, used or not by default by the compiler:

```
    xchal_ncp_store a3, a6,a7,a8,a9                \
              select=XTHAL_SAS3(-1,XTHAL_SAS_CC,XTHAL_SAS_CALR)    \
              alloc=XTHAL_SAS3(-1,-1,XTHAL_SAS_CALR|XTHAL_SAS_CALE)
```

xchal_sa_ptr_restore

Restore the *ptr* register after invoking one of the `xchal_*_load` or `xchal_*_store` macros described above. These macros track how much the *ptr* register was incremented, using an assembler symbol. So `xchal_sa_ptr_restore` simply expands to an `ADDI` instruction if the *ptr* register did get clobbered (incremented).

Example:

```
    xchal_ncp_load a2, a4,a5,a6,a7      // restore NCP state from a2,
                                        //  clobbering a2
    xchal_sa_ptr_restore a2             // restore a2
```

xchal_atmps_store

xchal_atmps_load

Save and restore any temporary address registers needed by save and restore macros, that are not already available for use. May be used to optimize save and restore sequences, by only saving and restoring as many temporary address registers as necessary, when less than four are otherwise available.

Synopsis:

```
    xchal_atmps_store  ptr offset ntmps [[[at1] at2] at3] at4
    xchal_atmps_load   ptr offset ntmps [[[at1] at2] at3] at4
```

The assumption is that the first (0 to 3) of the four temporary address registers (used for the save and restore macros) are already saved somewhere and thus available. The remaining 1 to 4 address registers are passed to `xchal_atmps_store` and `xchal_atmps_load`, which save and restore whichever of them are needed.

Parameters:

▪ *`ptr`*, *`offset`*	Address register and constant offset pointing to where to save the temporary address registers `at`*`n`*. There must be 4 bytes of space for each `at`*`n`* parameter passed.
▪ *`ntmps`*	Total number of temporary address registers needed by the save/restore macro(s). Normally one of the `XCHAL_*_NUM_ATMPS` constants.
▪ *`at1`*, *`at2`*, *`at3`*, *`at4`*	Address registers to potentially save and restore. From one to four address registers must be specified, and they must be the last one-to-four registers given as temporary registers to the save/restore macros (in the same order).

Example:

We call a save/restore macro (`xchal_ncp_load`) using `a0`, `a3`, `a4`, `a5` as temporary registers. We've already saved `a0` earlier (not shown here), so `a0` must be the first temporary register. We use `xchal_atmps_store` and `xchal_atmps_load` to save and restore only as many of the remaining temporary registers (`a3`, `a4`, `a5`) as needed. There is available space to save 3 address registers (12 bytes) at `a1+SOMEOFS`.

```
xchal_atmps_store a1, SOMEOFS, XCHAL_NCP_NUM_ATMPS, a3, a4, a5
xchal_ncp_load    a2, a0,a3,a4,a5
xchal_atmps_load  a1, SOMEOFS, XCHAL_NCP_NUM_ATMPS, a3, a4, a5
```

3.6.6 Core Specific Constants

XCHAL_NCP_SA_SIZE

`XCHAL_NCP_SA_SIZE` gives the required size of the non-coprocessor ("extra") save area in bytes. Note that the save area may contain gaps, due to alignment and non-power-of-2 sized registers for example. Thus, the size of the save area may be larger than the sum of its components' sizes. Note that this size does not include extra space required for initial pointer alignment.

```
extern const uint32_t Xthal_extra_size;
```

XCHAL_NCP_SA_ALIGN

`XCHAL_NCP_SA_ALIGN` gives the required alignment of the non-coprocessor save area in bytes. This value is always a power of 2, and is at least 1. It is a function of the largest alignment required to load or store registers in this save area. Note that in processors configured with a load/store interface wider than 128 bits, alignment can be greater than 16 bytes (even though the ABI stack alignment is still 16 bytes for most functions).

```
extern const uint32_t Xthal_extra_align;
```

XCHAL_NCP_NUM_ATMPS

`XCHAL_NCP_NUM_ATMPS` is the number of temporary address registers required by the `xchal_ncp_load` and `xchal_ncp_store` assembler macros. It ranges from 0 to 4.

XCHAL_CP*n*_SA_SIZE and Xthal_cpregs_size

`XCHAL_CP`*n*`_SA_SIZE` gives the required size of the save area for coprocessor *n*. As with `XCHAL_NCP_SA_SIZE`, the save area may contain gaps, and the indicated size is not padded or rounded up for alignment.

The `Xthal_cpregs_size` array gives save area sizes for each of the coprocessors. For example, `Xthal_cpregs_size[0]` is set to `XCHAL_CP0_SA_SIZE`: it reports the save area size for coprocessor zero.

```
extern const uint32_t Xthal_cpregs_size[XTHAL_MAX_CPS];
```

XCHAL_CP*n*_SA_ALIGN and Xthal_cpregs_align

`XCHAL_CP`*n*`_SA_ALIGN` gives the required save area alignment for coprocessor *n*, in bytes. As with `XCHAL_NCP_SA_ALIGN`, its value is always a power of 2, is at least 1, and may be greater than 16 (bytes) for processors configured with a load/store interface wider than 128 bits. It is a function of the largest alignment required to load or store registers in this save area.

The `Xthal_cpregs_align` array gives the required alignment of the save area for each of the coprocessors. For example, `Xthal_cpregs_align[0]` is set to `XCHAL_CP0_SA_ALIGN`: it reports the save area alignment for coprocessor zero.

```
extern const uint32_t Xthal_cpregs_align[XTHAL_MAX_CPS];
```

XCHAL_CP*n*_NUM_ATMPS

`XCHAL_CP`*`n`*`_NUM_ATMPS` is the number of temporary address registers required by the `xchal_cp`*`n`*`_load` and `xchal_cp`*`n`*`_store` assembler macros for coprocessor *n*. This number ranges from 0 to 4.

XCHAL_TOTAL_SA_SIZE

`XCHAL_TOTAL_SA_SIZE` gives the sum of all save areas defined above, with some alignment padding included. Specifically, it is the sum of `XCHAL_NCP_SA_SIZE` and of `XCHAL_CP`*`n`*`_SA_SIZE` for each coprocessor, each rounded up to the worst-case alignment (or to 16 bytes, whichever is higher) required of *any* save area.

WARNING: It is up to the caller to save and restore state to/from each save area. Thus, the exact layout and size of such a collection of save areas is up to the user of this API. The size indicated by this constant may or may not be sufficient to contain it; it is provided as a convenience for simple layouts. If you are unsure, given the definition above, that this size is sufficient, you may need to compute a particular layout's size using the individual save area sizes and alignment requirements.

```
    extern const uint32_t Xthal_all_extra_size;
```

XCHAL_TOTAL_SA_ALIGN

`XCHAL_TOTAL_SA_ALIGN` is the highest required alignment of all save areas defined above. Specifically, it is the maximum of `XCHAL_NCP_SA_ALIGN` and of `XCHAL_CP`*`n`*`_SA_ALIGN` for each coprocessor. Its value is always a power of 2, is at least 1, and may be greater than 16 (bytes) for processors configured with a load/store interface wider than 128 bits.

```
    extern const uint32_t Xthal_all_extra_align;
```

XCHAL_SA_NUM_ATMPS

`XCHAL_SA_NUM_ATMPS` is the maximum number of temporary address registers required by any save/restore macro. It is MAX(`XCHAL_NCP_NUM_ATMPS`, `XCHAL_CP`*`n`*`_NUM_ATMPS`) for all *n*. This number ranges from 0 to 4.

Example Usage

The caller is responsible for laying out the save areas in the stack. Often this code will be done in assembly during a context switch. The following code is an example of saving all of the processor's additional state (both non-coprocessor and coprocessor state) into the stack. The code ensures that the base of the stack remains 16-byte aligned.

```
        //  This macro is invoked for each save area.
        .macro    savestate      size, align, func
        .if \size
        movi      a15, \size          // size of save area
        movi      a11, 0xfffffff0 & -\align     // align. mask
        sub       a14, a14, a15       // allocate on base of stack
        and       a14, a14, a11       // align as required (at least 16)
        mov       a2, a14             // 1st arg: where to save state
        call0     \func               // save state to [a2]
        .endif
        .endm

        //  Sequence starts here.
        movi      a2, cur_tcb_ptr     // task control block ptr address
        l32i      a2, a2, 0           // task control block ptr
        l32i      a14, a2, TCB_SP_BASE_OFFSET // save at base of stack
        //  Here, a14 points to end of stack. Allocate growing down.

        //  Save non-coprocessor state.
   savestate XCHAL_NCP_SA_SIZE, XCHAL_NCP_SA_ALIGN, xthal_save_extra_nw

        //  Save state of all coprocessors.
   savestate XCHAL_CP0_SA_SIZE, XCHAL_CP0_SA_ALIGN, xthal_save_cp0_nw
   savestate XCHAL_CP1_SA_SIZE, XCHAL_CP1_SA_ALIGN, xthal_save_cp1_nw
   savestate XCHAL_CP2_SA_SIZE, XCHAL_CP2_SA_ALIGN, xthal_save_cp2_nw
   savestate XCHAL_CP3_SA_SIZE, XCHAL_CP3_SA_ALIGN, xthal_save_cp3_nw
   savestate XCHAL_CP4_SA_SIZE, XCHAL_CP4_SA_ALIGN, xthal_save_cp4_nw
   savestate XCHAL_CP5_SA_SIZE, XCHAL_CP5_SA_ALIGN, xthal_save_cp5_nw
   savestate XCHAL_CP6_SA_SIZE, XCHAL_CP6_SA_ALIGN, xthal_save_cp6_nw
   savestate XCHAL_CP7_SA_SIZE, XCHAL_CP7_SA_ALIGN, xthal_save_cp7_nw
```

This is an inefficient implementation of TIE coprocessor switching because all coprocessors are saved and restored on every context switch. Implementing lazy switching under the `CPENABLE` exceptions avoids this work for coprocessors that the task does not use. Nevertheless, the code shown above and below serves for illustration purposes.

C code to save all of the state to a block of `malloc()`ed memory might look like this:

```
   // Align to 'align', with minimum 16-byte alignment:
   #define ALIGN(ptr,align) ( ((int)(ptr) + (((align)-1) | 15)) \
                             & -(align) & -16 )

   void *save_all(int in_size)
   {
        char *p = malloc(XCHAL_TOTAL_SA_SIZE + XCHAL_TOTAL_SA_ALIGN - 1);
        char *ptr = p;
        int cp;
        ptr = (char*) ALIGN(ptr, XCHAL_NCP_SA_ALIGN);
        xthal_save_extra( ptr );
        ptr += XCHAL_NCP_SA_SIZE;
        for( cp = 0 ; cp < XTHAL_MAX_CPS; cp++ )
        {
             ptr = (char*) ALIGN(ptr, Xthal_cpregs_align[cp]);
             xthal_save_cpregs( ptr, cp );
             ptr += Xthal_cpregs_size[cp];
        }

        return base;
   }
```

The caller is able to set its own layout. In this case, the storage layout is shown in Table 3–14 (with possible alignment padding between each area, not shown).

Table 3–14. Storage Layout

Storage Layout
(Low Address)
Non-coprocessor state
Coprocessor 0 state
Coprocessor 1 state
. . .
Coprocessor *N*-1 state
(High Address)

3.6.7 Coprocessor Enable and Disable Functions

Each coprocessor has a bit in the `CPENABLE` register indicating whether or not access to any of the coprocessor’s states and register files is allowed. If the bit is 0, executing an instruction that accesses the coprocessor causes an exception. If the bit is 1, such accesses are allowed and do not cause a coprocessor exception.

In general, all routines in the link-time HAL accept any parameters that could be valid in some valid configuration. For example, the validate and invalidate calls below accept a coprocessor ID of 7 even if the processor is not configured with coprocessor ID 7. In such cases the routines return “safe” values and perform no operation, as appropriate.

xthal_validate_cp

These routines set a bit in the `CPENABLE` register to indicate that a particular coprocessor can be accessed. After this call, the specified coprocessor's registers can be accessed without causing a coprocessor exception. Note that if the specified coprocessor does not exist, the function returns harmlessly.

```
void xthal_validate_cp(int32_t cp);// enable specified coprocessor.
void xthal_validate_cp_nw(int32_t cp);// non windowed version.
```

xthal_invalidate_cp

These routines clear a bit in the `CPENABLE` register to indicate that a particular coprocessor cannot be accessed. After this call, accesses to the specified coprocessor's registers cause a coprocessor exception. Note that if the specified coprocessor does not exist, the function returns harmlessly.

```
void xthal_invalidate_cp(int32_t cp);   // disable specified
coprocessor.
void xthal_invalidate_cp_nw(int32_t cp);// non windowed version
```

xthal_get_cpenable

This function returns the value of the `CPENABLE` special register. If the register does not exist, it returns zero.

```
uint32_t xthal_get_cpenable( void );    // read CPENABLE register
```

xthal_set_cpenable

This function sets the `CPENABLE` special register. If the register does not exist, it has no effect.

```
void xthal_set_cpenable( uint32_t );    // write CPENABLE register
```

Example Usage

The C code in the previous section assumed that the coprocessors were available for use when calling the save routine. The following version of the `save_all` function no longer makes this assumption. It sets the coprocessors as valid before accessing them.

```
void *save_all(int in_size)
{
     char *p = malloc(XCHAL_TOTAL_SA_SIZE + XCHAL_TOTAL_SA_ALIGN - 1);
     char *ptr = p;
```

```
        int cp;

        ptr = (char*) ALIGN(ptr, XCHAL_NCP_SA_ALIGN);
        xthal_save_extra(ptr);
        ptr += XCHAL_NCP_SA_SIZE;
        for(cp = 0; cp < XTHAL_MAX_CPS; cp++)
        {
             xthal_validate_cp(cp);
             ptr = (char*) ALIGN(ptr, Xthal_cpregs_align[cp]);
             xthal_save_cpregs(ptr, cp);
             ptr += Xthal_cpregs_size[cp];
        }

        return base;
   }
```

3.6.8 Configuration Dependent State Save Area Initialization

The following functions initialize a non-coprocessor state and a coprocessor state save areas in memory to valid default values. You can always safely restore the relevant state from a save area initialized this way. This is useful at system startup to put everything in a known state. It is particularly useful when creating or initializing a task, to set the initial value of this state used by that task without involving actual registers or state (until they are used for the first time by that task).

At this time, initialization merely zeros the contents of the save areas. However, it is safer to use these functions to initialize save areas to allow for the future possibility of coprocessor or non-coprocessor states for which an all-zeroed save area does not represent a valid state.

xthal_init_mem_extra

This function initializes the non-coprocessor save area memory with valid initial values. After restoring a save area initialized with `xthal_init_mem_extra()`, the non-coprocessor state is set to a safe and ready-to-use condition.

```
   void xthal_init_mem_extra(void *base);  // initialize non-coproc state
```

The base pointer must be aligned according to `XCHAL_NCP_SA_ALIGN` (or `Xthal_extra_align`).

Note: In a multithreading OS, it is sometimes desirable to use a non-coprocessor state as initialized by an early startup sequence, as the default values used by each thread. (For example, to get a consistent default value of the `PREFCTL` register, if cache prefetch is configured.) In this case, you may find it preferable to save a non-coprocessor state in a global save area just before starting multithreading, to capture this default value of a non-coprocessor state. Then, when creating a thread, its non-coprocessor save area can be copied from that global one rather than calling `xthal_init_mem_extra()`.

xthal_init_mem_cp

This call is analogous to the `xthal_init_mem_extra` calls except that it initializes a coprocessor state rather than a non-coprocessor state.

```
void xthal_init_mem_cp(void *base, int32_t cp);// initialize a
coprocessor
```

Example Usage

The following C code allocates and initializes all save areas for a task:

```
void init_task_optional_state(TCB *tcb)
{
    int i;
    tcb->extra_ptr = malloc_align( Xthal_extra_size, Xthal_extra_align );
    xthal_init_mem_extra( tcb->extra_ptr );
    for(i = 0; i < XTHAL_MAX_CPS; i++)
    {
       tcb->cp_ptr[i] = malloc_align( Xthal_cpregs_size[i], Xthal_cpregs_align[i] );
       xthal_init_mem_cp( tcb->cp_ptr[i], i );
    }
}
```

The following C code initializes all optional state:

```
void init_optional_state( void )
{
        int i;
        void *ptr = malloc_align( Xthal_extra_size, Xthal_extra_align );
        xthal_init_mem_extra( ptr );
        xthal_restore_extra( ptr );
        free( ptr );
```

```
for(i = 0; i < XTHAL_MAX_CPS; i++)
{
        xthal_validate_cp(i);
        ptr = malloc_align( Xthal_cpregs_size[i], Xthal_cpregs_align[i] );
        xthal_init_mem_cp( ptr, i );
        xthal_restore_cpregs( ptr, i );
        free( ptr );
        xthal_invalidate_cp(i);     /* depending on usage */
}
```

3.7 Interrupts

In general, the interrupt section of the HAL provides configuration-specific information about a processor's interrupts and provides basic manipulation of the architecture interrupt registers.

For XEA3, data entries are restricted to the number of interrupts and interrupt levels actually configured.

For XEA2 and XEA1, when the designer has configured less than 32 interrupts (or less than 15 levels), the link-time HAL still supports data entries for these unconfigured resources. The designer may depend on these labels being in the link-time HAL independent of the presence of the actual item in the configuration.

3.7.1 Architectural Constants

These constants are fixed by the Xtensa ISA; they are not configuration dependent. Hence they are defined as preprocessor constants rather than global variables.

Other constants are also described under "`XCHAL_INT`*`n_TYPE and Xthal_inttype`*" in Section 3.7.2.

XTHAL_MAX_INTERRUPTS

This define is the architectural limit to the number of interrupts in the processor.

For XEA3, up to 256 interrupts can be configured.

For XEA2/XEA1, up to 32 interrupts can be configured.

```
#define XTHAL_MAX_INTERRUPTS 32 /* max number of interrupts (0..255)
*/
```

XTHAL_MAX_INTLEVELS

This define is the maximum number of interrupt levels that an Xtensa processor can have. Note that this is the ISA-defined maximum. The actual maximum allowed by the implementation and enforced by the Xtensa Processor Generator may be less than this.

For XEA2/XEA1, only seven levels are allowed (levels zero thru six).

```
#define XTHAL_MAX_INTLEVELS 16 /* max interrupt levels (0..15) */
```

3.7.2 Core Specific Constants

XCHAL_NUM_INTLEVELS

This constant contains the highest interrupt level configured in the Xtensa processor.

```
extern const uint8_t Xthal_num_intlevels;
```

XCHAL_NUM_INTERRUPTS

This variable has the number of interrupts actually configured in the processor. Interrupts are numbered from zero to `XCHAL_NUM_INTERRUPTS-1`.

```
extern const uint8_t Xthal_num_interrupts;
```

XCHAL_INTLEVEL*n*_MASK and Xthal_intlevel_mask

For XEA2/XEA1 only.

Bitmask of interrupts configured at level *n*.

Knowledge of all interrupts at a given interrupt level allows for easy masking of an entire priority level of interrupts. This can be quite important, especially with level-one interrupts. The link-time HAL defines an array of words, one for each interrupt level. Each bit in the word is a one if the interrupt is in the level and is a zero if the interrupt is not in the level. `Xthal_intlevel_mask[`*n*`] == XCHAL_INTLEVEL`*n*`_MASK`.

```
extern const uint32_t Xthal_intlevel_mask[XTHAL_MAX_INTLEVELS];
```

The index to this array is the interrupt level. The value at index zero is always zero.

XCHAL_INTLEVEL*n*_ANDBELOW_MASK and Xthal_intlevel_andbelow_mask

For XEA2/XEA1 only.

Bitmask of interrupts configured at level *n* or below.

Turning off all interrupts of a given level and below is often a useful operation. To aid in this, the HAL supplies a second array of interrupt masks, indexed by interrupt level, indicating which bits are in that level and all levels below it. `Xthal_intlevel_andbelow_mask[n] == XCHAL_INTLEVELn_ANDBELOW_MASK`.

```
extern const uint32_t
Xthal_intlevel_andbelow_mask[XTHAL_MAX_INTLEVELS];
```

The value at index zero is always zero. The value at index `XTHAL_MAX_INTLEVELS–1` is always the set of all interrupts, i.e., `(1<<Xthal_num_interrupts)–1`, by definition.

XCHAL_INT*n*_LEVEL and Xthal_intlevel

Level of interrupt *n*.

Each interrupt has a level. The `Xthal_intlevel` array encodes the interrupt level for each interrupt number. Indexing this array with the number of the interrupt gives the level of that interrupt.

```
extern const uint8_t Xthal_intlevel[XTHAL_MAX_INTERRUPTS];
```

XCHAL_INT*n*_TYPE and Xthal_inttype

Type of interrupt *n* (one of `XTHAL_INTTYPE_<type>`, below).

Each interrupt has a type. The `Xthal_inttype` array encodes the interrupt type for each interrupt number. Indexing this array with the number of the interrupt gives the type of that interrupt.

```
extern const uint8_t Xthal_inttype[XTHAL_MAX_INTERRUPTS];
```

The entries in this array will each be one of the following constants:

▪ `XTHAL_INTTYPE_UNCONFIGURED`	Unconfigured interrupt. Only array entries with indices `Xthal_num_interrupts` through `XTHAL_MAX_INTERRUPTS–1` (inclusive) are set to this value.
▪ `XTHAL_INTTYPE_SOFTWARE`	Software interrupt.
▪ `XTHAL_INTTYPE_EXTERN_EDGE`	External edge-triggered interrupt.

`XTHAL_INTTYPE_EXTERN_LEVEL`	External level-triggered interrupt.
`XTHAL_INTTYPE_TIMER`	Timer interrupt.
`XTHAL_INTTYPE_NMI`	External non-maskable interrupt.
`XTHAL_INTTYPE_WRITE_ERROR`	Write error interrupt.
`XTHAL_INTTYPE_PROFILING`	Internal interrupt used for profiling.
`XTHAL_INTTYPE_IDMA_DONE`	Integrated DMA completion interrupt.
`XTHAL_INTTYPE_IDMA_ERR`	Integrated DMA error interrupt.
`XTHAL_INTTYPE_GS_ERR`	Gather/Scatter (Super-Gather) interrupt.
`XTHAL_INTTYPE_L2_ERR`	Level-2 cache event interrupt.

Additionally, the following constant is also defined:

`XTHAL_MAX_INTTYPES`	Number of interrupt types. The above constants are assigned small numbers in the range 0 through `XTHAL_MAX_INTTYPES–1` inclusive.

XCHAL_INTTYPE_MASK_<*type*> and Xthal_inttype_mask

These exist for XEA2 and XEA1 only.

The `Xthal_inttype_mask[]` array gives bit masks of which interrupts are configured for each interrupt type. This allows the designer to easily get the set of all interrupts of a given type. The index to this array is one of the `XTHAL_INTTYPE_<`*`type`*`>` constants defined above.

```
extern const uint32_t Xthal_inttype_mask[XTHAL_MAX_INTTYPES];
```

This information is also available in the following macros:

`XTHAL_INTTYPE_MASK_UNCONFIGURED`	Interrupts past `XCHAL_NUM_INTERRUPTS`.
`XTHAL_INTTYPE_MASK_SOFTWARE`	Software interrupts.
`XTHAL_INTTYPE_MASK_EXTERN_EDGE`	External edge-triggered interrupts.
`XTHAL_INTTYPE_MASK_EXTERN_LEVEL`	External level-triggered interrupts.
`XTHAL_INTTYPE_MASK_TIMER`	Timer interrupts.
`XTHAL_INTTYPE_MASK_NMI`	External non-maskable interrupts.

XCHAL_EXTINT*n*_NUM

Reports the interrupt number corresponding to *external interrupt n* (`BInterrupt` pin *n*).

Of the many Xtensa processor interrupt types listed above, only the edge-triggered, level-triggered, and NMI interrupts are visible on the processor's external interrupt signals (`BInterrupt`). This signal group only contains bits for external interrupts, not for internal ones; and its numbering is zero based and contiguous. Thus, interrupt numbers as visible on the external `BInterrupt` wires *differ* from the programmer-visible interrupt numbers. Each `XCHAL_EXTINT`*n*`_NUM` macro converts external (`BInterrupt`) interrupt number *n* to the corresponding internal (programmer-visible) interrupt number.

3.7.3 Functions

xthal_disable_interrupts

This routine disables all interrupts except NMI. It returns the current state of the interrupt disable. Note that this routine does not affect the enable status of individual interrupts. Instead it manipulates the `PS` register to accomplish its goal.

```
static inline uint32_t xthal_disable_interrupts(void);
```

xthal_enable_interrupts

This routine enables all interrupts. It returns the current state of the interrupt disable. Note that this routine does not affect the enable status of individual interrupts. Instead it manipulates the `PS` register to accomplish its goal.

```
static inline uint32_t xthal_enable_interrupts(void);
```

xthal_restore_interrupts

This routine restores previous interrupt status using the provided flag, which should be the return value from a previous call to `xthal_disable_interrupts()`. It does not return anything. Note that this routine does not affect the enable status of individual interrupts. Instead it manipulates the `PS` register to accomplish its goal.

```
static inline void xthal_restore_interrupts(uint32_t);
```

xthal_intlevel_get

This routine returns the current interrupt priority level in effect.

For XEA3, the priority level is read from the interrupt controller.

For XEA2/XEA1, the priority level is read from the `PS` register.

```
static inline uint32_t xthal_intlevel_get(void);
```

xthal_intlevel_set

This routine sets a new interrupt priority level.

For XEA3, the priority level is written to the interrupt controller.

For XEA2/XEA1, the priority level is written to the `PS` register. It returns the existing priority level. Be aware that calling this function could result in the priority level being lowered, and an interrupt being taken immediately.

```
static inline uint32_t xthal_intlevel_set(uint32_t);
```

xthal_intlevel_set_min

This routine sets a new interrupt priority level, but only if the new level is higher than the existing priority level. It therefore guarantees that the interrupt priority level will never be lowered as a result of calling this function. The function returns the existing interrupt priority level.

```
static inline uint32_t xthal_intlevel_set_min(uint32_t);
```

Note: The above interrupt functions may not be safe to use when running with an operating system or other runtime. Use the functionality provided by the OS/runtime instead.

xthal_interrupt_pri_get

This routine returns the priority level of the specified interrupt.

For XEA3, the priority level may be programmable and is read from the interrupt controller.

For XEA2/XEA1, the priority level is fixed and is read from the `Xthal_intlevel[]` array.

```
static inline uint32_t xthal_interrupt_pri_get(uint32_t);
```

xthal_interrupt_pri_set

This routine sets the priority level of the specified interrupt.

For XEA3, the priority level may be programmable and is set at the interrupt controller.

For XEA2/XEA1, the priority level is fixed so this function does nothing.

```
static inline void xthal_interrupt_pri_set(uint32_t, uint8_t);
```

xthal_interrupt_sens_get

This routine returns the sensitivity type (edge or level) of the specified interrupt.

It is applicable for XEA3 only, and returns zero for edge and nonzero for level.

For XEA2/XEA1, the function does nothing and returns 0.

```
static inline uint32_t  xthal_interrupt_sens_get(uint32_t);
```

xthal_interrupt_sens_set

This routine sets the sensitivity type (edge or level) of the specified interrupt.

It is applicable for XEA3 only, and returns nothing.

For XEA2/XEA1, the function does nothing and returns nothing.

```
static inline void xthal_interrupt_sens_set(uint32_t, uint8_t);
```

xthal_interrupt_type

This routine returns the type of the specified interrupt. It simply reads the `Xthal_int-type[]` array.

```
static inline uint32_t xthal_interrupt_type(uint32_t);
```

xthal_interrupt_enabled

This routine returns true if the specified interrupt is enabled.

```
static inline uint32_t xthal_interrupt_enabled(uint32_t);
```

xthal_interrupt_pending

This routine returns true if the specified interrupt is pending.

```
static inline uint32_t xthal_interrupt_pending(uint32_t);
```

xthal_interrupt_active

This routine returns true if the specified interrupt is active.

It is applicable for XEA3 only.

For XEA2/XEA1, it does nothing and returns 0.

```
static inline uint32_t xthal_interrupt_active(uint32_t);
```

xthal_interrupt_enable

This routine enables the specified interrupt. It does not return anything.

```
static inline void xthal_interrupt_enable(uint32_t intnum);
```

This function is not safe to use in runtimes (or operating systems or kernels) that manage level-masking or software prioritization of interrupts using the `INTENABLE` register, such as in XTOS. Such environments typically provide their own functions to enable and disable individual interrupts. For example, XTOS provides the functions `_xtos_interrupt_enable()` and `_xtos_interrupt_disable()`.

xthal_interrupt_disable

This routine disables the specified interrupt. It does not return anything.

```
static inline void xthal_interrupt_disable(uint32_t intnum);
```

This function is not safe to use in runtimes (or operating systems or kernels) that manage level-masking or software prioritization of interrupts using the `INTENABLE` register, such as in XTOS. Such environments typically provide their own functions to enable and disable individual interrupts. For example, XTOS provides the functions `_xtos_interrupt_enable()` and `_xtos_interrupt_disable()`.

xthal_interrupt_trigger

This routine triggers (sets) the specified interrupt. It does not return anything.

```
static inline void xthal_interrupt_trigger(unint32_t intnum);
```

xthal_interrupt_clear

This routine clears the specified interrupt. It does not return anything.

```
static inline void xthal_interrupt_clear(uint32_t intnum);
```

xthal_get_intenable

The routine returns the current value of the `INTENABLE` register. Each bit in this register corresponds to an interrupt, in order from the least-significant bit for interrupts numbered from zero.

In assembly, this register is read with the `RSR` instruction.

```
static inline uint32_t  xthal_get_intenable(void);
```

xthal_set_intenable

This routine is deprecated for enabling or disabling interrupts. Use the functions `xthal_interrupt_enable()` or `xthal_interrupt_disable()` instead. This routine sets the value of the `INTENABLE` register. It is important to remember that the `INTENABLE` register cannot be read and set in a single atomic operation. As a result, manipulation of the `INTENABLE` should only be performed when the processor interrupt level is at a safe value.

```
static inline void       xthal_set_intenable(uint32_t);
```

In assembly, this register is written with the `WSR` instruction.

This function is not safe to use in runtimes (or operating systems or kernels) that manage level-masking or software prioritization of interrupts using the `INTENABLE` register, such as in XTOS. Such environments typically provide their own functions to enable and disable individual interrupts. For example, XTOS provides `_xtos_ints_on()` and `_xtos_ints_off()`, which must be used instead of `xthal_set_intenable()`.

xthal_get_interrupt

The routine reads the value of the `INTERRUPT` register and returns it to the caller.

```
static inline uint32_t xthal_get_interrupt(void);
```

This function is performed in assembly with an `RSR` of the `INTERRUPT` register.

Note: This function was named `xthal_get_intread()` in T1050 and prior releases. The old name is still available for backward compatibility only.

xthal_set_intset

This function is deprecated for triggering interrupts. Use `xthal_interrupt_trigger()` instead. This function sets certain bits in the `INTERRUPT` register. Bits that are set in the passed value are set in the interrupt register. Note that not all interrupt types allow the value in the `INTERRUPT` register to be set in this way.

```
static inline void      xthal_set_intset(uint32_t);
```

This function is performed in assembly with a `WSR` of the `INTSET` special register.

xthal_set_intclear

This function is deprecated for clearing interrupts. Use `xthal_interrupt_clear()` instead. This function clears certain bits in the `INTERRUPT` register. Bits that are set in the passed value will be cleared in the register. Note that not all interrupt types allow the value in the `INTERRUPT` register to be cleared in this way.

```
static inline void void      xthal_set_intclear(uint32_t);
```

This function is performed in assembly with a `WSR` of the `INTCLEAR` special register.

3.7.4 Interrupt Trampolines--For XEA2/XEA1 Only

Note: This discussion pertains to XEA2/XEA1 only. For XEA3, interrupt trampolining is not likely to be required.

Interrupt trampolining refers to the technique of invoking a lower-priority interrupt from a higher-priority one, under control of the high-priority interrupt handler software. This is typically accomplished using a software interrupt, although other means of doing the same thing are also possible—*e.g.*, using an external interrupt if the external hardware provides a means for software to trigger that interrupt.

Trampolines are useful and often required when writing high-priority interrupt handlers, because:

- High-priority interrupt handlers often cannot execute most if any OS calls, because at the point where they are taken, the state of the OS is generally indeterminate. Normally, an OS only disables level-one and medium-priority interrupts in its critical sections, to avoid increasing the latency of high-priority interrupts whose sole purpose and design is to execute with minimal latency.
- High-priority interrupt handlers nevertheless need a way to signal events to tasks and other OS entities. Such signaling can usually be done only at interrupt level zero (all interrupts enabled), level one, or at medium-priority levels.

- High-priority interrupt handlers cannot execute C code without excessive context save/restore overhead[4] (because they can interrupt register window spills and fills) which is counter to their fast, low-latency design. But they may sometimes need to trigger an event that should be handled using C code.

High-priority interrupt handlers are usually written in assembler, use non-windowed calling conventions, and `save/restore` all registers and any other processor state that they use. A typical high-priority interrupt handler will save minimal state, process the interrupt quickly, clear it, restore state and return. For example, see Section 2.6.6 and Section 2.6.7.

For some applications, a given interrupt may be an independent activity that needs little or no communication with the rest of the software—i.e., it can process the interrupt completely and return without explicitly interacting with the rest of the software. But more frequently, the handler will need to communicate or otherwise interact with other parts of the system software (tasks, message queues, etc.).

In some cases, the handler can communicate with the rest of the software (*e.g.*, tasks, *etc*.) by simply setting data accessible via some global variables. If so, software will pick it up asynchronously, either by polling or examining this data upon some other appropriate events. If this is insufficient, the handler may need to signal some event to the rest of the software in an event-driven fashion (e.g., wake up a task, queue a message, or other action that may initiate processing at interrupt level 0 through EXCM Level). Usually the high-priority interrupt handler will try to avoid doing this every time it is invoked, for performance reasons—*e.g.*, it may need to signal such an event only when a buffer is filled or emptied, or when a particular condition is detected. Because of the strict limitations imposed on high-priority interrupt handlers as described above, the usual way to do this is to “trampoline” down to a level-one or medium-priority interrupt that is able to execute the required action.

3.7.5 Asynchronous Trampolines--For XEA2/XEA1 Only

A high-priority interrupt handler generally initiates a trampoline by simply triggering a level-one or medium-priority interrupt (*e.g.,* a software interrupt) which we’ll refer to as the *trampoline interrupt*. The high-priority handler then clears the high-priority interrupt condition and returns as usual.

If multiple high-priority interrupts (or multiple signaling conditions) share the same trampoline interrupt, some state needs to be written atomically before triggering the trampoline interrupt.

4. With the CALL0 ABI, it is possible to dispatch high-priority interrupts to C code without excessive overhead. This is because the CALL0 ABI involves less processor states and does not impose the possibility of exceptions, as opposed to the Windowed ABI where window, alloca, and syscall exceptions may occur. It is even possible, in a Windowed ABI application, to handle high-priority interrupts using the CALL0 ABI, as long as the interrupt handler and application do not share any code. However, existing runtimes (such as XTOS) do not support such scenarios without modifications.

The trampoline interrupt handler can pick up this state to determine which event(s) to process. (If all high-priority interrupts are at the same level, atomicity is trivial; if not, either a different variable must be used for each level, or interrupts must be masked at the highest level sharing the trampoline interrupt when updating the event bit mask for instance, to avoid losing events due to race-conditions among the various high-priority interrupts.)

Note that the high-priority interrupt remains enabled while trampolining occurs. This is very useful when some processing must occur with low latency, but the rest of the processing can occur at lower priority.

Note also that the trampoline interrupt handler will not necessarily execute as soon as the high-priority interrupt handler returns. Other high-priority interrupts, possibly nested, as well as other level-one and medium-priority interrupts and exceptions and OS critical sections that disable them, may delay execution of the trampoline interrupt.

Here is an example trampoline implementation. The example code presented is for a hypothetical device and thus has not been tested: it is for illustration purposes only.

An unbuffered input device generates an interrupt every time a character is received. The interrupt handler must retrieve the character from the device (at address `CHARDEV_IOADDR`) before the next character comes in. The characters and corresponding interrupts come in at a very high rate so the device's interrupt signal is connected to a high-priority interrupt (interrupt number `CHARDEV_INTNUM` at level two) to ensure it can always be serviced with minimal latency. This assumes of course that higher-level interrupts, if any are configured and used, will not add enough latency to the level-two interrupts to interfere with servicing of the character device.

Characters are double-buffered in software. When a buffer is full, application software is signaled using a level-one software interrupt (number `CHARDEV_LEVEL1SW_INTNUM`), *i.e.* the trampoline interrupt. While the application software processes this buffer, the high-priority handler continues retrieving characters from the device into the alternate buffer.

Here is the level-two interrupt handler. It assumes it starts at the level-two interrupt vector, that the vector was configured large enough to contain this handler, and that no other interrupts were configured at level-two.

```
#include <xtensa/coreasm.h>

     // Global variables used by the level-2 interrupt handler.
     // These are place in a structure to reduce need for registers
     // and increase performance.
     //
     .struct 0
pbd_start:.space 4// ptr to start of buffer descriptor table
pbd_end:  .space 4// ptr to end (last+1) of buffer desc. table
pbd_cur:  .space 4// ptr to current buffer descriptor entry
```

```
bufnext:  .space 4// next to write in current buffer, 0 if drop
bufend:   .space 4// end of current buffer
a3save:   .space 4// saved a3
a4save:   .space 4// saved a4
L2VARS_SIZE:

     // Buffer descriptor structure.
     //
     .struct 0
PBD_BUFPTR:.space 4 // ptr to buffer
PBD_BUFSIZE:.space 4// size of buffer in bytes
PBD_FULL: .space 4  // 1=full (owned by application),
                    // 0=not full (owned by L2 handler)
PBD_SIZE:

     .commLevel2Vars, L2VARS_SIZE, 4

     .begin         literal_prefix.Level2InterruptVector
     .section       .Level2InterruptVector.text, “ax”

     .globlLevel2Vector
Level2Vector:
     wsr  a2, EXCSAVE_2  // save a2
     movi a2, Level2Vars
     s32i a3, a2, a3save // save a3
     s32i a4, a2, a4save // save a4
     l32i a4, a2, bufnext
     movi a3, CHARDEV_IOADDR
     beqz a4, getnext    // no current buf? get one
                         // (drops 1st char, for now)
     l8ui a3, a3, 0      // retrieve character
     s8i  a3, a4, 0      // save character in buffer
     l32i a3, a2, bufend
     addi a4, a4, 4
     bgeu a4, a3, full   // buffer full? give it
     s32i a4, a2, bufnext// update bufnext

done:// Exit the handler.

     // We must clear the source of the high-priority interrupt
     // before returning. How this is done depends on the type of
     // interrupt. For now let’s assume this high-priority interrupt
     // is edge-triggered:
     //
     movi a3, (1<<CHARDEV_INTNUM)
     wsr  a3, INTCLEAR

     // Then restore state and return:
     l32i a3, a2, a3save // restore a3
     l32i a4, a2, a4save // restore a4
     rsr  a2, EXCSAVE_2  // restore a2
     rfi  2              // done

full://  We have filled an input buffer.  Give it to the application
     //  for processing:  mark it as full, and initiate trampoline.
     //
     l32i a3, a2, pbd_cur
     movi a4, 1
     s32i a4, a3, PBD_FULL

     // Now signal the application that it has a buffer to process.
     // We do this by triggering the “trampoline interrupt”, that is,
     // by triggering a level-one software interrupt:
     //
     movi a4, (1<<CHARDEV_LEVEL1SW_INTNUM)
     wsr  a4, INTSET

     //  Advance to next buffer descriptor:
     l32i a4, a2, pbd_end
     addi a3, a3, PBD_SIZE
     bltu a3, a4, 1f
     l32i a3, a2, pbd_start
1:   s32i a3, a2, pbd_cur

getnext:// Get next buffer from application, if any available:
     l32i a3, a2, pbd_cur
     l32i a4, a3, PBD_FULL// is it available?
     bnez a4, nobuf      // if not, error
     // Current buffer now available, start receiving into it:
     l32i a4, a3, PBD_BUFPTR
     l32i a3, a3, PBD_BUFSIZE
     s32i a4, a2, bufnext
     add  a3, a3, a4
     s32i a3, a2, bufend
     j    done

nobuf:// The application did not keep up; it did not return a buffer
     // to this handler fast enough to avoid dropping characters.
     // We could do some special handling here if required.
     // For this example, just clear bufnext to drop characters
     // until the application provides a buffer to this handler.
     //
     movi a4, 0
     s32i a4, a2, bufnext// note we’re dropping chars
     j    done

     .end literal_prefix
```

Here is the C portion of the application, responsible for initialization and processing of incoming characters a full buffer at a time. Real-time operating system, device, or application-specific sequences are indicated as such between ellipses ("...").

```
typedef struct BDesc {
   char*    bufptr;
   unsigned bufsize;
   volatile unsigned full;
} BDesc;

typedef struct L2Vars {
   BDesc* pbd_start;// ptr to start of buffer descriptor table
   BDesc* pbd_end;// ptr to end (last+1) of buffer desc. table
   BDesc* pbd_cur;// ptr to high-priority handler's current desc.
   char*  bufnext;// next in current buf, 0 if dropping chars
   char*  bufend;// end of current buffer
   unsigned a3save;// saved a3
   unsigned a4save;// saved a4
} L2Vars;

extern L2Vars Level2Vars;// global variables used by level-2 handler

#define BUFSIZE 512
static char bufs[2][BUFSIZE];
static BDesc bds[2];
static BDesc* pbd_next;/* next buffer to process */

/*
 * Level-one software interrupt handler (trampoline int. handler).
 * This is registered to be invoked on occurrences of interrupt
 * number CHARDEV_LEVEL1SW_INTNUM in an OS/runtime specific manner.
 */
void trampoline_handler( void )
{
  /*
   *  Clear software interrupt *before* processing, to avoid losing
   *  trampolines by the level-two handler while we're processing:
   */
  xthal_set_intclear( 1<<CHARDEV_LEVEL1SW_INTNUM );

  /*  Either process data on the spot, or wake a task to do so:  */
  if( ...designer chooses to process data in the level-one handler... )
    process_buffers();
  else {
    ...wake up or signal task that will call process_buffers()...
    ...(this is done in an OS-specific manner)...
  }
}
```

```
/*
 *  Process incoming buffers. This is high latency work, and is done
 *  in a level-one interrupt handler or in a task rather than in the
 *  level-two interrupt handler.
 */
void process_buffers( void )
{
  while( pbd_next->full ) {
    char *ptr = pbd_next->bufptr;
    int  size = pbd_next->bufsize;

    ... process data in ptr[0 .. size-1] ...

    pbd_next->full = 0;/* processed, give back to level-2 handler */
    if( ++pbd_next >= Level2Vars.pbd_end )
      pbd_next = Level2Vars.pbd_start;
  }
}

void init_buffers( void )
{
  int i;
  for( i = 0; i < 2; i++ ) {
    bds[i].bufptr = bufs[i];
    bds[i].bufsize = BUFSIZE;
    bds[i].full = 0;
  }
  Level2Vars.pbd_start = bds;
  Level2Vars.pbd_end = bds + 2;
  Level2Vars.pbd_cur = bds;
  Level2Vars.bufnext = bufs[0];
  Level2Vars.bufend  = bufs[0] + BUFSIZE;
  pbd_next = bds;

  /*
   * Register trampoline handler to interrupt CHARDEV_LEVEL1SW_INTNUM.
   * E.g. using the Tensilica basic run-time (XTOS), this would be:
   *  _xtos_set_interrupt_handler( CHARDEV_LEVEL1SW_INTNUM,
   *                               &trampoline_handler );
   */
  ... OS-specific ...

  /*
   * Initialize device to start recv. chars and generating interrupts:
   */
  ... device-specific ...
}
```

3.8 Timers

The Xtensa processor generator allows the designer to configure up to three timers. Each "timer" is a comparator (`CCOMPARE`*n* register) that triggers an interrupt when it matches a common free-running counter (`CCOUNT` register). The HAL indicates whether a given timer comparator is configured and to what interrupt it is tied.

Note: The `CCOUNT` register is part of the Xtensa processor core. It stops counting during core power shut-off (see the *Xtensa LX Microprocessor Data Book*) and when the processor is stopped in OCD Mode (see the *OCD Software Tools Development* chapter of the *Xtensa Debug Guide*).

3.8.1 Architectural Constants

XTHAL_MAX_TIMERS

This architectural constant defines the maximum number of timers (comparators) that may be configured. The actual maximum allowed by the implementation and enforced by the Xtensa Processor Generator is three timers. However, for historical reasons, as the ISA was designed to allow up to four timers, this constant is reported as 4.

```
#define XTHAL_MAX_TIMERS 4
```

3.8.2 Core Specific Constants

XCHAL_HAVE_CCOUNT

This global constant reports whether the timer option is configured. The `CCOUNT` special register is present with the timer option.

```
extern const uint8_t Xthal_have_ccount;
```

XCHAL_NUM_TIMERS

This global constant reports the number of timers, or more precisely of `CCOMPARE`*n* special registers, configured in the processor. Possible values are `0` to `XTHAL_MAX_TIMERS`. A non-zero value implies the timer option is configured (*i.e.,* that `XCHAL_HAVE_CCOUNT` is also non-zero).

```
extern const uint8_t Xthal_num_ccompare;
```

XCHAL_TIMER*n*_INTERRUPT and Xthal_timer_interrupt

The XCHAL_TIMER*n*_INTERRUPT constant and `Xthal_timer_interrupt[`*n*`]` array entries report what interrupt timer *n* is tied to. Array index *n* is a value between zero and `XTHAL_MAX_TIMERS`–1.

```
extern const int32_t Xthal_timer_interrupt[XTHAL_MAX_TIMERS];
```

Valid values for each entry in this array are `0..255` for XEA3, `0..31` for XEA2/XEA1, and `–1`. In the case where a timer is not configured, the value is –1. Otherwise, the value is the interrupt to which the timer is tied. Timers are ordered sequentially. Therefore, if two timers are configured, they are represented in entries zero and one of the array.

3.8.3 Functions

xthal_get_ccount

This function returns the current value of the `CCOUNT` register. If the register does not exist (*i.e.*, timer option is not configured), it always returns zero.

```
static inline uint32_t xthal_get_ccount(void);
```

xthal_set_ccompare

This function sets the value of the specified `CCOMPARE`*n* register. Parameter *n* ranges from zero to `XTHAL_MAX_TIMERS–1` (*i.e.,* zero to three). Note that when handling a timer interrupt, setting the corresponding `CCOMPARE`*n* register clears the interrupt. Setting a non-existent `CCOMPARE` register has no effect.

```
extern void xthal_set_ccompare(int32_t n, uint32_t value);
```

xthal_get_ccompare

This function gets the value of a particular `CCOMPARE`*n* register. Parameter *n* ranges from zero to `XTHAL_MAX_TIMERS–1` (*i.e.,* zero to three). Reading a non-existent `CCOMPARE` register returns zero.

```
extern uint32_t xthal_get_ccompare(int32_t n);
```

3.9 Memory Management

xthal_static_v2p

This function converts a virtual address to a physical address.

For processors configured with XEA1, with region protection, or with MMU v3 (RC release or later, which resets in a state similar to region protection), no translation is done. That is, the physical address returned is the same as the virtual address provided.

If an MMU v1 or v2 (RB release or earlier) is configured and the virtual address does not correspond to one of the static maps, this function returns -1. Otherwise, it returns the physical address via the `paddrp` argument and returns 0 to indicate success.

```
extern int32_t xthal_static_v2p (uint32_t vaddr, uint32_t *paddrp);
```

xthal_static_p2v

This function converts a physical address to a virtual address.

For processors configured with XEA1, with region protection, or with MMU v3 (RC release or later, which resets in a state similar to region protection), no translation is done. That is, the virtual address returned is the same as the physical address provided.

If an MMU v1 or v2 (RB release or earlier) is configured and the physical address does not correspond to one of the static maps, this function returns -1. Otherwise, it returns the virtual address via the `vaddrp` argument and returns 0 to indicate success.

Parameter `cached` is a boolean argument. If more than one virtual address maps to the physical address provided, a non-zero value for this argument causes the function to prefer virtual addresses in cached regions to those in bypass regions. A zero value for `cached` causes the function to prefer bypass regions over cached regions.

```
extern int32_t xthal_static_p2v (uint32_t paddr,
                                 uint32_t *vaddrp,
                                 uint32_t cached);
```

xthal_v2p

This function dynamically converts a virtual address to a physical address by probing the state of the data TLB. If there is a translation for the supplied virtual address, then the physical address is returned via the paddrp argument. If the `wayp` argument is a non-zero value, then the TLB way used for the translation is returned via the `wayp` argument. If the `cattrp` argument is a non-zero value, then the cache attribute for the `vaddr` argument is returned via the `cattrp` argument. On processors configured with an MPU, the value returned in `cattrp` consists of four bits[3:0] of access rights and nine bits [12:4] of memory type. That value is suitable for passing directly as the `cattr` parameter to the `xthal_set_region_attribute()` function.

If there is a translation for the virtual address, then the function returns 0. If there is no translation for the virtual address, then the function returns `XTHAL_NO_MAPPING`.

```
extern int32_t xthal_v2p (void* vaddr,
                          void **paddrp,
                          uint32_t* wayp,
                          uint32_t* cattrp);
```

xthal_invalidate_region

This function removes the translation for the region beginning at `vaddr` from the TLB. This function is only supported on processors configured with the MMU v3.

If this function is successful it returns 0. If the supplied address is not aligned to the start of a region, then this function returns `XTHAL_INVALID_ADDRESS`. If this function is called on an unsupported processor, it returns `XTHAL_UNSUPPORTED_ON_THIS_ARCH`.

```
extern int32_t xthal_invalidate_region (void* vaddr);
```

xthal_set_region_translation_raw

This function establishes a translation from the virtual address region starting at `vaddr` to the physical address region starting at `paddr`. The cache attribute for the region is set to `cattr`.

This functions assumes that `vaddr` and `paddr` are aligned to the start of a 512 MB region. It also assumes that its caller commits all cache writes in the region before calling it, and that all cache entries for that region are invalidated.

Calling this function on the region where instructions are currently being fetched is undefined.

This function is only supported on processors configured with the region translation option or the MMU v3. If this function is successful it returns 0. If this function is not supported on the processor configuration it returns `XTHAL_UNSUPPORTED_ON_THIS_ARCH`.

```
extern int32_t xthal_set_region_translation_raw (void* vaddr,
                                                 void* paddr,
                                                 uint32_t cattr);
```

xthal_set_region_translation

This function sets the translation of the region of virtual memory specified by `vaddr` and `size` to the region of physical memory beginning at paddr. This function is only supported on processors configured with the region translation option or the MMU v3. This function automatically writes back and invalidates relevant cache entries to ensure memory coherency; see below for details and Section 3.11.8 on page 146 for general coverage of the coherency issues.

Calling this function on the region where instructions are currently being fetched is undefined.

```
extern int32_t xthal_set_region_translation (void* vaddr,
                                              void* paddr,
                                              uint32_t size,
                                              uint32_t cattr,
                                              uint32_t flags);
```

If the specified memory range exactly covers a series of consecutive 512 MB regions, the translation and cache attributes of these regions are updated. If this is not the case, *e.g.*, if either or both the start and end of the range only partially cover a 512 MB region, one of three results are possible:

- By default, the translation and attribute of all regions covered, even just partially, is updated.
- If the `XTHAL_CAFLAG_EXACT` flag is specified, an non-zero error code is returned.
- If the `XTHAL_CAFLAG_NO_PARTIAL` flag is specified (but not the `EXACT` flag), only regions fully covered by the specified range are updated.

Table 3–15 and Table 3–16 describe the function parameters and return values, respectively.

Table 3–15. `xthal_set_region_translation` Parameters

Parameter	Description
`vaddr`	Starting virtual address of region of memory
`paddr`	Starting physical address of region of memory
`size`	Number of bytes in the region of memory

Table 3–15. `xthal_set_region_translation` Parameters (continued)

Parameter	Description
`cattr`	Cache attribute (encoded), typically one of the following compile-time HAL constants: `XCHAL_CA_BYPASS` cache disabled (bypassed) `XCHAL_CA_BYPASSBUF` cache disabled, writes bufferable `XCHAL_CA_WRITEBACK` cache enabled, write-back mode (allocate on store) `XCHAL_CA_WRITEBACK_NOALLOC` cache enabled, write-back mode (no allocate on store) `XCHAL_CA_ILLEGAL` no access (load, store, fetch) allowed (exception) For more details on the meaning of these cache attributes, see the *Xtensa LX Microprocessor Data Book,* specifically in the *Memory-Management Model Types* chapter, *Region Protection* section, and *Access Modes* subsection. To use these constants, include `<xtensa/config/core.h>`. On processors configured with the MMU v3, the cache attribute can be bitwise combined with the following rights flags: `XCHAL_CA_R` region can only be read. `XCHAL_CA_RX` region can be read and instructions can be executed from it. `XCHAL_CA_RW` region can be read and written. `XCHAL_CA_RWX` region can be read, written, and instructions can be executed from it. This is the default.
`flags`	Bitwise combination of the following flags (0 if none), defined in `<xtensa/hal.h>`: `XTHAL_CAFLAG_EXACT` return error if the requested translation cannot be applied to the exact range specified `XTHAL_CAFLAG_NO_PARTIAL` only apply the requested translation to regions or pages completely covered by the specified range; do not modify regions/pages only partially covered `XTHAL_CAFLAG_NO_AUTO_WB` do not automatically writeback dirty data. `XTHAL_CAFLAG_NO_AUTO_INV` do not automatically invalidate cache.

Table 3–16. **`xthal_set_region_translation` return values**

Return Value	Description
`0`	Successful, or size is zero
`XTHAL_NO_REGIONS_COVERED`	`XTHAL_CAFLAG_NO_PARTIAL` flag specified and address range is valid with a non-zero size, however no 512 MB region (or page) is completely covered by the range
`XTHAL_INEXACT`	`XTHAL_CAFLAG_EXACT` flag specified, and address range does not exactly specify a set of 512 MB regions (or pages)
`XTHAL_ADDRESS_MISALIGNED`	The translation failed because vaddr and paddr are not aligned. The offset within the region of vaddr and paddr must be equal.
`XTHAL_UNSUPPORTED`	Function not supported in this processor configuration

When caches are writeback caches, this function automatically writes back dirty data when changing the translation of a region that is in writeback mode. This writeback is done safely, that is, by first switching to writethrough mode, then invoking `xthal_dcache_all_writeback`(), then changing the region translation. Such a sequence is necessary to ensure there is no longer any dirty data in the memory region by the time this function returns, even in the presence of interrupts, speculation, etc. The `XTHAL_CAFLAG_NO_AUTO_WB` flag disables this automatic writeback behavior.

This function automatically invalidates any cache entries for the specified region. For performance reasons, invalidation may be done across the entire cache instead of just the affected region(s) of memory. The `XTHAL_CAFLAG_NO_AUTO_INV` flag disables the automatic cache invalidation.

3.10 Register Windows

In general, the register window handlers are configuration independent. The size of the register file is configurable.

The link-time HAL includes a window-spilling routine. This routine spills the current contents of the register window back into the memory stack. This routine is provided for convenience since it is a relatively complex and tricky operation.

3.10.1 Core Specific Constants

XCHAL_NUM_AREGS

This value has the number of physical address registers in the `ar` register file.

```
extern const uint32_t Xthal_num_aregs;
```

XCHAL_NUM_AREGS_LOG2

This value has the base 2 logarithm of the number of physical address registers in the `ar` register file. The number of physical address registers is always a power of 2, hence `XCHAL_NUM_AREGS == (1 << XCHAL_NUM_AREGS_LOG2)`.

```
extern const uint8_t Xthal_num_aregs_log2;
```

3.10.2 Functions

xthal_window_spill

This function spills live register windows to the stack. It will spill all register windows except its own window, and possibly that of its caller. (Currently, the caller's window is spilled then reloaded when this function returns. This may change with future optimizations.)

```
extern void xthal_window_spill(void);
```

xthal_window_spill_nw

This function is equivalent to `xthal_window_spill()` but is meant to be called by low-level assembler code, such as an exception handler. Since XEA3 allows exception and interrupt handlers to be written in C, this function is not required to be used. It does **not** follow the standard non-windowed calling conventions (CALL0 ABI). Instead it requires the following calling convention.

On Entry:

```
PS.WOE = 0
PS.INTLEVEL >= XCHAL_EXCM_LEVEL
a0 = return PC
a1 = valid stack pointer
a4 - a15 must be valid if they are part of the windows to spill.
WINDOWSTART[WINDOWBASE] = 1
LCOUNT = 0
```

On Exit:

```
PS.WOE unchanged
PS.INTLEVEL unchanged
WINDOWBASE unchanged
WINDOWSTART changed per spilled windows (equals 1<<WINDOWBASE if
successful)
a0 unchanged
a1 unchanged
a2 = return code (see below)
a3 changed
a4, a5, a8, a9, a12, a13 unchanged
a6, a7, a10, a11, a14, a15 changed if they were part of windows to
spill
SAR changed
LCOUNT unchanged
```

Possible values for return (in `a2`) are:

- 0: call was successful
- 1: failure ((1 << `WINDOWBASE`) & `WINDOWSTART`) == 0
- 2: failure (invalid `WINDOWSTART`)

3.11 Cache

The Xtensa ISA specifies separate D (data) and I (instruction) caches, conventionally named “dcache” and “icache” respectively. These caches are internal, or Level-1 (L1). Multiple levels of external caches may exist, that are not managed by the functions provided here. However, some Xtensa cores may be configured with an integrated Level-2 (L2) cache managed by this API. Manipulation of the caches is separate for instruction and data caches, that is, with separate constants and functions for each cache. Data cache operations are generally applied to integrated level-2 caches as well (where indicated), but instruction cache operations are not.

The link-time HAL for cache is presented as four discrete portions: organization, global control, per-region control, and per-line control. These are described in the next subsections. All functions have an equivalent non-windowed entry point.

3.11.1 Core Specific Constants

The following parameters describe the physical properties and sizes (organization) of the instruction and data caches.

`XCHAL_ICACHE_LINESIZE`	Size of instruction cache line in bytes; or fetch width if there is no instruction cache.
`XCHAL_DCACHE_LINESIZE`	Size of data cache line in bytes; or load/store width if there is no data cache.
`XCHAL_ICACHE_LINEWIDTH`	log2(`XCHAL_ICACHE_LINESIZE`)
`XCHAL_DCACHE_LINEWIDTH`	log2(`XCHAL_DCACHE_LINESIZE`)
`XCHAL_ICACHE_SETWIDTH`	log2(icache lines per way), 0 if no icache
`XCHAL_DCACHE_SETWIDTH`	log2(dcache lines per way), 0 if no dcache
`XCHAL_ICACHE_WAYS`	Inst. cache set associativity (num. of ways)
`XCHAL_ICACHE_WAYS_LOG2`	log2(`XCHAL_ICACHE_WAYS`)
`XCHAL_DCACHE_WAYS`	Data cache set associativity (num. of ways)
`XCHAL_DCACHE_WAYS_LOG2`	log2(`XCHAL_DCACHE_WAYS`)
`XCHAL_L1SCACHE_WAYS`	L1S cache set associativity (num. of ways)
`XCHAL_L1SCACHE_WAYS_LOG2`	log2(`XCHAL_L1SCACHE_WAYS`)
`XCHAL_ICACHE_SIZE`	Inst. cache size in bytes, or 0 if no icache
`XCHAL_ICACHE_SIZE_LOG2`	log2(`XCHAL_ICACHE_SIZE`), or 0 if no icache
`XCHAL_DCACHE_SIZE`	Data cache size in bytes, or 0 if no dcache
`XCHAL_DCACHE_SIZE_LOG2`	log2(`XCHAL_DCACHE_SIZE`), or 0 if no dcache
`XCHAL_L1SCACHE_SIZE`	L1S cache size in bytes, or 0 if no L1S cache
`XCHAL_L1SCACHE_SIZE_LOG2`	log2(`XCHAL_L1SCACHE_SIZE`), or 0 if no L1S cache
`XCHAL_DCACHE_IS_WRITEBACK`	Set if data cache supports writeback
`XCHAL_DCACHE_IS_COHERENT`	Set if data cache supports MP coherence
`XCHAL_HAVE_PREFETCH`	Set if cache prefetch option is configured
`XCHAL_HAVE_PREFETCH_L1`	Set if cache prefetch to L1 is configured
`XCHAL_PREFETCH_CASTOUT_LINES`	Number of lines in the prefetch cast-out buffer
`XCHAL_HAVE_CACHE_BLOCKOPS`	Set if cache block prefetch option is configured

`XCHAL_PREFETCH_ENTRIES`	Number of configured prefetch entries.
`XCHAL_PREFETCH_BLOCK_ENTRIES`	Number of configured block prefetch entries, *i.e.*, the number of block operations that can execute in parallel.
`XCHAL_HAVE_ICACHE_DYN_WAYS`	Set if inst. cache has dynamic way support
`XCHAL_HAVE_DCACHE_DYN_WAYS`	Set if data cache has dynamic way support
`XCHAL_ICACHE_LINE_LOCKABLE`	Set if inst. cache supports per-line locking
`XCHAL_DCACHE_LINE_LOCKABLE`	Set if data cache supports per-line locking
`XCHAL_ICACHE_ECC_PARITY`	0 or `XTHAL_MEMEP_{PARITY, ECC}`
`XCHAL_DCACHE_ECC_PARITY`	0 or `XTHAL_MEMEP_{PARITY, ECC}`
`XCHAL_ICACHE_ACCESS_SIZE`	Inst. cache access size in bytes (for SICW)
`XCHAL_DCACHE_ACCESS_SIZE`	Data cache access size in bytes
`XCHAL_L1SCACHE_ACCESS_SIZE`	L1S cache access size in bytes
`XCHAL_L1SCACHE_BANKS`	Number of L1S cache banks
`XCHAL_HAVE_L2`	Set if integrated L2 cache is configured
`XCHAL_L2_SIZE`	Total size of integrated L2 memory in bytes (shared between L2RAM and L2 cache)
`XCHAL_L2CACHE_LINESIZE`	Size of integrated L2 cache line in bytes; undefined if there is no integrated L2 cache.
`XCHAL_L2CACHE_LINEWIDTH`	log2(`XCHAL_L2CACHE_LINESIZE`)
`XCHAL_L2CACHE_WAYS`	L2 Cache set associativity (num of ways)
`XCHAL_L2CACHE_LOCKABLE`	Set if L2 cache supports per-line locking

The link-time HAL defines the following corresponding variables:

```
extern const uint8_t Xthal_icache_linewidth;// size of icache line in bytes log2

extern const uint8_tXthal_dcache_linewidth;// size of dcache line in bytes log2

extern const uint8_t Xthal_icache_linesize;// size of icache line in bytes

extern const uint8_t Xthal_dcache_linesize;// size of dcache line in bytes

extern const uint8_t Xthal_icache_setwidth;// number of icache sets log2

extern const uint8_t Xthal_dcache_setwidth;// number of dcache sets log2

extern const uint32_t  Xthal_icache_ways;// icache set associativity

extern const uint32_t  Xthal_dcache_ways; // dcache set associativity

extern const uint32_t  Xthal_icache_size;// size of the icache in bytes

extern const uint32_t  Xthal_dcache_size; // size of the dcache in bytes
extern const uint8_t Xthal_dcache_is_writeback; // dcache writeback option flag

extern const uint8_t  Xthal_icache_line_lockable; // icache line locking option

extern const uint8_t  Xthal_dcache_line_lockable; // dcache line locking option

extern uint32_t Xthal_L2cache_size;// size of L2 cache (configured at initialization
time)

extern uint32_t Xthal_L2ram_size;// size of L2RAM
```

Note that there is some redundancy in these constants, for convenience. For example, the cache size constants can be computed as: `ways * 2`$^{\texttt{setwidth + linewidth}}$.

3.11.2 Global Cache Control Functions

Global control operations apply to an entire cache.

XTHAL_L2_SETUP

```
    #include <xtensa/core-macros.h>
    XTHAL_L2_SETUP(ram_paddr, ram_fraction, cache_fraction);
```

The `XTHAL_L2_SETUP()` macro sets global symbols used by the reset vector to setup the integrated L2 cache controller, and by cache functions that operate on the L2 cache.

This macro is NOT invoked as a function call. Rather, it must be invoked outside any function a single time, as it only sets symbols. This approach allows correct configuration of the L2 cache controller early in the reset vector, for better performance.

The integrated L2 controller's memory can be used as either an L2 cache, as an external system memory (L2RAM), or partitioned as some combination of both. The `XTHAL_L2_SETUP()` macro configures this partitioning, as well as the physical address of L2RAM. The `XTHAL_L2_SIZE` constant reports the total size of L2 memory in bytes.

Table 3–17. `XTHAL_L2_SETUP` Parameters

Parameter	Description
`ram_paddr`	Physical address of L2RAM. Alignment must satisfy constraints of the L2 controller (see the *Xtensa Microprocessor Data Book* for details). To preserve the default reset value of this address, use `XCHAL_L2RAM_RESET_PADDR`. If `XTHAL_L2_SETUP()` is not invoked, default is dependent on startup code (for XTOS, it is to `XCHAL_L2RAM_RESET_PADDR`).
`ram_fraction`	Fraction of L2 memory used as L2RAM, in sixteenths. Must be a decimal constant (not a constant expression) from 0 to 16. If the L2RAM does not support the requested fraction, the closest smaller supported value is used. If `XTHAL_L2_SETUP()` is not invoked, default is dependent on startup code (for XTOS, it is 8 -- half L2RAM, half L2 cache).

Table 3–17. `XTHAL_L2_SETUP` Parameters (continued)

Parameter	**Description** (continued)
`cache_fraction`	Fraction of L2 memory used as L2 cache, in sixteenths. Must be a decimal constant (not a constant expression) from 0 to 16. If the L2 cache does not support the requested fraction, the closest smaller supported value is used. (At of this writing, the L2 cache only supports 0, 2, 4, 8, 16.) If `XTHAL_L2_SETUP()` is not invoked, default is dependent on startup code (for XTOS, it is 8 -- half L2RAM, half L2 cache).

The sum of `ram_fraction` and `cache_fraction` must be from 0 to 16. Behavior is undefined if any parameters are out of range.

For example, to configure three quarters of L2 memory as L2RAM and one quarter as L2 cache:

```
#include <xtensa/core-macros.h>

XTHAL_L2_SETUP(XCHAL_L2RAM_RESET_PADDR, 12, 4);
```

The variables `Xthal_L2cache_size` and Xthal_L2ram_size are set to the sizes computed by `XTHAL_L2_SETUP()`.

xthal_L2_repartition

The xthal_L2_repartition() function repartitions an already initialized L2 memory in order to change the allocations to `ram` and `cache`. The xthal_L2_repartition() function has two arguments for the fractions (in sixteenths) of the memory to be allocated to the `ram` and `cache.` If the L2 controller doesn't support the specified fraction, then the closest smaller value is used. The sum of `ram` and `cache` must be between 0 and 16. If successful, `xthal_L2_repartition()` returns 0, otherwise -1 is returned and the partition is not changed.

```
int32_t xthal_L2_repartiton(uint32_t ram, uint32_t cache);
```

For example, to configure three quarters of L2 memory as L2RAM and one quarter as L2 cache:

```
xthal_L2_repartition(12, 4);
```

The `xthal_L2_repartition()` function invalidates the L2 cache. It is typically necessary to disable writeback caching and writeback the entire L2 cache before calling `xthal_L2_repartition()` to prevent data loss. The L2RAM portion of the L2memory that is common to both the previous and new partitions is valid after repartitioning.

The variables `Xthal_L2cache_size` and `Xthal_L2ram_size` are set to the sizes computed by `xthal_L2_repartition()`.

The `xthal_L2_repartition()` function is subject to the same interrupt considerations as the asynchronous cache control functions (refer to Section 3.11.6). In addition, all memory that could potentially be accessed from an interrupt handler should be non-cacheable in the L2 cache.

xthal_get_cacheattr

xthal_set_cacheattr

The `cacheattr` operations allow reading and safely[5] writing the cache attributes (or access modes) of each of eight 512 MB regions of memory, on processors configured with such a layout. These functions may be called, for example, to effectively enable and disable caches on most processor configurations. They are not useful on processors that predate the RC-2009.0 release and are configured with a full MMU, as their memory layout is not arranged in eight regions. These functions are not supported on processors configured with an MPU.

Attributes of all eight regions are written or read at the same time. They are represented as a 32-bit integer consisting of eight 4-bit attributes (nibbles), with the least significant nibble corresponding to the first 512 MB region, at address zero (0x0000_0000). Possible values of each nibble are briefly described below. This models the `CACHEATTR` register that was present in processors configured with Xtensa Exception Architecture 1 (XEA1). On these processors, these functions simply read or write the `CACHEATTR` register directly. The same function definition was kept for backward compatibility.

In processors with Xtensa Exception Architecture 2 (XEA2) and simple region protection (with or without translation), the `xthal_get_cacheattr()` function reads the cache attributes from the data TLB, and `xthal_set_cacheattr()` writes the cache attributes of both instruction and data TLBs. They convert the `CACHEATTR` register layout to and from the TLB instructions and format as appropriate.

In processors from the RC-2009.0 or later releases with a full MMU (MMU v3, also XEA2), backward compatibility is also provided as long as the MMU is left in its default state, that is, mapping the entire 4 GB virtual space using TLB way 6. These functions operate similarly to the case of region protection (above), but using TLB way 6 instead of TLB way 0. If other TLB ways than way 6 are used, the effect of these functions is undefined; in this situation, caches are controlled using MMU facilities.

5. The Xtensa ISA states that it is illegal (or rather undefined) to change the cache attribute of the memory from which instructions are being fetched. However, Xtensa processor micro-architectures up to and including this release do allow it under certain specific conditions. For example, in Xtensa exception architecture 1 (XEA1), the code that sets `CACHEATTR` must be properly aligned relative to instruction caches, and execute a specific sequence that includes ISYNC and NOPs. The `xthal_set_cacheattr()` function takes care of executing the proper sequence so that enabling or disabling the instruction cache of any 512 MB region is safe.

On processors without any TLB (or CACHEATTR register), for example Xtensa TX, `xthal_set_cacheattr()` has no effect and `xthal_get_cacheattr()` returns 0.

In every other case, *i.e.*, in processors with XEA2 and MMU v1 (T1040 or T1050 release) or MMU v2 (RA or RB release), `xthal_set_cacheattr()` has no effect and the value returned by `xthal_get_cacheattr()` is undefined. Caches are instead controlled using MMU facilities.

On processors where this function has an effect, each encoded cache attribute (nibble) has several possible values, described in more detail in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* or relevant *Xtensa Microprocessor Data Book*. For the most common ones, the following compile-time HAL constants can be used:

▪ `XCHAL_CA_BYPASS`	cache disabled (bypassed)
▪ `XCHAL_CA_BYPASSBUF`	cache disabled, writes bufferable
▪ `XCHAL_CA_WRITETHRU`	cache enabled, write-through mode
▪ `XCHAL_CA_WRITEBACK`	cache enabled, write-back mode (allocate on store)
▪ `XCHAL_CA_WRITEBACK_NOALLOC`	cache enabled, write-back mode (no allocate on store)
▪ `XCHAL_CA_ILLEGAL`	no access allowed (exception on any load, store, or fetch)

For more details on the meaning of these cache attributes, see the *Xtensa LX Microprocessor Data Book,* specifically in the *Memory-Management Model Types* chapter, *Region Protection* section, and *Access Modes* subsection.

To use these constants, include `<xtensa/config/core.h>`.

For example, to disable the cache and enable access to the entire address space, you can use:

```
    xthal_set_cacheattr(XCHAL_CA_BYPASS * 0x11111111);
```

The `xt-genldscripts` tool defines several symbols in its generated linker scripts, whose addresses can be used as default values to pass to `xthal_set_cacheattr()`. The default XTOS reset vector uses this. See the *Xtensa Linker Support Packages (LSPs) Reference Manual* for more details.

```
    uint32_t xthal_get_cacheattr(void);     // read CACHEATTR register
    void xthal_set_cacheattr(uint32_t);     // write CACHEATTR register
```

xthal_get_icacheattr

xthal_set_icacheattr

xthal_get_dcacheattr

xthal_set_dcacheattr

These functions are equivalent to corresponding functions `xthal_get_cacheattr()` and `xthal_set_cacheattr()` described above, but apply to the instruction and data caches independently, where possible. These functions are not supported on processors configured with an MPU.

In processors configured with XEA1, these functions have the same effect as the corresponding functions `xthal_get_cacheattr()` and `xthal_set_cacheattr()` described above. That is, calling any of these functions affects both instruction and data cache attributes.

Otherwise, these functions only access the instruction or data TLB, respectively, as long as the corresponding `xthal_get_cacheattr()` and `xthal_set_cacheattr()` functions described earlier are defined. If not, these functions are also undefined.

```
uint32_t xthal_get_icacheattr(void); // read icache CACHEATTR equiv
void     xthal_set_icacheattr(uint32_t);// write  "
uint32_t xthal_get_dcacheattr(void); // read dcache CACHEATTR equiv
void     xthal_set_dcacheattr(uint32_t);// write  "
```

xthal_icache_sync

xthal_dcache_sync

The `sync` operation insures any previous cache operations are visible to subsequent code. It does not write out the contents of a write-back cache to memory, or invalidate any cache contents. `xthal_icache_sync()` is essentially an `ISYNC` instruction, and `xthal_dcache_sync()` essentially does nothing as no particular instruction is required to synchronize data cache operations.

```
void xthal_icache_sync(void);            // sync icache and memory
void xthal_dcache_sync(void);            // sync dcache and memory
```

xthal_icache_get_ways

xthal_dcache_get_ways

xthal_icache_set_ways

xthal_dcache_set_ways

This group of functions allows reading and setting of the number of enabled cache ways, separately for the instruction and data caches. The get() functions always return the number of ways enabled, even for caches that do not support dynamic way enable/disable. If no cache is present, the return value will be zero. The set() functions can be used to enable any number of cache ways from zero up to the maximum configured in hardware. If no cache is present, the function will do nothing.

Dynamic cache way support is only available from the RF-2014.0 release onwards.

```
uint32_t xthal_icache_get_ways(void); // read enabled icache ways
void     xthal_icache_set_ways(uint32_t);// write  "
uint32_t xthal_dcache_get_ways(void); // read enabled dcache ways
void     xthal_dcache_set_ways(uint32_t);// write  "
```

xthal_icache_all_invalidate

xthal_dcache_all_invalidate

The `invalidate` operation invalidates the contents of the entire cache. This may be called whether the cache is enabled or disabled.

`xthal_dcache_all_invalidate()` also operates on the integrated L2 cache if one is configured.

Note: Invalidating the cache does not affect locked lines. To invalidate locked lines, unlock them first.

Note: It is unlikely anyone would want to call `xthal_dcache_all_invalidate()` while the data cache is enabled — if the data cache works in write-back mode, any writes to memory that were cached and not yet written out are lost.

```
void xthal_icache_all_invalidate(void); // invalidate the icache
void xthal_dcache_all_invalidate(void);      // invalidate the dcache
```

xthal_dcache_all_writeback

The `writeback` operation writes back the contents of data cache (that is, any “dirty” entries) to memory. If the cache is not a write-back cache, this function does nothing but return. Any necessary synchronization instructions are also executed here.

This function has effect up to and including the integrated L2 cache if one is configured.

```
void xthal_dcache_all_writeback(void);  // write-back dcache to memory
```

xthal_dcache_all_writeback_inv

The `writeback` and `invalidate` operation is equivalent to a writeback followed by an invalidate. However it is faster than invoking these two operations separately, and avoids a race-condition with interrupts that might be taken between the two operations causing dirty entries that get corrupted by the invalidate operation. If the cache is not a write-back cache, only the invalidate operation is executed.

This function has effect up to and including the integrated L2 cache if one is configured.

```
void xthal_dcache_all_writeback_inv(void);// write dirty data and
invalidate
```

xthal_icache_all_unlock

xthal_dcache_all_unlock

xthal_L2_all_unlock

The `unlock` operation unlocks any cache line locked using the region or line lock operations (or equivalently, using the prefetch-and-lock instructions). If the cache does not support locking, this function does nothing but return.

```
void xthal_icache_all_unlock(void);
void xthal_dcache_all_unlock(void);
void xthal_L2_all_unlock(void);
```

xthal_set_cache_prefetch

This function sets the hardware cache prefetch mode. This controls automatic hardware prefetch for instruction and data caches and block prefetch to data cache, but not software cache prefetch (that is, the `DPFR`, `DPFRO`, `DPFW`, `DPFWO`, and `IPF` instructions).

```
int32_t xthal_set_cache_prefetch(uint64_t mode);
```

The function returns the previous cache prefetch mode setting, which may be passed to a subsequent call to `xthal_set_cache_prefetch()` to restore the cache prefetch mode. This return value is not meant to be used or interpreted directly.

The `mode` parameter can be given one of the following four values:

- The value returned from a previous call to `xthal_set_cache_prefetch()` or `xthal_get_cache_prefetch()`.
- The constant `XTHAL_PREFETCH_ENABLE` to enable cache prefetch and set up default values (see `XCHAL_CACHE_PREFCTL_DEFAULT` in Table 3–8). This applies to both instruction and data caches.
- The constant `XTHAL_PREFETCH_DISABLE` to disable cache prefetch. This applies to both instruction and data caches. Note: This does not affect block prefetch settings.
- A bit-wise OR of one or more of the following cache prefetch mode settings:

  ```
      [instruction cache prefetch aggressiveness]
    | [data cache prefetch aggressiveness]
    | [whether to prefetch directly to level-1 data cache]
    | [number of prefetch entries to use for block prefetch]
  ```

  At most one constant or macro can be used for each setting (see below). At least one setting must be specified, and any non-specified settings remain unchanged.

  Instruction cache prefetch aggressiveness:

`XTHAL_ICACHE_PREFETCH_OFF`	(disable instruction cache prefetch)
`XTHAL_ICACHE_PREFETCH_LOW`	(enable, less aggressive prefetch)
`XTHAL_ICACHE_PREFETCH_MEDIUM`	(enable, midway aggressive prefetch)
`XTHAL_ICACHE_PREFETCH_HIGH`	(enable, more aggressive prefetch)
`XTHAL_ICACHE_PREFETCH(`*n*`)`	`PREFCTL` *InstCtl* field value (0..15)[6]

  Data cache prefetch aggressiveness:

`XTHAL_DCACHE_PREFETCH_OFF`	(disable data cache prefetch)
`XTHAL_DCACHE_PREFETCH_LOW`	(enable, less aggressive prefetch)
`XTHAL_DCACHE_PREFETCH_MEDIUM`	(enable, midway aggressive prefetch)
`XTHAL_DCACHE_PREFETCH_HIGH`	(enable, more aggressive prefetch)
`XTHAL_DCACHE_PREFETCH(`*n*`)`	`PREFCTL` *DataCtl* field value (0..15)[7]

6. For details on possible values of the InstCtl field, see the `PREFCTL` register description in the *Prefetch Architectural Additions* section of the *Prefetch Unit Option* chapter in the *Xtensa Microprocessor Data Book*.

Whether to prefetch directly to level-1 data cache, if that feature is configured:

`XTHAL_DCACHE_PREFETCH_L1_OFF`	(prefetch data to prefetch buffers)
`XTHAL_DCACHE_PREFETCH_L1`	(prefetch directly to L1 data cache)

The number of prefetch entries to use for block prefetch, from 0 to the number of configured prefetch entries (`XCHAL_PREFETCH_ENTRIES`). These entries are used for *upgrade* operations, such as prefetch for read or write. Other block operations, such as prefetch for modify, and *downgrades* such as invalidate and writeback, have two dedicated prefetch entries not affected by this parameter. The default, selected using `XTHAL_PREFETCH_ENABLE` and also usually by software at startup, is to use half the number of configured prefetch entries.

`XTHAL_PREFETCH_BLOCKS(`*n*`)`	Decoded `PREFCTL` *BlockCtl* field (0..16)

Note: In some processor implementations, turning off either data or instruction prefetch has a side-effect of invalidating (clearing) the prefetch cache. Also, all features are not available in all implementations. See the *Xtensa Microprocessor Data Book* for details.

For example, the following code sequence temporarily disables data cache prefetch, then restores prefetch to its previous setting, without affecting other settings such as instruction cache prefetch and prefetch to level-1 cache:

```
int prefetch_save;
 :
prefetch_save = xthal_set_cache_prefetch(XTHAL_DCACHE_PREFETCH_OFF);
 :
 (code here executes with data cache prefetch disabled)
 :
xthal_set_cache_prefetch(prefetch_save);
```

See also the block prefetch example in Section 3.11.5 on page 139.

xthal_get_cache_prefetch

This function returns the current setting of the hardware cache prefetch mode. In the current implementation, this is simply the value of the `PREFCTL` register. However, for portability, consider avoiding this assumption, e.g. by only passing the value returned to a subsequent call to `xthal_set_cache_prefetch()`.

```
int32_t xthal_get_cache_prefetch(void);
```

7. For details on possible values of the DataCtl field, see the `PREFCTL` register description in the *Prefetch Architectural Additions* section of the *Prefetch Unit Option* chapter in the *Xtensa Microprocessor Data Book*.

3.11.3 Cache Region Control Functions

Region control operations apply to an arbitrarily sized contiguous sequence of bytes in memory.

Note: Except for `xthal_set_region_attribute()`, execution time for the current implementation of these functions is proportional to the number of cache lines covered by that sequence of bytes. Thus, unreasonably large requests could take a relatively long time to complete. The `xthal_set_region_attribute()` function takes time proportional to the size of the cache if it automatically writes back or invalidates the cache, and is relatively fast otherwise.

xthal_icache_region_invalidate

xthal_dcache_region_invalidate

xthal_dcache_region_writeback

xthal_dcache_region_writeback_inv

xthal_icache_region_lock

xthal_icache_region_unlock

xthal_dcache_region_lock

xthal_dcache_region_unlock

xthal_L2_region_lock

xthal_L2_region_unlock

In these functions, regions specified with non-cache-aligned addresses are handled inclusively. That is, all cache lines containing any of the bytes specified in the region will be affected, and no other. Behavior is undefined if `addr+size` exceeds the address space of the processor (*i.e.*, if region “wraps around” memory). No cache lines are affected if `size` is zero; however, synchronization instructions that would be required for a non-zero `size` may or may not get executed in this case (with no ill effect unless the caller incorrectly depends on their execution when `size` is zero).

Note that cache locking functions make no attempt to avoid locking too many lines. Calling `xthal_dcache_region_lock` or `xthal_icache_region_lock` will result in an exception if locking would result in no cache line available at a given cache index. It is up to the caller to either avoid locking too many lines, or handle the exception appropriately. `xthal_L2_region_lock` will not lock lines that result in no available cache lines at a given cache index, but a status bit will be set to indicate the locking failure.

`xthal_dcache_region_invalidate()`, `xthal_dcache_region_writeback()` and `xthal_dcache_region_writeback_inv()` have effect up to and including the integrated L2 cache if one is configured.

```
void xthal_icache_region_invalidate(void *addr, uint32_t size);
void xthal_dcache_region_invalidate(void *addr, uint32_t size);
void xthal_dcache_region_writeback(void *addr, uint32_t size);
void xthal_dcache_region_writeback_inv(void *addr, uint32_t size);
void xthal_icache_region_lock(void *addr, uint32_t size);
void xthal_icache_region_unlock(void *addr, uint32_t size);
void xthal_dcache_region_lock(void *addr, uint32_t size);
void xthal_dcache_region_unlock(void *addr, uint32_t size);
void xthal_L2_region_lock(void *addr, uint32_t size);
void xthal_L2_region_unlock(void *addr, uint32_t size);
```

xthal_set_region_attribute

This function sets the encoded memory access modes, or cache attributes, of the region of memory specified by `vaddr` and `size`. This function is only supported on processors configured with region protection, an MPU, or v3 MMU (RC or later release MMU) left in its default mapping state. Refer to the xthal_mpu_set_region_attribute for information specific to processors with an MPU configured. It has no effect otherwise, *i.e.*, on processors configured with a v1 or v2 MMU (prior to RC). This function automatically writes back and/or invalidates relevant cache entries to ensure memory coherency; see below for details, and Section 3.11.8 on page 146 for general coverage of the coherency issues.

```
int32_t xthal_set_region_attribute( void *vaddr, uint32_t size,
                                    uint32_t cattr, uint32_t flags );
```

The full (4 GB) address space may be specified with an address of zero and a size of `0xFFFFFFFF` (or -1); in fact whenever `vaddr+size` equal `0xFFFFFFFF`, `size` is interpreted as one byte greater than that specified.

If the specified memory range exactly covers a series of consecutive 512 MB regions, the cache attributes of these regions are updated with the requested attribute. If this is not the case, *e.g.*, if either or both the start and end of the range only partially cover a 512 MB region, one of three results are possible:

- By default, the cache attribute of all regions covered, even just partially, is changed to the requested attribute.
- If the `XTHAL_CAFLAG_EXACT` flag is specified, an non-zero error code is returned.
- If the `XTHAL_CAFLAG_NO_PARTIAL` flag is specified (but not the `EXACT` flag), only regions fully covered by the specified range are updated with the requested attribute.

Table 3–18 and Table 3–19 describe the function parameters and return values, respectively.

Table 3–18. `xthal_set_region_attribute` Parameters

Parameter	Description
`vaddr`	Starting virtual address of region of memory
`size`	Number of bytes in the region of memory
`cattr`	Cache attribute (encoded), typically one of the following compile-time HAL constants: `XCHAL_CA_BYPASS` cache disabled (bypassed) `XCHAL_CA_BYPASSBUF` cache disabled, writes bufferable `XCHAL_CA_WRITETHRU` cache enabled, write-through mode `XCHAL_CA_WRITEBACK` cache enabled, write-back mode (allocate on store) `XCHAL_CA_WRITEBACK_NOALLOC` cache enabled, write-back mode (no allocate on store) `XCHAL_CA_ILLEGAL` no access (load, store, fetch) allowed (exception) For more details on the meaning of these cache attributes, see the *Xtensa LX Microprocessor Data Book,* specifically in the *Memory-Management Model Types* chapter, *Region Protection* section, and *Access Modes* subsection. To use these constants, include `<xtensa/config/core.h>`. In XEA1, this corresponds to the value of a nibble in the CACHEATTR register. In XEA2, this corresponds to the value of the cache attribute (CA) field of each TLB entry.

Table 3–18. `xthal_set_region_attribute` Parameters (continued)

Parameter	Description	
`flags`	Bitwise combination of the following flags (0 if none), defined in `<xtensa/hal.h>`:	
	`XTHAL_CAFLAG_EXACT`	return error if the requested attribute cannot be applied to the exact range specified
	`XTHAL_CAFLAG_NO_PARTIAL`	only apply the requested attribute to regions or pages completely covered by the specified range; don't modify regions/pages only partially covered
	`XTHAL_CAFLAG_EXPAND`	only expand allowed access to the specified range (see below). Not supported on processors configured with an MPU
	`XTHAL_CAFLAG_NO_AUTO_WB`	don't automatically writeback dirty data when leaving writeback attribute (see above)
	`XTHAL_CAFLAG_NO_AUTO_INV`	don't automatically invalidate cache after entering bypass attribute (disabling the cache) (see above)

Table 3–19. `xthal_set_region_attribute` return values

Return Value	Description
`XTHAL_SUCCESS`	Successful, or size is zero
`XTHAL_NO_REGIONS_COVERED`	`XTHAL_CAFLAG_NO_PARTIAL` flag specified and address range is valid with a non-zero size, however no 512 MB region (or page) is completely covered by the range
`XTHAL_INEXACT`	`XTHAL_CAFLAG_EXACT` flag specified, and address range does not exactly specify a 512 MB region (or page)
`XTHAL_INVALID_ADDRESS`	Invalid address range specified (wraps around the end of memory)
`XTHAL_UNSUPPORTED`	Function not supported in this processor configuration
`XTHAL_BAD_ACCESS_RIGHTS`	The specified access rights are not valid (only applicable to the MPU)
`XTHAL_BAD_MEMORY_TYPE`	The specified memory type is not valid (only applicable to the MPU)
`XTHAL_OUT_OF_ENTRIES`	The requested attributes could not be set because the MPU is out of map entries (only applicable to the MPU)

When caches are writeback caches, this function automatically writes back dirty data when switching a region from writeback mode to a non-writeback mode. This writeback is done safely, that is, by first switching to writethrough mode, then invoking `xthal_dcache_all_writeback`(), then switching to the selected `cattr` mode. Such a sequence is necessary to ensure there is no longer any dirty data in the memory region by the time this function returns, even in the presence of interrupts, speculation, etc. This avoids memory coherency problems when switching from writeback to bypass mode (in bypass mode, loads go directly to memory, ignoring any dirty data in the cache; also, such dirty data can still be cast out due to seemingly unrelated stores). The `XTHAL_CAFLAG_NO_AUTO_WB` flag disables this automatic writeback behavior.

When disabling the cache, i.e. when switching a region from non-bypass mode to bypass mode, this function automatically invalidates any corresponding entries from the cache. This ensures memory coherency if you disable the cache using this function, modify memory, then re-enable the cache. Otherwise, stale cache entries may be present when re-enabling the cache.

For performance reasons, invalidation may be done across the entire cache instead of just the affected region(s) of memory. In this case, combined writeback-invalidate operations are used instead of simple invalidate operations to ensure coherency. The `XTHAL_CAFLAG_NO_AUTO_INV` flag disables the automatic invalidate behavior.

The `XTHAL_CAFLAG_EXPAND` flag prevents attribute changes to regions whose current cache attribute already provide greater access than the requested attribute. The `XTHAL_CAFLAGS_EXPAND` flag is not supported on processors configured with the MPU option. This ensures access to each region can only "expand", and thus continue to work correctly in most instances, possibly at the expense of performance. This helps make this flag safer to use in a variety of situations. For the purposes of this flag, cache attributes are ordered (in "expansion" order, from least to greatest access) as follows:

▪ `XCHAL_CA_ILLEGAL`	no access allowed
▪ (certain other cache attribute values)	various special and reserved attributes
▪ `XCHAL_CA_WRITEBACK`	cache enabled, write-back mode
or `XCHAL_CA_WRITEBACK_NOALLOC`	cache enabled, write-back mode, don't allocate a cache line on store miss
▪ `XCHAL_CA_WRITETHRU`	cache enabled, write-through mode
▪ `XCHAL_CA_BYPASSBUF`	cache disabled, writes bufferable
▪ `XCHAL_CA_BYPASS`	cache disabled (bypassed)

For more details on the meaning of these cache attributes, see the *Xtensa LX Microprocessor Data Book,* specifically in the *Memory-Management Model Types* chapter, *Region Protection* section, and *Access Modes* subsection.

This is consistent with requirements of certain devices that no caches be used, or in certain cases that writethrough caching is allowed but not writeback. Thus, bypass mode is assumed to work for most/all types of devices and memories (albeit at reduced performance compared to cached modes), and is ordered as providing greatest access (to most devices).

Thus, the `XTHAL_CAFLAG_EXPAND` flag has no effect when requesting the `XCHAL_CA_BYPASS` attribute (one can always expand to bypass mode). And at the other extreme, no action is ever taken by this function when specifying both the `XTHAL_CAFLAG_EXPAND` flag and the `XCHAL_CA_ILLEGAL` cache attribute.

3.11.4 Cache Block Control Functions (Block Prefetch)

Cache block control operations are similar to region control operations in that they apply to a variable sized contiguous sequence of bytes in memory. However, they are implemented using the cache block instructions available with the Block Prefetch Option or the Cache Management Engine Option.

Note: None of these function operate on the integrated L2 cache if one is configured. In particular, writeback and invalidate dcache operations only affect the L1 dcache, not L2.

Block operations can be roughly divided into three different sets of operations - *Upgrades*, *Downgrades*, and *Prefetch-and-Modify*. The functions implemented follow these three sets of operations, providing the following capabilities:

- *Upgrade* (or prefetch) - bringing a region of memory into the processor's data or instruction cache. C functions that perform actual upgrades have *_prefetch_* as a part of the functions name. These operations can be terminated at any time without having affect on the program or the system functionality.
- *Downgrade* - evicting a block of data from the cache. These operations need to be completed as the code or actual system might depend on their correct execution.
- *Prefetch-and-Modify* - a special operation that prepares the cache for future write operations. Unlike normal prefetches, this operation affects the effective contents of memory.

All Block Prefetch functions complete immediately from a *local* functional point of view, while the actual cache block operations can carry on indefinitely.These functions require either the Block Prefetch or Cache Management Engine option. If the Cache Management Engine option is configured, only the downgrade operations, `xthal_dcache_block_wait`, `xthal_dcache_block_required_wait`, and `xthal_dcache_block_abort` are supported.

Note: If an MPU is configured, any block operation that crosses a MPU region will result in an exception.

Using Cache Block Operations

Segments of code typically need more than one block to accomplish their work. In addition, the program might want to overlap prefetching of data for two sequential code segments so that the work can be more effectively averaged over time. To accomplish this, several Block Prefetch operations can be scheduled before the first has completed in the background. The maximum number of currently outstanding block prefetch operations is a configuration parameter in is available using `XCHAL_PREFETCH_BLOCK_ENTRIES` define. Issuing more block operations than this limit results in the block prefetch hardware stalling the core pipeline until it's able to accept more requests. This, in turn, results in stalled program execution.

Note: All the scheduled block operations don't necessarily execute in parallel. For example, the hardware might impose the limit on maximum number of block operations that can execute in parallel.

Block operations can also be started with additional mark called "Group Begin". Any Block Prefetch marked as "Group Begin" is only scheduled but doesn't executed as long as there are outstanding prefetches in older groups. The parallel operation can, by this method, be limited to the operations needed by the earliest code segment while still allowing the software to queue up additional operations to be started when the critical group is complete. See the *Xtensa Microprocessor Data Book* for more details on Block Prefetch architecture, configuration and implementation restrictions, limitations and available prefetch strategies.

The most important question regarding the prefetches is when to use them in an application. The very first thing to understand is that block operations are meant for the performance improvements. Thus, the successful use of a block operations depends on the application and needs to be determined by the performance optimization flow. First of all, the user needs to identify performance bottlenecks in an application which come from the stalls due to memory accesses. Then, the user needs to determine which, if any, block operations can be performed earlier in the program to avoid those latter memory access penalties. This helps identifying exactly which block operations should be added to improve the performance. Finding exact place in the program where the prefetch should happen is a part of the performance optimization techniques and likely requires many iterations to achieve the optimal, or just satisfactory result.

xthal_dcache_block_invalidate

xthal_dcache_block_invalidate_max

```
void xthal_dcache_block_invalidate(void *addr, uint32_t size);
void xthal_dcache_block_invalidate_max(void *addr, uint32_t size,
                                        uint32_t constmax);
```

Invalidate the cache lines that fall within the block specified with the `addr` (non-cache-aligned) and `size` (in bytes). Partially contained lines are processed as well.

The *_max* variant can be used when `size` is known not to exceed `constmax` (in bytes), for possible better performance if `constmax` is no larger than the data cache size. The _max variant provides no performance advantage when the Cache Management Engine is configured.

xthal_dcache_block_writeback

xthal_dcache_block_writeback_max

```
void xthal_dcache_block_writeback(void *addr, uint32_t size);
void xthal_dcache_block_writeback_max(void *addr, uint32_t size,
                                        uint32_t constmax);
```

Writeback the cache lines that fall within the block specified with the `addr` (non-cache-aligned) and `size` (in bytes). Partially contained lines are processed as well.

The *_max* variant can be used when `size` is known not to exceed `constmax` (in bytes), for possible better performance if `constmax` is no larger than the data cache size. The _max variant provides no performance advantage when the Cache Management Engine is configured.

xthal_dcache_block_writeback_inv

xthal_dcache_block_writeback_inv_max

```
void xthal_dcache_block_writeback_inv(void *addr, uint32_t size);
void xthal_dcache_block_writeback_inv_max(void *addr, uint32_t size,
                                            uint32_t constmax);
```

Writeback and invalidate the cache lines that fall within the block specified with the `addr` (non-cache-aligned) and `size` (in bytes). Partially contained lines are processed as well.

The *_max* variant can be used when `size` is known not to exceed `constmax` (in bytes), for possible better performance if `constmax` is no larger than the data cache size. The _max variant provides no performance advantage when the Cache Management Engine is configured.

The above six functions are also called the *Downgrade* HAL functions. All the *Downgrades* affect no cache lines if their `size` argument is zero. The *_max* versions of all the *Downgrade* functions take a constant argument that the caller promises will not be exceeded by `size`: this allows avoiding a loop around the block operation instruction if the maximum is within the data cache size.

Note: When the Block Prefetch option is configured - *Downgrade* functions use underlying block-operation instructions, all of which generate exception if the `size` exceeds the data cache size. *Downgrade* HAL functions do not generate exceptions because they split large block requests into multiple smaller requests (if necessary only for *`_max`* variants), each one being smaller than the data cache size. If the Cache Management Engine is configured, the underlying instructions operate efficiently even when `size` exceeds the data cache size.

xthal_dcache_block_prefetch_for_read

xthal_dcache_block_prefetch_for_read_grp

```
void xthal_dcache_block_prefetch_for_read(void *addr, uint32_t size);
void xthal_dcache_block_prefetch_for_read_grp(void *addr, uint32_t
                                              size);
```

Fetch the memory block indicated with `addr` (non-cache-aligned) and `size` (in bytes), into the L1-DCache. Partially contained lines are fetched as well. These functions are intended for prefetching the cache lines which are expected to be accessed by the read instructions. In coherent systems, the lines are requested using an exclusive request.

xthal_dcache_block_prefetch_for_write

xthal_dcache_block_prefetch_for_write_grp

```
void xthal_dcache_block_prefetch_for_write(void *addr, uint32_t size);
void xthal_dcache_block_prefetch_for_write_grp(void *addr, uint32_t
                                               size);
```

Fetch the memory block indicated with `addr` (non-cache-aligned) and `size` (in bytes), into the L1-DCache. Partially contained lines are fetched as well. These functions are intended for prefetching the cache lines which are expected to be accessed by the write as well as the read instructions. In coherent systems, the lines are requested using an exclusive request.

xthal_dcache_block_prefetch_read_write

xthal_dcache_block_prefetch_read_write_grp

```
void xthal_dcache_block_prefetch_read_write(void *addr,
                                             uint32_t size);
void xthal_dcache_block_prefetch_read_write_grp(void *addr,
                                             uint32_t size);
```

Fetch the memory block indicated with `addr` (non-cache-aligned) and `size` (in bytes), into the L1-DCache. Partially contained lines are fetched as well. These functions are intended for prefetching the cache lines which are expected to be accessed by the write as well as the read instructions. In contrast to `xthal_dcache_block_prefetch_for_write`, these functions exist for both writeback and write-thru types of data caches. In coherent systems, the lines are requested using an exclusive request.

xthal_dcache_block_prefetch_modify

xthal_dcache_block_prefetch_modify_grp

```
void xthal_dcache_block_prefetch_modify(void *addr,
                                         uint32_t size);
void xthal_dcache_block_prefetch_modify_grp(void *addr,
                                         uint32_t size);
```

Fetch the memory block indicated with `addr` (non-cache-aligned) and `size` (in bytes) into the L1-DCache. Partially contained lines are fetched as well. These functions are intended for allocating the cache lines which are expected to be accessed by the write instructions. The fetched lines are arbitrarily modified. The function has advantage over `xthal_dcache_block_prefetch_for_write` as the lines are actual not read from the memory but are simply allocated. In coherent systems, a message is sent informing other masters to drop requested lines. The lines which are partially contained within the specified block range are requested as if `xthal_dcache_block_prefetch_for_write` was used.

The purpose of the *Prefetch-and-Modify* operation is to avoid cache misses, and subsequent cache refill operations from the main memory, in case when certain caches lines are going to be rewritten anyway. By using the *Prefetch-and-Modify* function, the user can, ahead of the actual write operation, validate all the cache lines in the given block, ensuring that the actual write does not generate cache misses. This operation does not access the memory, but caches only. By validating possibly invalid data in the cache the operations is functionally destructive. The application must ensure it is used correctly and the data that it modified is not accessed by any other code until the actual write validates all the cache lines.

The above eight functions are also referred to as the *Upgrade* HAL functions. All the *Upgrades* do the block prefetch up to 2x data cache size (1x for *Prefetch-and-Modify* operations) and ignore data beyond this limit. The *_grp* version of any *Upgrade* operation will start the block prefetch operation as a part of a new prefetch group. All *Upgrades* from the same group are not executed until there is an older group of prefetches in progress.

xthal_dcache_block_required_wait

```
    void xthal_dcache_block_required_wait();
```

Wait for all required block operations to end (i.e. wait for downgrades to end) and for any writes caused by the block operation to complete.

xthal_dcache_block_wait

```
    void xthal_dcache_block_wait();
```

Wait for all block operations to end, either upgrades (prefetches) or downgrades and for any writes caused by the block operation to complete.

xthal_dcache_block_abort

```
    void xthal_dcache_block_abort();
```

End all block operations (scheduled or in progress). This is a destructive operation as it also aborts all the downgrade operations (invalidate/writeback) while the program might depend on those being executed properly.

xthal_dcache_block_end

```
    void xthal_dcache_block_end();
```

End all the cache block operations which are not destructive (i.e. prefetches - upgrades). Safe to execute as the prefetch itself is just a performance optimization operation.

xthal_dcache_block_newgrp

```
    void xthal_dcache_block_newgrp();
```

Instructs the block prefetch logic to start a new group. All the block cache operations that follow are executed within this new group of block operations. Note that these operations will not get scheduled for execution until any cache block operation from an older group is still in progress.

3.11.5 Block Prefetch Example

The following code illustrates a simple use of block prefetch to copy a block of bytes.

```
    #include <stdio.h>
    #include <string.h>
    #include <xtensa/core-macros.h>
```

```
/*  Alignment is not required but helps performance slightly.  */
#define CACHE_ALIGN __attribute__((aligned(XCHAL_DCACHE_LINESIZE)));

short int a_vector[1024] CACHE_ALIGN;
short int b_vector[1024] CACHE_ALIGN;

int main(void)
{
  /* allocate max 8 entries for block prefetch */
  int prefetch_state = xthal_set_cache_prefetch
(XTHAL_PREFETCH_BLOCKS(8) |
        XTHAL_DCACHE_PREFETCH_MEDIUM | XTHAL_ICACHE_PREFETCH_MEDIUM |
        XTHAL_DCACHE_PREFETCH_L1);

  /* for debugging only */
  printf("PREFETCH default: %08x\n", prefetch_state);
  printf("PREFETCH new    : %08x\n", xthal_get_cache_prefetch());

  /*- - - - - other code here - - - - - */

  /* prefetch-for-modify dest buffer, start block prefetch group */
  xthal_dcache_block_prefetch_modify_grp (b_vector, sizeof(b_vector));

  /* prefetch source buffer */
  xthal_dcache_block_prefetch_for_read (a_vector, sizeof(a_vector));

  /*- - - - - other code here - - - - - */

  /* copy 2048 bytes */
  memcpy ((char*)b_vector, (char*)a_vector, sizeof(b_vector));

  /*- - - - - other code here - - - - - */

  /* restore prefetch state */
  xthal_set_cache_prefetch (prefetch_state);
  printf("Test done\n");
  return 0;
}
```

3.11.6 Asynchronous Cache Operations

An asynchronous interface is provided that supports cache prefetch and downgrade operations. This interface is only useful if an L2 cache is configured. Only one asynchronous operation is permitted at a time, and the functions in the interface are not reentrant.

As a consequence, the asynchronous interface can only be used in one thread at a time, and, it cannot be used in interrupt handlers, unless all users of the asynchronous operations disable the applicable interrupt.

The asynchronous cache operations include prefetch (with and without locking) to the L2 cache and downgrade operations that affect both L1 and L2 caches. Only one asynchronous operation can be in progress at a time.

In principle, these operations are similar to the block operations defined in Section 3.11.4 “Cache Block Control Functions (Block Prefetch)”, but there are a few important differences:

- The asynchronous dcache functions apply to both the L1 and L2 cache.
- Only one asynchronous operation can be in progress at a time, and are not reentrant.
- The asynchronous operations require an explicit wait to ensure the operation completes. Unlike the block operations, the asynchronous cache operations do not have locally immediate functionality.

xthal_async_L2_prefetch

Initiates a prefetch operation to bring the address range [`addr`, `addr + size - 1`] into the L2 cache. This function only affects the contents of the L2 cache and has no effect on the L1 cache.

This function is useful in improving performance, by bringing data into the L2 cache before it is needed. The prefetch operates asynchronously, so the core initiating the prefetch is not blocked during the prefetch.

This function differs from the other asynchronous functions in calling any other asynchronous function while `xthal_async_L2_prefetch` is in progress causes the prefetch to be aborted. This function is silently ignored if an asynchronous downgrade operation is already in progress, or if another prefetch is in progress.

The only way to ensure that the prefetch has completed is to call `xthal_async_L2_wait()`.

The results are undefined if:

- `size` is greater than the size of the L2 cache
- `addr + size` exceeds the address space of the processor (*i.e.*, if the region "wraps around" memory).

```
void xthal_async_L2_prefetch(void* addr, uint32_t size)
```

xthal_async_L2_region_lock

Initiates a prefetch and lock operation to bring the address range [`addr`, `addr + size - 1`] into the L2 cache. This memory address range is locked into the L2 cache. This function only affects the contents of the L2 cache and has no effect on the L1 cache.

This function is useful in improving performance, by locking frequently used data into the L2 cache. The prefetch and lock operates asynchronously, so the core initiating it is not blocked during the prefetch.

This function makes no attempt to prevent the locking of too many lines. If locking a cache line would lead to possible deadlock, then the line is left unlocked.

If an asynchronous prefetch (without locking) operation is in progress, the prefetch is terminated and the prefetch and lock operation is started. If another asynchronous operation is already in progress, then no new operation is started, and the function returns `XTHAL_ASYNC_BUSY`.

Regions specified with non-cache-aligned addresses are handled inclusively. If `addr+size` exceeds the address space of the processor or if `size` is greater than 16MB, then `XTHAL_ASYNC_RANGE` is returned.

The only way to ensure that the prefetch and lock has completed is to call `xthal_async_L2_wait()`.

```
    int32_t xthal_async_L2_region_lock(void* addr, uint32_t size)
```

xthal_async_L2_region_unlock

Initiates an unlock operation for all the cache lines included in the region [addr, addr + size - 1]. This function only affects the L2 cache and has no effect on the L1 cache.

If an asynchronous prefetch (without locking) operation is in progress, the prefetch is terminated and the unlock operation is started. If another asynchronous operation is already in progress, then no new operation is started, and the function returns `XTHAL_ASYNC_BUSY`.

Regions specified with non-cache-aligned addresses are handled inclusively. If `addr+size` exceeds the address space of the processor or if `size` is greater than 16MB, then `XTHAL_ASYNC_RANGE` is returned.

The only way to ensure that the unlock has completed is to call `xthal_async_L2_wait()`.

```
    int xthal_async_L2_region_unlock(void* addr, uint32_t size)
```

xthal_async_L2_all_unlock

Initiates an unlock operation for the entire L2 cache.

If an asynchronous prefetch (without locking) operation is in progress, the prefetch is terminated and the unlock operation is started. If another asynchronous operation is already in progress, then no new operation is started, and the function returns `XTHAL_ASYNC_BUSY`.

```
    int32_t xthal_async_L2_region_unlock(void)
```

xthal_async_L2_wait

Blocks until the previous asynchronous operation is complete. A prefetch operation that is abandoned because another asynchronous operation is in progress, or one that is ended (by `xthal_async_L2_prefetch_end()`) is considered complete.

```
    void xthal_async_L2_wait(void)
```

xthal_async_L2_prefetch_end

If an L2 prefetch operation is in progress, then it is canceled. The cancellation is asynchronous, so the portion of a canceled prefetch that completes is undefined. This function does not wait for the controller to finish the cancellation.

```
    void xthal_async_L2_prefetch_end(void)
```

xthal_async_dcache_region_writeback

xthal_async_dcache_region_writeback_inv

xthal_async_dcache_region_invalidate

Initiates an asynchronous cache downgrade operation (writeback, writeback and invalidate, or invalidate) on the specified range of addresses for both the L1 data cache and L2 cache (if configured).

If an asynchronous downgrade operation is in progress, no new operation is started, and the function returns `XTHAL_ASYNC_BUSY`. If an asynchronous prefetch operation is in progress, the prefetch is terminated and the downgrade operation is started.

In these functions, regions specified with non-cache-aligned addresses are handled inclusively. If `addr+size` exceeds the address space of the processor or if `size` is greater than 16MB, then `XTHAL_ASYNC_RANGE` is returned.

The cache state must be consistent with the MPU's memory type. If L1 (or L2) cache is writethrough, all lines must be clean (not dirty) in the respective cache. A suitable state is guaranteed, if:

- All updates to the MPU are done through the HAL API
- The `XTHAL_CAFLAG_NO_AUTO_INV` flag is not used

The functions are asynchronous. They are known to complete only after a call to `xthal_async_dcache_wait()`, or when `xthal_async_dcache_busy()` returns false.

Upon success, these functions return 0.

```
int32_t xthal_async_dcache_region_writeback(void* addr, uint32_t size)
int32_t xthal_async_dcache_region_writeback_inv(void* addr, uint32_t
size)
int32_t xthal_async_dcache_region_invalidate(void* addr, uint32_t
size)
```

xthal_async_dcache_all_writeback

xthal_async_dcache_all_writeback_inv

Initiates an asynchronous cache downgrade operation (writeback, or writeback and invalidate) on the entirety of both the L1 data cache and L2 caches (if configured). If an asynchronous prefetch operation is in progress, that operation is terminated and the downgrade operation is started. If another asynchronous downgrade option is in progress, no new operation is started, and these functions return `XTHAL_ASYNC_BUSY`.

Upon success, these functions return 0.

```
int32_t xthal_async_dcache_all_writeback_inv(void)
int32_t xthal_async_dcache_all_writeback(void)
```

xthal_async_dcache_wait

If an asynchronous downgrade operation is in progress, `xthal_async_dcache_wait()` blocks until that operation has been completed.

```
void xthal_async_dcache_wait(void)
```

xthal_async_dcache_busy

Returns 1 if an asynchronous downgrade operation is in progress, and 0 otherwise. Note that 0 is returned if an L2 prefetch is in progress.

```
    int32_t xthal_async_dcache_busy(void)
```

xthal_async_dcache_abort

Cancels any asynchronous downgrade operation that is in progress. The cancellation is asynchronous, so the portion of a canceled downgrade that completes is undefined. This function does not wait for the controller to finish the cancellation. `xthal_async_dcache_wait()` or `xthal_async_dcache_busy()` can be used to ensure the cancellation is complete.

```
    void xthal_async_dcache_abort(void)
```

3.11.7 Cache Line Control Functions and C macros

Finally, there are cache control functions that operate on a single cache line. Each of these functions takes an address parameter, `addr`, which identifies a single cache line. In cases where `addr` is not aligned, the cache line is that which includes the byte pointed to by `addr` (*i.e.*, the address is implicitly rounded down).

`xthal_dcache_line_invalidate()`, `xthal_dcache_line_writeback()` and `xthal_dcache_line_writeback_inv()` have effect up to and including the integrated L2 cache if one is configured.

Note that cache locking may result in taking an exception. See Section 3.11.3 for details.

```
    void xthal_icache_line_invalidate(void *addr);
    void xthal_dcache_line_invalidate(void *addr);
    void xthal_dcache_line_writeback(void *addr);
    void xthal_dcache_line_writeback_inv(void *addr);
    void xthal_icache_line_lock(void *addr);
    void xthal_icache_line_unlock(void *addr);
    void xthal_dcache_line_lock(void *addr);
    void xthal_dcache_line_unlock(void *addr);
    void xthal_L2_line_lock(void *addr);
    void xthal_L2_line_unlock(void* addr);
```

xthal_dcache_line_prefetch_for_write

```
    void xthal_dcache_line_prefetch_for_write(void *addr);
```

Prefetch a L1-DCache line from the memory in the expectation of future writes to that cache line. The parameter, `addr`, identifies the cache line.

xthal_dcache_line_prefetch_for_read

```
    void xthal_dcache_line_prefetch_for_read(void *addr);
```

Prefetch a L1-DCache line from the memory in the expectation of future reads from that cache line. The parameter, `addr`, identifies the cache line.

3.11.8 Safely Changing Memory Access Modes

This section describes the cache maintenance operations that may be necessary when changing the cache attribute (memory access mode) of a specific area of memory. It first outlines general principles, followed by specific sequences starting on page 150.

Note that the `xthal_set_region_attribute()` function described in Section 3.11.3 (or the `xthal_mpu_set_region_attribute()` function described in Section 3.12.6) generally handles these cache maintenance operations automatically.

General Principles

Cache operations required to change attributes are a function of the old and new cache attributes, and of what is known about the state of cache entries covering the area of memory affected. This is illustrated by the diagram in Figure 3–1, which depicts relevant combinations of cache attribute and cache state. This is an exhaustive diagram showing a few states that are not relevant to software; however, it provides a better understanding for the simplified diagrams in Figure 3–2 and Figure 3–3.

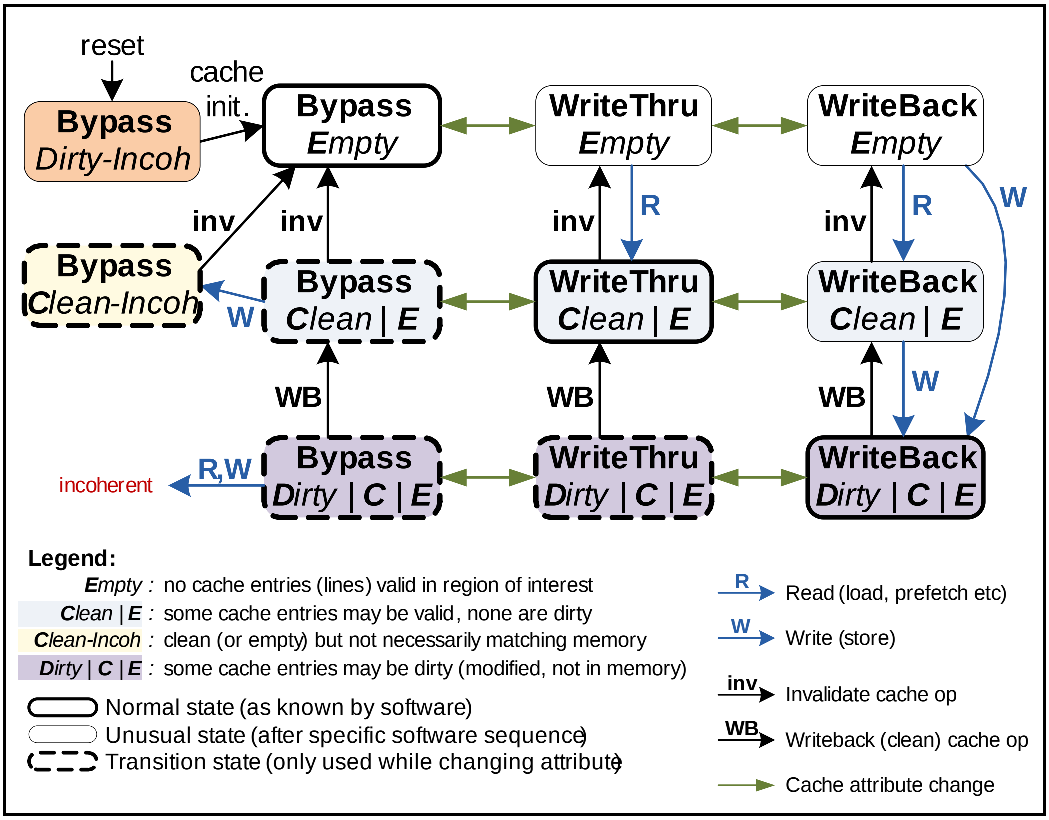
reset
cache init.
Bypass Dirty-Incoh
Bypass Empty
WriteThru Empty
WriteBack Empty
inv
inv
inv
inv
R
R
W
Bypass Clean-Incoh
W
Bypass Clean | E
WriteThru Clean | E
WriteBack Clean | E
WB
WB
WB
W
incoherent
R,W
Bypass Dirty | C | E
WriteThru Dirty | C | E
WriteBack Dirty | C | E
Legend:
Empty : no cache entries (lines) valid in region of interest
Clean | E : some cache entries may be valid, none are dirty
Clean-Incoh : clean (or empty) but not necessarily matching memory
Dirty | C | E : some cache entries may be dirty (modified, not in memory)
Normal state (as known by software)
Unusual state (after specific software sequence)
Transition state (only used while changing attribute)
R Read (load, prefetch etc)
W Write (store)
inv Invalidate cache op
WB Writeback (clean) cache op
Cache attribute change

Figure 3–1. Hypothetical Cache State Diagram – Reads and Writes can be Suspended

Note that Figure 3–1 only considers accesses by a single core. Multiple cores sharing memory must normally use identical or compatible cache attributes; how they coordinate changes of attributes, or manage cache coherency of shared accesses, is outside the scope of this discussion.

Three basic cache attributes are shown: bypass-cache (uncached), write-through, and writeback. Transitions between similar cache attributes require no special action and are not shown (such as between bufferable and non-bufferable variants of bypass, or between allocate and no-allocate variants of writeback). Attributes that allow no access at all are also not shown: a memory region may be switched to them and back at any time. Although not shown explicitly, cache attributes may transition directly across multiple allowed transitions, such as directly between Bypass and WriteBack in a given row.

These are combined with three possible states of the cache for the specific area of memory affected. These represent the state of the cache *as known by software*, not the actual hardware state of the cache. Thus, “Dirty **|** C **|** E” means the cache might be dirty, but it might also be clean or empty; and “Clean **|** E” means the cache may or may not be empty but it is clean (not dirty). “Empty” means there are no valid cache entries covering that area of memory (this state reflects both hardware and what is known by software).

The state changes as cache entries are filled as shown in a blue **R** (typically due to a load or prefetch of some kind in cached mode), and when dirtied as shown in a blue **W** (normally due to a write operation in writeback cache mode). These actions *upgrade* cache entries from Empty to Clean to Dirty. Conversely, cache state also changes, *downgrading* from Dirty to Clean to Empty, as cache maintenance operations are executed, including writeback (shown as **WB**), invalidate (shown as **inv**; destructive invalidates from “Dirty **|** C **|** E” are not shown), and writeback-invalidate (not shown; in this diagram it is equivalent to **WB** immediately followed by **inv**[8]).

What makes this diagram in Figure 3–1 hypothetical is that in practice, software generally has little control of cache fills. While it may be possible on a specific Xtensa core implementation with careful consideration of all configured features, architecturally there is no guarantee due to cache prefetch, speculation and other factors (not to mention compiler re-ordering and debugger activities): cache entries may be filled (become valid) at any time for cacheable areas of memory. Thus, any **R** transition must be assumed to always occur immediately, collapsing away the state from which it came. This leads to Figure 3–2.

Figure 3–2 shows the possible transitions when software is able to suspend all writes to the memory region while changing its cache attribute. In any transition involving a state from which a **W** transition is possible, software must have suspended all such writes to avoid the **W** transition.

8. In practice, note that separate writeback and invalidate operations are generally not equivalent to a combined writeback-invalidate operation, because a simple invalidate operation destroys dirty entries and thus requires write permission, whereas writeback-invalidate does not.

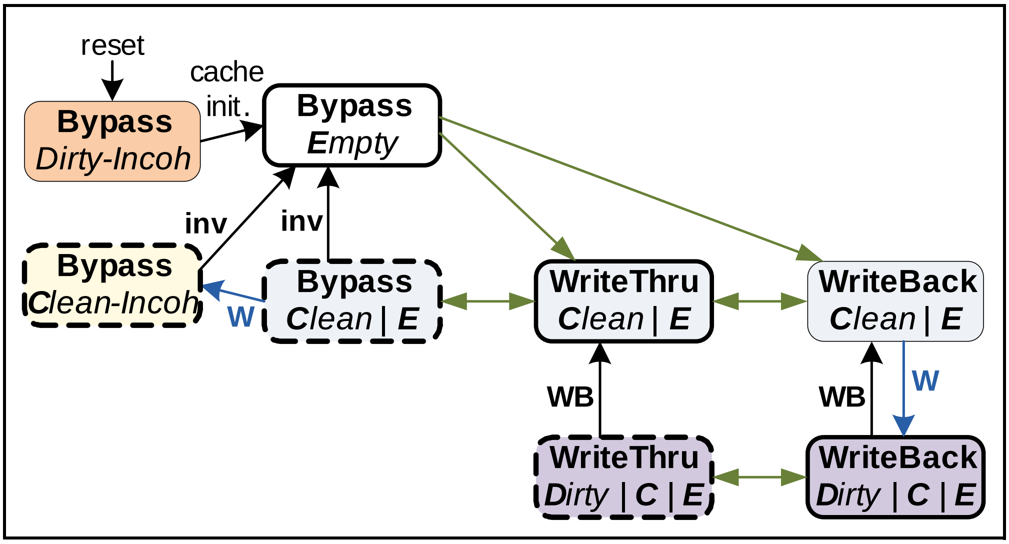

Figure 3–2. Cache State Diagram – Writes can be Suspended

It is commonly desirable to let writes to the memory region of interest continue through the attribute change. For example, this occurs when changing the attribute of a region of memory in use by other threads, by interrupt or exception handlers, or similar activity. In that case, states from which a **W** transition is possible are collapsed, leading to the simplified diagram of Figure 3–3. This is the cache state transition diagram usually recommended for changing cache attributes, because it generally works under all situations.

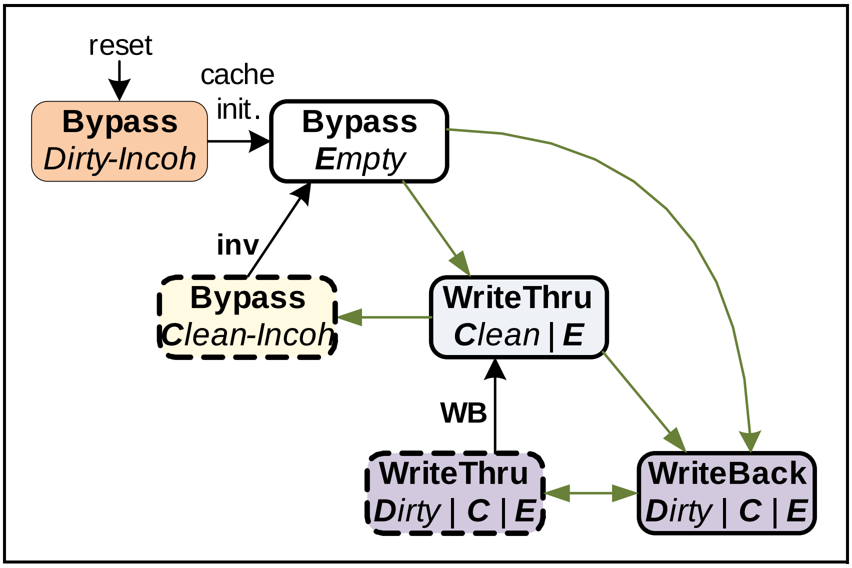

Figure 3–3. Cache State Diagram – with Uninterrupted Memory Accesses

The above discusses the data cache. The same can be applied to the instruction cache by dropping WriteBack and Dirty states, and WB and W transitions.

The following subsections elaborate on specific transitions through the above diagrams. They present the most common relevant scenarios and their general solution. One may be able to optimize this for specific cases. All subsections but the last assume a one-to-one mapping from virtual to physical memory space. In this discussion, write-through mode refers to both write-through mode on the data cache and cached mode on the instruction cache, whereas writeback mode applies only to data caches.

After Reset

The state of the caches is undefined at reset. To avoid cache coherency issues (and performance issues when cache lines are improperly left locked), instruction and data caches must be completely unlocked (if cache locking is configured) and invalidated before use, *i.e.* before any access to external memory using a cache attribute that enables the cache. This is typically done in reset code before setting cache attributes that enable the cache.

From Bypass to Write-Through or Write-Back

Asa general rule, before enabling the cache for a given memory region (such as changing the cache attribute from bypass mode to write-through or writeback mode), any cached version of that memory must be invalidated. This is true even if caching was previously enabled then disabled: if you enable, disable, then re-enable the cache for a memory region, the cache contents over that memory region must generally be invalidated *before* re-enabling the cache.

More precisely, the cache must be coherent with (its contents must match) that memory region. If any stores were made to that memory region while in bypass mode (or if the cache was never initialized), any cached copy of that memory still present in the cache is no longer coherent and needs to be invalidated before enabling caching for that memory region.

If multiple virtual address regions map to the same physical address region, the above applies when the physical region starts being cached: typically, when all mappings use bypass mode, and at least one of them is changing to a cached mode.

From Write-Through to WriteBack

No special step is needed to change a cache attribute from write-through mode to writeback mode, or to switch between writeback-allocate and writeback-no-allocate.

From WriteBack to Write-Through

No special step is needed to change the cache attribute for a memory region from writeback to write-through mode.

However, such a transition is often meant as a step towards ensuring that whatever part of that memory region exists in the cache is *clean*, that is, matches the contents of memory as opposed to having *dirty* cache entries (created only in writeback mode) whose changed contents are not yet written back to memory.

To that end, a *writeback* (or clean) cache operation over that region is necessary. Otherwise, dirty entries may remain indefinitely after switching from writeback to write-through mode, as they are automatically cast-out (written back to memory) only as needed, *such* as to make space for other data being read.

To avoid any such dirty cache entries once in write-through mode, the relevant cache entries must be written back to memory *after* changing the cache attribute to write-through. However, if it is possible to suspend all writes to that memory region during the cache mode transition (including writes by other threads, interrupt handlers, exception handlers including window overflows, compiler reordered stores, and so forth), the writeback operation may be done either before or after the transition, as follows:

1. Suspend writes to the memory region
2. Do the following in any order:
   - Invoke writeback cache operation over the region
   - Switch the region’s cache mode from writeback to write-through
3. Resume writes to the memory region

If multiple virtual address regions map to the same physical address region, the above applies when the physical region stops being used in writeback mode: for example, when most mappings use writethrough mode except for one that uses writeback, which transitions to writethrough mode like the rest.

From Write-Through to Bypass

Changing a cache attribute from write-through mode to bypass, thus disabling the cache for that region of memory, requires no special step as long as no dirty cache entries (from a previous use of write-back mode) exists over that region of memory. See the previous subsection.

Note that, as described above (From Bypass to Write-Through or Write-Back), cache entries covering that region of memory must be invalidated before re-enabling the cache to prevent cache hits on stale cache contents.

From WriteBack to Bypass

This cache mode transition generally proceeds along one of two sequences.

If it is possible to suspend all writes to the affected memory region during the transition (including writes by other threads, interrupt handlers, exception handlers including window overflows, compiler reordered stores, and so forth), the writeback operation may be done either before or after the transition, as follows:

1. Suspend writes to the memory region
2. Do the following in any order:
   - Invoke writeback cache operation over the region
   - Switch the region’s cache mode from writeback to write-through
3. Resume writes to the memory region

It is not safe to change a cache attribute directly from write-back mode to bypass mode. The cache may contain dirty data, *i.e.,* data not yet written to memory, which becomes inaccessible in bypass mode. Furthermore, any such dirty data may subsequently get written back to memory if the data cache is enabled over any other memory region (*e.g.* to allocate other cache lines). It is not sufficient to write-back dirty data while in writeback mode, in fact this is not recommended. As noted above in *From WriteBack to Write-Through*, new dirty data can easily be created during and after writing-back dirty data and before changing the cache attribute.

It is thus necessary to change to write-through mode as an intermediate step. Changing a cache attribute from write-back mode to bypass mode essentially involves following the steps described above to change from write-back mode to write-through mode, then from write-through mode to bypass mode.

In summary, the sequence is:

- change the cache attribute to write-through mode,
- write-back the cache contents to memory, and
- change the cache attribute to bypass mode.

Optionally, you can then invalidate any cache entries covering that region of memory in anticipation of re-enabling the cache later on over that area of memory.

Changing To and From Illegal Access Modes

Certain cache attributes prevent any access to memory, causing an exception if such an access is attempted. Here we call such attributes *illegal access modes*.

No special steps are necessary to change a cache attribute to an illegal access mode, or from an illegal access mode to the previous access mode (*i.e.*, back to the same access mode as was in effect before changing to an illegal access mode). Note that if any portion of the memory now protected by an illegal access mode was cached, such cached entries may still get written back to memory (if dirty) or invalidated if there is any other cache activity over other memory regions.

However, changing from cache attribute A to an illegal access mode, then to cache attribute B, where A and B are different cache attributes, is effectively the same as changing a cache attribute from A to B for the purpose of this discussion. In that case, follow the appropriate recommendation for such a change to the cache attribute.

When Multiple Virtual Addresses Map to the Same Physical Address

In processor configurations that support programmable virtual to physical address mapping, multiple virtual addresses can map to the same physical address. Because cache attributes are specified for each virtual address range, this means a given physical memory range may be accessed by different cache attributes depending on the virtual address used.

To ensure cache coherency for such a range of physical memory within a *single* processor[9], all cache attributes of virtual memory ranges used to access that physical memory must be compatible. Illegal access modes are always compatible with any other cache attribute, because their corresponding virtual memory range is effectively unused. Write-back and write-through cache modes are also compatible with each other.

However, bypass (cache disabled) mode is incompatible with write-back or write-through (cache enabled) mode. If a region of physical memory is accessed using both bypass and cache-enabled modes, coherency is not maintained automatically. For example, a write in bypass mode may not be seen when read in cache-enabled mode if that address was cached before the store. Or, a write in write-back mode may not be seen when read in bypass mode if the dirty data has not yet been written back; and a subsequent write to the same address in bypass mode may be overwritten by dirty data cached earlier.

It is often possible to deal with these issues by careful use of cache control instructions or their corresponding HAL functions. However, unless the relevant accesses are few and occur at well-defined points, this approach can significantly complicate application code and is error-prone.

The most common or typical uses of cache control instructions and functions are to ensure cache coherency for memory shared with other processors, for memory shared with other bus masters (such as in drivers for DMA capable devices), or to ensure coherency between the instruction and data caches.

3.12 The Memory Protection Unit (MPU)

The Xtensa processor Memory Protection Unit (MPU) configuration option is present **only** on specific pre-configured cores. The MPU provides access control and cache attribute control, at a much finer granularity than the 512MB regions supported by the Region Protection option. The MPU only provides memory access control, and does not provide any form of address translation. The memory map for the MPU is specified as a configuration time fixed background map, and a dynamically changeable foreground map.

Both the foreground and background maps are composed of a vector of entries. The vector length is fixed during the configuration. Each entry consists of the following fields:

9. Ensuring cache coherency among multiple processors is beyond the scope of this section. It typically involves the use of cache invalidate and write-back operations, or alternatively avoiding the use of caches, or using hardware cache coherency support (contact Cadence for the latter).

- `vStartAddress`: An unsigned integer, which specifies the starting virtual address for the region. The number of significant bits is specified by the architectural constant `XCHAL_MPU_ALIGN_BITS`. The remaining bits should be zero.
  Note that the foreground and background maps are ordered so that the `vStartAddress` of the entries are monotonically increasing.
- `valid`: A 1-bit field, which specifies whether this foreground entry is active. This field is only applicable to foreground entries.
- `accessRights`: A 4-bit field, which specifies the access rights for the region. Various combinations of read, write, execute, kernel, and user permissions are coded into these four bits. Note that the MPU does not support every combination.
- `memoryType`: A 9-bit field, which encodes the cache and sharing attributes of the memory region. Note that the MPU does not support every combination.

Note that no explicit specification of the length of the region or end address for the region is given. Instead, the region ends at `vStartAddress` - 1 of the next entry. The region defined by the last entry extends to the end of the address space, at `0xFFFFFFFF`.

The access rights and memory type for a memory access at address `vaddr` are determined by finding the highest foreground entry whose `vStartAddress` is less than or equal to `vaddr`. If that entry is valid, then the memory type and access rights are determined by that entry. If that entry is not valid or if no foreground entry has a `vStartAddress` that is less than or equal to `vaddr`, the memory type and access rights are taken from the highest background map entry whose `vStartAddress` is less than or equal to `vaddr`. The first background map entry always has a `vStartAddress` of 0, so every address is mapped.

On reset, all the foreground entries are invalid, thus the background map is used as the memory map.

3.12.1 C Structure Types

xthal_MPU_entry

The type `xthal_MPU_entry` represents an MPU entry and is used by the functions that accept or return MPU entries. The fields of the `xthal_MPU_entry` are accessed by macros:

```
XTHAL_MPU_ENTRY(vStartAddress, valid, accessRights, memoryType)
```

The `XTHAL_MPU_ENTRY` expands into the C initialization for the MPU entry. This is useful for constructing an MPU entry, or an entire map of entries. For example, the following code constructs an MPU map with three entries:

```
const struct xthal_MPU_entry mpumap[] =
{
XTHAL_MPU_ENTRY( 0x00000000, 1, XTHAL_AR_RWXrwx, XTHAL_MEM_WRITEBACK),
XTHAL_MPU_ENTRY( 0xE0000000, 1, XTHAL_AR_RWXrwx,
XTHAL_MEM_NON_CACHEABLE,
XTHAL_MPU_ENTRY( 0xF0000000, 1, XTHAL_AR_RWX, XTHAL_MEM_NON_CACHEABLE)
};
```

These macros get and set the corresponding field in the xthal_MPU_entry structure, entry:

```
XTHAL_MPU_ENTRY_GET_VSTARTADDR(entry)
XTHAL_MPU_ENTRY_SET_VSTARTADDR(entry, vStartAddress)
XTHAL_MPU_ENTRY_GET_VALID(entry)
XTHAL_MPU_ENTRY_SET_VALID(entry, valid)
XTHAL_MPU_ENTRY_GET_ACCESS(entry)
XTHAL_MPU_ENTRY_SET_ACCESS(entry, accessRights)
XTHAL_MPU_ENTRY_GET_MEMORY_TYPE(entry)
XTHAL_MPU_ENTRY_SET_MEMORY_TYPE(entry, memoryType)
```

3.12.2 Core Specific Constants

XCHAL_HAVE_MPU

The `XCHAL_HAVE_MPU` global constant reports if an MPU is configured.

XCHAL_MPU_ENTRIES

The `XCHAL_MPU_ENTRIES` global constant reports the number of foreground entries in the MPU.

XCHAL_MPU_BACKGROUND_ENTRIES

The `XCHAL_MPU_BACKGROUND_ENTRIES` global constant reports the number of background entries in the MPU.

XCHAL_MPU_ALIGN_REQ

The `XCHAL_MPU_ALIGN_REQ` constant indicates if the processor configuration requires alignment between the foreground and background entries. If `XCHAL_MPU_ALIGN_REQ` is non-zero, the correct MPU foreground maps must meet the following constraints:

- If `entry[0]'s vStartAddress` is not zero and `entry[0]'s valid bit is not zero`, then `entry[0]'s vStartAddress` must equal the `vStartAddress` of one of the background map's entries.

- If `entry[N]'s valid bit is zero, and entry[N+1]'s valid bit is one,` then `entry[N+1]'s vStartAddress` must equal the `vStartAddress` of one of the background map's entries.

The `xthal_mpu_set_region_attribute()` function automatically adds foreground entries corresponding to the background entries when necessary to meet those constraints. Users of `xthal_write_map()` or `xthal_write_map_raw()` must obey those constraints if `XCHAL_MPU_ALIGN_REQ` is non-zero.

XCHAL_MPU_ALIGN

The `XCHAL_MPU_ALIGN` constant is the minimum alignment value for the MPU region boundaries. The minimum value for this constant is 32, but it can be larger on some processor configurations.

XCHAL_MPU_ALIGN_BITS

The `XCHAL_MPU_ALIGN_BITS` constant is $\log_2$(`XCHAL_MPU_ALIGN`), which has a value greater than or equal to five.

3.12.3 Access Rights

Table 3–20 lists the supported MPU access rights, which are encoded into four bits. Note that every combination of access rights is not supported. The kernel and user modes have separate rights. The kernel rights are in effect after reset unless and until the processor is switched to the user mode. Typically, the user mode rights only apply to processes running under an operating system that supports separate access rights for user and kernel code.

If an access is attempted that is not allowed by the access rights, an exception is generated.

Table 3–20. Access Rights Constants

Access Rights Constant	Access Rights
`XTHAL_AR_NONE`	No access
`XTHAL_AR_R`	Kernel read, User no access
`XTHAL_AR_RX`	Kernel read/execute, User no access
`XTHAL_AR_RW`	Kernel read/write/execute, User no access
`XTHAL_AR_RWX`	Kernel read/write/execute, User no access
`XTHAL_AR_Ww`	Kernel write, User write
`XTHAL_AR_RWrwx`	Kernel read/write, User read/write/execute

Table 3–20. Access Rights Constants (continued)

Access Rights Constant	Access Rights
`XTHAL_AR_RWr`	Kernel read/write, User read
`XTHAL_AR_RWXrx`	Kernel read/write/execute, User read/execute
`XTHAL_AR_Rr`	Kernel read, User read
`XTHAL_AR_RXrx`	Kernel read/execute, User read/execute
`XTHAL_AR_RWrw`	Kernel read/write, User read/write
`XTHAL_AR_RWXrwx`	Kernel read/write/execute, User read/write/execute

3.12.4 Memory Type

The following groups of constants combine using bitwise OR to specify the memory type to macros and functions that accept a memory type specifier. The MPU Memory Type Field is documented in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* section titled Memory Protection Unit Option.

The constants listed in Table 3–21 specify the *cacheability* portion of the memory type. One of these constants is always required when specifying a memory type (all other constants are optional). By default, they specify both the external (beyond the memory interface) and local (processor) cacheability. Optionally, they can be combined with a macro to specify independent external and local cacheability as:

```
        XTHAL_MEM_PROC_CACHE(ext,loc)
```

where *`ext`* and *`loc`* are each one of the cacheability constants other than `XTHAL_MEM_DEVICE`.

Table 3–21. Cacheability Constants

Cacheability Constant	Description
`XTHAL_MEM_DEVICE`	Memory mapped device
`XTHAL_MEM_NON_CACHEABLE`	Memory is not cacheable
`XTHAL_MEM_WRITETHRU_NOALLOC`	Write-through caching, no new cache lines are allocated
`XTHAL_MEM_WRITETHRU_WRITEALLOC`	Write-through caching, new cache lines are allocated on both reads and writes
`XTHAL_MEM_WRITETHRU`	Write-through caching, new cache lines are allocated on reads, but not writes
`XTHAL_MEM_WRITEBACK_NOALLOC`	Write-back caching, new cache lines are allocate only on reads
`XTHAL_MEM_WRITEBACK`	Write-back caching, new cache lines are allocated on both reads and writes

The memory type can be further specified by combining (again with bitwise OR) one or more of the flags from Table 3–22.

Table 3–22. Other Memory Type Flags

Memory Type Flag	Description
`XTHAL_MEM_INTERRUPTIBLE`	Indicates that reads are interruptible, and may thus be repeated. Only applicable to device memory type, for which reads are normally non-interruptible; reads are interruptible with every other memory type. May improve worst-case interrupt latency, however may not be used to access devices with read side-effects.
`XTHAL_MEM_BUFFERABLE`	Indicates that writes to this memory region are bufferable: a response may be issued before the write reaches its final destination. Only applicable to device and non-cacheable memory types. This is generally used to improve performance. The system (outside the core) is responsible for maintaining memory ordering. However, it may be more difficult to order writes to such a region with respect to non-memory -mapped events (such as interrupts, TIE interfaces, and so on): the S32NB instruction may be used to make specific writes non-bufferable.

The constants listed in Table 3–23 specify the *shareability* portion of the memory type. This specifies whether and how far memory may be shared with other processors in the system. This distance is defined in terms of inner, outer and system domains, whose scopes depend on system configuration. The inner domain is typically a local group of processors. It is a subset of (or equal to) the outer domain, which is a subset of (or equal to) the system domain. The system domain typically includes all cores that can reach a given memory or memory-mapped device. Note that shareability is specified at the processor core level: multiple threads on the same processor all count as one core, and thus may always share memory (to the degree the software environment facilitates it).

Table 3–23. Shareability Constants

Shareability Constants	Description
`XTHAL_MEM_NON_SHARED`	Indicates that the memory is not shareable. Memory transactions may be optimized assuming only the local processor accesses this memory. This is the default for memory types other than `DEVICE`, and is not applicable to `DEVICE` type, so providing this flag has no effect -- it is present only for code readability.
`XTHAL_MEM_INNER_SHAREABLE`	Indicates that the memory is shareable between processors in the inner domain.Only applicable to cacheable memory types.
`XTHAL_MEM_OUTER_SHAREABLE`	Indicates that the memory is shareable between processors in the outer domain.Only applicable to cacheable memory types
`XTHAL_MEM_SYSTEM_SHAREABLE`	Indicates system wide sharing. It is the only allowed (and thus assumed) shareability for `DEVICE` memory type. Generally only applicable with non-cacheable memory type (`XTHAL_MEM_NON_CACHEABLE`).

The following examples show various memory types:

- `XTHAL_MEM_WRITEBACK`

  Non-shareable memory with write-back caching enabled.

- `XTHAL_MEM_PROC_CACHE(XTHAL_MEM_WRITEBACK, XTHAL_MEM_WRITETHRU) | XTHAL_MEM_INNER_SHAREABLE`

  Memory accesses may be write-back cached externally, but are only write-through cacheable on the processor. Memory may be shared among processor cores within the inner domain.

- `XTHAL_MEM_DEVICE | XTHAL_MEM_INTERRUPTIBLE`

  Device memory type, always shareable across the system. Reads from memory of this type may be interrupted, and thus repeated, appropriate for devices with no read side-effects.

3.12.5 Variables

Xthal_mpu_bgmap

The variable `Xthal_mpu_bgmap` contains the processor's background map represented as a constant vector of MPU entries. The length of the vector is `XCHAL_MPU_BACKGROUND_ENTRIES.`

```
extern const xthal_MPU_entry Xthal_mpu_bgmap[];
```

3.12.6 Functions

Access Rights Query

These functions return a one if the supplied `accessRights` argument meets the condition implied by the function name, otherwise return a zero:

```
extern int32_t xthal_is_kernel_readable(int32_t accessRights);
extern int32_t xthal_is_kernel_writeable(int32_t accessRights);
extern int32_t xthal_is_kernel_executable(int32_t accessRights);
extern int32_t xthal_is_user_readable(int32_t accessRights);
extern int32_t xthal_is_user_writeable (int32_t accessRights);
extern int32_t xthal_is_user_executable(int32_t accessRights);
```

xthal_encode_memory_type

The `xthal_encode_memory_type` function converts a bitwise combination of the `XTHAL_MEM_` constants to the corresponding MPU memory type (9-bits). If no `XTHAL_MEM_` constants are present in the argument, then bits[12:4] (nine-bits) of x are returned. This supports using an encoded memory type (perhaps obtained from an `xthal_MPU_entry` structure) as an input to `xthal_mpu_set_region_attribute()` or `xthal_set_region_attribute()`.

This function checks that the supplied constants are a valid and supported combination. If not, it returns `XTHAL_BAD_MEMORY_TYPE`.

```
extern int32_t xthal_encode_memory_type(uint32_t x);
```

The `XTHAL_ENCODE_MEMORY_TYPE` macro provides the equivalent functionality, except that no error checking is done on its arguments:

```
XTHAL_ENCODE_MEMORY_TYPE(x)
```

Memory Type Query

The memory type query functions accept an encoded memory type, and return a one if the supplied memory type has a property specified by the function name.

```
extern int32_t xthal_is_cacheable(uint32_t memoryType);
extern int32_t xthal_is_writeback(uint32_t memoryType);
extern int32_t xthal_is_device(uint32_t memoryType);
```

xthal_read_map

The `xthal_read_map` function copies the current MPU foreground map to `entries`, which must point to available memory of at least `sizeof(xthal_MPU_entry) * XCHAL_MPU_ENTRIES` bytes.

If successful, this function returns `XTHAL_SUCCESS`, otherwise it returns the error code `XTHAL_INVALID` or `XTHAL_UNSUPPORTED`.

```
extern int32_t xthal_read_map(xthal_MPU_entry* entries);
```

xthal_write_map

xthal_write_map_raw

The `xthal_write_map()`and `xthal_write_map_raw()` functions write the map pointed to by `entries` to the MPU. The `xthal_write_map()` function updates the cache and CACHEADRDIS register if needed, while the `xthal_write_map_raw()` function only updates the MPU map.

The `xthal_write_map()` function commits any uncommitted cache writes, and invalidates the cache if necessary before updating the MPU.The `xthal_write_map_raw()` function does not initiate any cache ops.

The `xthal_write_map`() function updates the CACHEADRDIS register if needed. On LX7 processors, the `CACHEADRDIS` register will be set to enable caching on each 512 MB region where at least some portion of that region is cacheable. Caching will be disabled if none of the 512 MB region is cacheable. The `CACHEADRDIS` register is only present on LX7 hardware.

Note: This function disables the MPU foreground map during the MPU update and relies on the background map. As a result, the `xthal_write_map()` or `xthal_write_map_raw()` functions must be readable and executable in the background map, and the caller must ensure the following before calling this function, to avoid asynchronous use of the MPU during the call:

- All interrupts must be disabled.
- The Integrated DMA engine (iDMA), if configured, must be idle.
- All previous block prefetch requests must be complete.

This `xthal_write_map()` does not check the map for correctness. Generally `xthal_check_map()` should be called first to check the map.

If `num_entries` equals zero, the existing map is cleared and no new map is written.The foreground entries are not completely zeroed out, but valid bit of each foreground entry is cleared, and the foreground map is completely disabled.This is useful for returning to the reset state.

If `num_entries` is greater than zero, but less than `XCHAL_MPU_ENTRIES`, a new map is written with padding entries added to ensure a properly ordered map. The position of the padding entries should not be relied upon.

If num_entries is equal to XCHAL_MPU_ENTRIES, the complete MPU map is replaced by the supplied map.

```
   extern void xthal_write_map(const xthal_MPU_entry* entries,
   uint32_t num_entries);

   extern void xthal_write_map_raw(const xthal_MPU_entry* entries,
   uint32_t num_entries);
```

xthal_check_map

The xthal_check_map function checks whether a vector of MPU entries is a correct MPU map. It checks that the entries are correctly ordered by their vStartAddress, that the entries are aligned to the background map if necessary, and that the access rights for each entry are correct.

If the map is valid, it returns XTHAL_SUCCESS, otherwise it returns one of the following error codes:

XTHAL_MAP_NOT_ALIGNED, XTHAL_BAD_ACCESS_RIGHTS, XTHAL_OUT_OF_ENTRIES, XTHAL_OUT_OF_ORDER_MAP, or XTHAL_UNSUPPORTED.

```
   extern int32_t xthal_check_map(const xthal_MPU_entry* entries,
   uint32_t num_entries);
```

xthal_get_entry_for_address

The xthal_get_entry_for_address function returns the MPU entry that maps vaddr. If infgmap is non-NULL, *infgmap is set to one when vaddr is mapped by the foreground map, or *infgmap is set to zero when vaddr is mapped by the background map.

```
   extern xthal_MPU_entry
             xthal_get_entry_for_address(void* vaddr, int32_t infgmap);
```

xthal_mpu_set_region_attribute

The xthal_mpu_set_region_attribute function sets the memory type and access rights for a region of memory specified by vaddr and size. This function is specifically for use with the MPU and is called by xthal_set_region_attribute() for processors with an MPU configured. Refer to xthal_set_region_attribute() in Section 3.11.3 "Cache Region Control Functions" on page 129 for information on the flags argument and the error codes that can be returned.

This function typically moves, adds, and merges entries from the MPU map during execution; Thus, the resulting map is different from the map before the function call. This function makes the following guarantees when vaddr and size are aligned to XCHAL_MPU_ALIGN:

- The MPU access map remains in a valid state at all times during its execution.
- At all times, during and after function completion, the memory type and access rights remain the same for all addresses that are not in the range `[vaddr, vaddr+size-1]`.
- If `XTHAL_SUCCESS` is returned, the range `[vaddr, vaddr+size-1]` has the access rights and memory type specified.

Depending on the MPU's state, this function requires from zero to three unused MPU entries.

On LX7 processors, the `CACHEADRDIS` register will be set to enable caching on each 512 MB region where at least some portion of that region is cacheable. Caching will be disabled if none of the 512 MB region is cacheable. The `CACHEADRDIS` register is only present on LX7 hardware.

The `accessRights` parameter should be either a 4-bit value corresponding to an MPU access mode (as defined by the `XTHAL_AR_` constants), or `XTHAL_MPU_USE_EXISTING_ACCESS_RIGHTS`. If `XTHAL_MPU_USE_EXISTING_ACCESS_RIGHTS` is supplied, the current access rights for `vaddr` are used for the new region.

The `memoryType` parameter should be either a bitwise-OR combination of `XTHAL_MEM_` constants as described in Section 3.12.4, a 9-bit MPU memory type value, or `XTHAL_MPU_USE_EXISTING_MEMORY_TYPE`. If `XTHAL_MPU_USE_EXISTING_MEMORY_TYPE` is supplied, the current memory type for `vaddr` is used for the new region.

The `flags` parameter should be a bitwise combination of the flags (0 if none) listed inTable 3–24.The flags are defined in the header <xtensa/hal.h>.

Table 3–24. xthal_mpu_set_region_attribute() flags

Flag	Description
`XTHAL_CAFLAG_EXACT`	Return error if the requested attribute cannot be applied to the exact range specified
`XTHAL_CAFLAG_NO_PARTIAL`	Only apply the requested attribute to regions completely covered by the specified range; don't modify regions only partially covered by the specified range
`XTHAL_CAFLAG_NO_AUTO_WB`	Do not automatically writeback dirty data when the memory type is changed from a writeback type to a non-writeback type
`XTHAL_CAFLAG_NO_AUTO_INV`	Do not automatically invalidate cache when the memory type is changed from a cacheable type to a non-cacheable type.

```
extern int32_t xthal_mpu_set_region_attribute(void* vaddr, uint32_t
size, int32_t accessRights, int32_t memoryType, uint32_t flags);
```

xthal_calc_cacheadrdis

On LX7 processors, `xthal_calc_cacheadrdis` scans the supplied MPU map and returns a value suitable for writing to the `CACHEADRDIS` register: Bits 0-7 are 1 if there are no cacheable areas in the corresponding 512 MB region or 0 otherwise. Bits 8-31 are undefined. On non-LX7 processors, the return value is undefined.

```
extern uint32_txthal_calc_cacheadrdis (const struct xthal_MPU_entry*e,
uint32_t n);
```

xthal_mpu_set_entry

The `xthal_mpu_set_entry` function updates a single entry of the MPU. This is useful to update the memory type or access rights of an existing MPU region. The caller must ensure that the resulting MPU map is properly ordered.

Typically, the entry argument is obtained by modifying the access rights and/or the memory type of an MPU entry obtained by calling `xthal_get_entry_for_address()`.

```
#include <xtensa/core-macros.h>
inline void xthal_mpu_set_entry (xthal_MPU_entry entry);
```

3.13 Debug Features

The Xtensa architecture optionally provides hardware support for debugging. Again, it is not the intent of the link-time HAL to functionally abstract these features. Instead the HAL describes the features so that the system software designer can write a configuration runtime. The debug features really only have three parameters: Is debug configured? How many instruction breakpoints exist? And how many data breakpoints exist?

The link-time HAL also provides routines that can be used to assist a debug agent running on a target. These routines help set and remove breakpoints.

3.13.1 Core Specific Constants

XCHAL_HAVE_DEBUG

This value is 0 if the debug option is not configured in the processor and 1 if it is.

```
extern const int32_t xthal_debug_configured; // 0 if not, 1 if so
```

XCHAL_NUM_IBREAK

This reports the number of hardware breakpoint registers configured in the processor.

```
extern const int32_t xthal_num_ibreak;  // number of ibreak registers
```

XCHAL_NUM_DBREAK

This reports the number of data address match registers configured in the processor.

```
extern const int32_t xthal_num_dbreak;  // number of dbreak registers
```

XCHAL_BYTE0_FORMAT_LENGTHS and Xthal_byte0_format_lengths

The length of an Xtensa instruction can always be determined from its first byte. `Xthal_byte0_format_lengths[]` is a 256-entry byte array that provides the length in bytes of an instruction when indexed by the value of its first byte.

`XCHAL_BYTE0_FORMAT_LENGTHS` is a pre-processor macro whose value is a comma-separated 256-entry list of instruction lengths. It can be used, for example, to initialize an array such as `Xthal_byte0_format_lengths`, like this:

```
const uint8_t fmt_lengths[256] = { XCHAL_BYTE0_FORMAT_LENGTHS };
```

For values of the first byte, if any, that have no associated instruction encoding, the array reports one of the valid instruction sizes (which one exactly is not defined) for the specific processor configuration. As of this writing, the Xtensa architecture supports not only standard instruction sizes of 2 bytes (16 bits, density option) and 3 bytes (24 bits, most instructions), but also many other larger instruction sizes used for the FLIX feature.

Note: The value of this array is a function of Xtensa processor extensions defined by the designer using the TIE language. The TIE compiler does not update the HAL header files and libraries. This array only reflects the processor options and extensions provided to the Xtensa Processor Generator, not local changes made using the TIE compiler.

```
extern const uint8_t Xthal_byte0_format_lengths[256];
```

XCHAL_OP0_FORMAT_LENGTHS and Xthal_op0_format_lengths

The `Xthal_op0_format_lengths[]` array and `XCHAL_OP0_FORMAT_LENGTHS` preprocessor macro are similar to the `Xthal_byte0_format_lengths[]` array and `XCHAL_BYTE0_FORMAT_LENGTHS` macro described above. However, they are not available for every Xtensa processor. They contain 16 entries instead of 256. They are used to determine the length of an Xtensa instruction from its first byte's 4-bit `op0` field (upper 4 bits for big-endian processors, and lower 4 bits for little-endian processors). The length of an instruction on an LX4 (RD-201x.x release) or older Xtensa processor can always be determined from this 4-bit `op0` field. Later Xtensa processors may use the entire first byte to decode instruction length: where this can only be represented using the longer 256-entry arrays, the shorter 16-entry arrays are not present.

These shorter arrays are now deprecated. They may be removed in a future release.

```
extern const uint8_t Xthal_op0_format_lengths[16];
```

3.13.2 Functions

xthal_set_soft_break

The following routine is used for setting up software breakpoints. In addition to writing the appropriate break instruction to the appropriate place, they synchronize the caches so that an instruction fetch of the BREAK instruction will not contain the previous I-cache value. The `xthal_set_soft_break()` routine returns a word containing the instruction that was replaced by the breakpoint, possibly in an encoded form that includes the size of the instruction. The corresponding `xthal_remove_soft_break()` routine requires the same address of the breakpoint as well as the word returned by this routine.

Note: The exact encoding of the word returned by `xthal_set_soft_break()` is subject to change. Future implementations of this function may plant a breakpoint instruction that is smaller than the instruction being overwritten, and return only the replaced portion of the overwritten instruction rather than its entirety. The value returned by `xthal_set_soft_break()` must be treated as a "magic cookie" that is interpreted properly only by the companion `xthal_remove_soft_break()` function.

```
uint32_t xthal_set_soft_break(void *addr);   // set sw breakpoint
```

xthal_remove_soft_break

This function removes a breakpoint planted with `xthal_set_soft_break()`. The value passed in parameter *inst* is the value returned by `xthal_set_soft_break()`.

```
void xthal_remove_soft_break(void *addr, int32_t inst);
```

3.14 Miscellaneous Functions

xthal_clear_regcached_code

This function clears any code or PCs cached in registers for the current thread. It needs to be called when code that may have previously been executed as part of this thread is unloaded and replaced (in overlapping memory locations) with new code that may be called by this thread.

```
    void xthal_clear_regcached_code(void);
```

Note that, unlike the instruction cache which is shared among all threads and thus easily invalidated at the point of loading (or unloading) code, registers that contain code or PCs need more involvement of the application developer. Each thread's registers are saved in some OS-specific manner, and cannot readily be accessed by the code loading process in a generic manner. Also, practically no OS has any framework to deal with this issue in their architecture porting layer. So, solving this issue at the point of loading code involves porting work for every OS with which that approach is used. The alternative is to have the application call `xthal_clear_regcached_code()` as described in the previous paragraph.

For the Xtensa architecture, the only registers that cache code or PC are the zero overhead loop registers: `LBEGIN`, `LEND`, and `LCOUNT`. It is legal to exit a loop early, leaving these registers active — with `LCOUNT` non-zero such that falling through `LEND` brings PC to `LBEGIN` — because from that point the fall-through to `LEND` is usually only reached by re-entering the loop which re-initializes the loop registers. The compiler generates such code, for example. The potential issue occurs when such a loop is left active, another thread loads new code where the function that contained this loop was located, and the thread with the stale active loop now calls the new code which proceeds to unexpectedly branch from PC=`LEND` to PC=`LBEGIN` in an area of the new code that does not contain the original zero overhead loop.

Thus, as of this writing, `xthal_clear_regcached_code()` simply clears `LCOUNT`, which disables the zero overhead loopback mechanism. Thus, there is no need to call this function unless the zero overhead loop option is configured in the processor. Code written to be portable across multiple Xtensa processors must always call the function where appropriate, rather than rely on knowledge of the zero overhead loop option, because future implementations may introduce other options with similar properties.

4. Library Loader

This chapter documents a library and tool flow that allows dynamic loading and unloading of libraries on an Xtensa processor without the aid of an operating system. This type of library, known as a “loadable library”, is available as two different types: a fixed-location overlay and a position-independent library.

With the first type, a fixed-location overlay, you can load code and data into predetermined locations in memory, similar to an overlay in other systems. With the second type, a position-independent library, you can load code and data at addresses determined at runtime. Each type has advantages and limitations that are described in this chapter. The type best suited for a particular library depends on the application's needs. Further, a single application can load libraries of both types, although a specific library can only be used as a single type.

Loadable libraries do not share name-spaces with the main program or each other. Thus a library and the main program could both have a function named `foo`, each copy of which does a completely different thing. Although it is possible for a library to call functions and refer to variables inside the main program (and vice-versa), doing so requires special handling as described below. Loadable libraries should therefore be largely self-contained.

Users typically profile their code prior to packaging it into a loadable library.

You link your library separately from the main application, package the library into a suitable form, and then, when linking your main application, link the packaged library into the main application. You link the loadable library using a special LSP named `piload` or `pisplitload` (see the *Xtensa Linker Support Packages (LSPs) Reference Manual*). Note that this loadable-library LSP almost never provides Reset or Debug vectors or any other fixed-location code —the main application provides that. If the library provides an interrupt service routine, the main application should obtain its address and do the necessary work to get it called (such as registering it with XTOS), at load time.

This chapter briefly describes the advantages and disadvantages of each type of library, then documents the flow to generate and use each type. Various source code and examples accompany this chapter. You can find them in the following directory:

```
<xtensa_root>/examples/LibraryLoader
```

The fixed-location library examples work unmodified on any Xtensa configuration that contains both an instruction RAM and a data RAM. The position-independent library example requires a system memory (accessible over the PIF interface).

Simple, but complete, source code and tests are in the `LibraryLoader` directory. To run the position-independent library test, use `xt-make test_pil`. To run the position-independent library test with separate code and data loading, use `xt-make test_pilsl`. To run the fixed-location overlay test, you must first build the custom LSPs, as described in Section 4.5. Then you can use `xt-make test_overlay` to run the test.

4.1 Requirements

Loadable libraries require standard, PC-relative L32R instructions, and will not work with the Extended L32R software option.

As this chapter deals with advanced linking concepts, you should be familiar with the linker support packages (LSPs), as described in the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

4.2 Fixed-Location Overlays

A video player might need to run an MPEG-4 decoder to play one video-stream, and then run an H.264 decoder to play back the next, but it will never need to do both of these at once. Because codecs have high-performance requirements and will never run at the same time, it makes sense to put both code and data into an Xtensa processor's local memories, which are quite fast. However, these memories tend to be small, and possibly too small to hold both codecs. With fixed-location overlays, you can link both codecs at the same location in an instruction memory, and load and unload them as required. Fixed location overlays also work in normal system RAM.

Advantages

- Code and data size: fixed-location libraries tend to have smaller size than position-independent though the overall difference in size might be small.
- Performance: A fixed-location library is marginally faster than a position-independent library.

Disadvantages

- Memory Partitioning: Memory is reserved for the loadable libraries very early in the tool flow, making it more difficult to use every available byte of memory in the system. Further, the tool flow documented here only allows you to load one fixed-location library at a time. Loading multiple fixed-location libraries at the same time is possible by partitioning memory into multiple segments, but that is beyond the scope of this chapter.

- LSP Editing: Using fixed-location overlays requires the generation of two custom LSPs: One for the main program and one for the overlays. Although this is not difficult, it is an extra step.

4.3 Position-Independent Libraries

A position-independent library can be loaded and run at any address that supports both code and data, like normal system RAM. Alternatively you can use the `pisplitload` LSP to load code and data into separate memory blocks located in local RAMs. Library location need not be decided until run time. Because their locations are not fixed, a variety of position-independent libraries can be loaded at the same time, without deciding at build time which sets are possible. For example, if you have five codecs, and three can be loaded at any one time, then they could be loaded in ten different combinations, which are too many to feasibly link in fixed-locations. Position-independent libraries solve that problem by allowing the codecs to be loaded in arbitrary locations, in arbitrary orders.

Advantages

- Flexibility: Several dynamic location libraries can be loaded in arbitrary combinations at arbitrary locations, all of which is decided at runtime.

Disadvantages

- Performance: A position-independent library must be compiled as position-independent code, which is marginally slower than position-dependent code.

4.4 Library Loader API

This section describes the functions that can be called from the main program to load and unload both types of libraries. To use these functions and the associated data structures, the application must include `xt_library_loader.h`.

```
#include <xt_library_loader.h>
```

Both sets of functions are declared in the same header file.

4.4.1 Fixed Location Overlay API

There is a newer version of the API used to manage fixed location overlays. This will be described first. The newer API handles library initialization and termination in a more complete manner, and supports C++ code. The older version of the API does not handle C++ code properly. It is still available for compatibility but not recommended for use.

```
void * xtlib_load_overlay_ext (xtlib_packaged_library * library,
                               xtlib_ovl_info *         info);

void *
xtlib_load_overlay_ext_custom_fn ( xtlib_packaged_library * library,
                                   xtlib_ovl_info *         info,
                                   memcpy_func              mcpy_fn,
                                   memset_func              mset_fn);
```

These functions load a fixed location overlay into memory and initialize it. `library` is a pointer to a packaged library which has been linked into your application. The process of doing so is described in the following sections. `info` is a pointer to an opaque data structure used to maintain library context. You must allocate an `xtlib_ovl_info` structure and pass the pointer to it, and this structure must remain allocated until the library has been unloaded.

The functions return a pointer to the library's entry point function, which must be named `_start`. See Section 4.5.2 for details.

Use the function `xtlib_load_overlay_ext_custom_fn()` if you want to provide your own custom functions for copying and setting memory. See Section 4.6 for more information.

```
void xtlib_unload_overlay_ext (xtlib_ovl_info * info);
```

This function unloads a previously loaded library. The `info` pointer must point to the `xtlib_ovl_info` structure allocated when the library was loaded. It calls the library termination functions to ensure proper cleanup and shutdown. After this function returns, the code and data memory used by the library can be freed or reused.

```
void * xtlib_load_overlay (xtlib_packaged_library * library);

void * xtlib_load_overlay_custom_fn (xtlib_packaged_library * library,
                                     memcpy_func mcpy_fn,
                                     memset_func mset_fn);
```

These older functions are retained for backward compatibility. We recommend you use the newer API described above. The older API did not provide an unload function. This did not handle e.g. calling destructors for static C++ objects in the library.

4.4.2 Position Independent Library API

There are two ways to load a position independent library. The library can be built so that code and data occupy the same memory region, or it can be built to separate the code and data regions (so that code can be loaded into program memory and data can be loaded into data memory).

```
    unsigned int xtlib_pi_library_size (xtlib_packaged_library * library);
```

This function returns the size of the memory block (in bytes) that must be available to load the library. The parameter `library` must point to a packaged library that was built to locate both code and data together (using the `piload` LSP). Your main application must reserve or allocate this amount of memory and pass it to the library load function.

```
    void * xtlib_load_pi_library (
              xtlib_packaged_library * library,
              void * destination_address,
              xtlib_pil_info * lib_info);

    void * xtlib_load_pi_library_custom_fn (
              xtlib_packaged_library * library,
              void * destination_address,
              xtlib_pil_info * lib_info,
              memcpy_func mcpy_fn,
              memset_func mset_fn);
```

These functions load a position independent library into memory and initialize it. `li-brary` is a pointer to a packaged library which has been linked into your application. The process of doing so is described in the following sections. `destination_address` is a pointer to the memory block that you have allocated for loading this library. It must be at least the size specified by `xtlib_pi_library_size()`. `lib_info` is a pointer to an opaque data structure used to maintain library context. You must allocate an `xtlib_pil_info` structure and pass the pointer to it, and this structure must remain allocated until the library has been unloaded.

The functions return a pointer to the library's entry point function, which must be named `_start`. See Section 4.5.2 for details.

Use the function `xtlib_load_pi_library_custom_fn()` if you want to provide your own custom functions for copying and setting memory. See Section 4.6 for more information.

```
unsigned int xtlib_split_pi_library_size (
          xtlib_packaged_library * library,
          unsigned int *code_size,
          unsigned int *data_size);
```

This function returns the sizes (in bytes) of the code and data areas that must be available to load the library. The parameter `library` must point to a packaged library that was built to locate code and data separately (using the `pisplitload` LSP). The required code and data sizes are returned in `code_size` and `data_size` respectively. Your main application must reserve or allocate this amount of memory and pass it to the library load function. The function's return value is `XTLIB_NO_ERR` on success. On error, it returns an error code.

```
void * xtlib_load_split_pi_library (
          xtlib_packaged_library * library,
          void * destination_code_address,
          void * destination_data_address,
          xtlib_pil_info * lib_info);

void * xtlib_load_split_pi_library_custom_fn (
          xtlib_packaged_library * library,
          void * destination_code_address,
          void * destination_data_address,
          xtlib_pil_info * lib_info,
          memcpy_func mcpy_fn,
          memset_func mset_fn);
```

These functions load a split position independent library into memory and initialize it. `library` is a pointer to a packaged library which has been linked into your application. The process of doing so is described in the following sections. The parameter `destination_code_address` is a pointer to the code memory block that you have allocated for loading this library. The parameter `destination_data_address` is a pointer to the data memory block that you have allocated for loading this library. These must be at least the sizes specified by `xtlib_split_pi_library_size()`. `lib_info` is a pointer to an opaque data structure used to maintain library context. You must allocate an `xtlib_pil_info` structure and pass the pointer to it, and this structure must remain allocated until the library has been unloaded.

The functions return a pointer to the library's entry point function, which must be named `_start`. See Section 4.5.2 for details.

Use the function `xtlib_load_split_pi_library_custom_fn()` if you want to provide your own custom functions for copying and setting memory. See Section 4.6 for more information.

```
void xtlib_unload_pi_library (xtlib_pil_info * lib_info);
```

This function unloads a previously loaded library. The `lib_info` pointer must point to the `xtlib_pil_info` structure allocated when the library was loaded. It calls the library termination functions to ensure proper cleanup and shutdown. After this function returns, the code and data memory used by the library can be freed or reused.

4.5 Tool Flow

The tool flow, usage type, and certain special considerations for each type of library are quite similar. To build and use a loadable library, follow the steps below. When the steps differ for a fixed-location overlay and a position-independent library, the differences are highlighted.

1. Find or generate the appropriate Linker Support Packages.
2. Modify and compile the source code for the library.
3. Link the code into a loadable library using the appropriate LSP from Step 1.
4. Package the loadable library into an object file using `xt-pkg-loadlib`.
5. Modify the main program to load the library using the API described in this document.
6. Link the main program against the object file from Step 4.

4.5.1 Step 1: Finding the Custom LSPs

Loadable libraries need to be linked with a custom linker support package, and a main application that loads a fixed location overlay also needs to be linked with a custom LSP. This first step instructs you in finding or building the appropriate custom LSPs.

The content of this step depends entirely on whether you are using position-independent libraries or fixed-location overlays. Again, you should be very familiar with LSPs, as documented in the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

LSPs for Fixed-Location Overlays

When using fixed-location overlays, memory is partitioned into two parts: the part reserved for the main program and the part reserved for the overlays. Partitioning memory requires two custom LSPs, which partition the memory map. In this example, they will be called `mainlsp` and `overlaylsp`, although you can use other appropriate names.

To generate `mainlsp`:

1. Choose a template LSP from the standard LSPs as described in the *Xtensa Linker Support Packages (LSPs) Reference Manual*. Alternatively, you can modify one you have developed yourself.
2. Copy this LSP to a new directory called `mainlsp`.
3. Change to this new directory.
4. Reserve areas of memory for the library by editing `memmap.xmm` and adding `RESERVE_SEGMENT_AREA` commands.

   For example, to reserve ten kilobytes from the end of iram0 and same amount from the end of dram0 and sysram, use the following lines:

   ```
   RESERVE_SEGMENT_AREA = ".iram0.text 10240 + 0 end"
   RESERVE_SEGMENT_AREA = ".dram0.data 10240 + 0 end"
   RESERVE_SEGMENT_AREA = ".sram.data 10240 + 0 end"
   ```

   The main application will not be able to use this memory.
5. Generate new linker scripts for this LSP using the command:
   ```
   xt-genldscripts -b .
   ```

To generate `overlaylsp`:

1. Copy the `mainlsp` LSP generated above into a new directory called `overlaylsp`.
2. Change to this new directory.
3. In `memmap.xmm`, change the words `RESERVE_SEGMENT_AREA` to `USE_SEGMENT_AREA`, and remove any line containing "`ROMING=true`".
4. Add the lines:

   ```
   ENTRY = _libinit
   USE_SEGMENT_AREA = ".iram0.text 10240 + 0 end"
   USE_SEGMENT_AREA = ".dram0.data 10240 + 0 end"
   USE_SEGMENT_AREA = ".sram.data 10240 + 0 end"
   ```
5. Simplify the file `specs` in the `overlaylsp` directory so it appears as follows:
   ```
   *startfile:
   crti%O%s crtbegin%O%s
   *endfile:
   crtend%O%s crtn%O%s
   *lib:
   ```
6. Generate the new linker scripts for this LSP using the command:
   ```
        xt-genldscripts -b .
   ```

You now have two custom LSPs, suitable for linking both the main application and the overlay.

LSP for Position-Independent Libraries

You do not need to generate or edit an LSP for position-independent libraries. Instead, you will link your position independent library against the standard `piload` or `pisplitload` LSP provided as part of your configuration.

4.5.2 Step 2: Modifying and Compiling the Library Source Code

Determining what to modify in your source code requires understanding two characteristics of loadable libraries. Both fixed-location overlays and position-independent libraries have these characteristics.

- First: Loadable libraries must be self-contained, meaning that they cannot directly refer to symbols that they do not provide themselves. Loadable libraries can link against archive files, such as `libm.a`, but any code from that archive will not be visible to the main program or other loadable libraries. This means that without care, some functions and data may appear in both the loadable library and the main program. Typically, this is only a code-size issue.
- Second: The API only allows the main program to directly access a single symbol in the library, the `_start` symbol. The library cannot directly access any symbol in the main program. Other references are possible, but must be made indirectly as described below.

Although at first glance these restrictions seem quite limiting, the solution is straightforward: Any other symbol's address must be passed to or from the library as an argument to the `_start` function.

Modifying Your Source Code

The easiest way to understand what needs to happen is by example. In the following example, an application uses a function `foo` from a library. Also in this example, the library needs to call the C library function `printf`, which is linked into the main application. Modify the library to provide a function named `_start` that takes a pointer to `printf` as shown below. Observe how the library uses a preprocessor `#define` to make all calls to `printf` indirect, through a pointer.

```
#include <stdio.h>

/* declare a printf function pointer */
int (*printf_ptr)(const char *format, ...);
/* replace all calls to printf with calls through the pointer. */
#define printf printf_ptr

/* This is the function provided by the library. */
char * foo(void)
```

```
{
  printf("executing function foo\n");
  return "this string returned from foo";
}

void * _start(int (*printf_func)(const char * format, ...))
{
  printf_ptr = printf_func;
  /* The main application wants to call the function foo, but
     can't directly reference it. Therefore, this function returns
     a pointer to it, and the main application will be able to call
     it via this pointer.  */
  return foo;
}
```

The main application calls `_start`, passing a pointer to `printf`, and taking a pointer to `foo` in return. The source code for this is shown below in Step 5. If the library and the main program need to communicate the value of more than one symbol, then `_start` can take and return arrays of pointers, rather than just single pointers. The function `_start` may have any prototype that you want, as long as both the main program and the library agree on what it is.

The easiest way to find out what functions the library needs to call indirectly is to link your library as described in the next step. Then note any undefined symbols and modify your source code to reference them via pointers set as described above.

Compiling Your Source Code

Fixed-Location Overlays: Compile your source code as usual, with your preferred debug options and optimization levels. In this example, the compile line looks like this for fixed-location overlays:

```
xt-xcc -c -o library.o library.c
```

Position-Independent Libraries: Because it can be loaded at any address, make sure that the code in the library is position-independent by using the `-fpic` flag along with your normal compile options, as shown below:

```
xt-xcc -fpic -O2 -o library.pic.o -c library.c
```

Use the C++ compiler invocation if compiling C++ code (or a mixture of C and C++).

4.5.3 Step 3: Linking the Library Code

In this step, you will link the library code into a loadable library using the appropriate LSP.

At the final link for your library, use the flag `-mlsp=<lspname>` where *`lspname`* is the name of the appropriate LSP from Step 1. Position-independent libraries require adding two flags to the link line: `-Wl,--shared-pagesize=128` and `-Wl,-pie`, in that order. Conventionally, this output file uses the extension `.lib`. Also, your library must be linked against the loader library `libloader.a`.

In this example, we use the following line for a fixed-location overlay:

```
xt-xcc -mlsp=../overlaylsp -o fixed_location.lib library.o -lloader
```

And the following line for a contiguous position-independent library:

```
xt-xcc -mlsp=piload -Wl,--shared-pagesize=128 -Wl,-pie \
  -o position_independent.lib library.pic.o -lloader
```

Or the following line for a position-independent library with code and data loadable separately:

```
xt-xcc -mlsp=pisplitload -Wl,--shared-pagesize=128 -Wl,-pie \
  -o position_independent.lib library.pic.o -lloader
```

The output file from this step will be used both for the input to the next step and for debugging.

4.5.4 Step 4: Packaging the Loadable Library

This step explains how to package the loadable library into an object file using the `xt-pkg-loadlib` tool.

The file produced in the previous step is an image of the library as it should appear in memory at execution time. This step packages that file into a data structure suitable for linking against the main application and storing in memory (for example in ROM). The library cannot easily be disassembled or examined after this step, which is why the output from step 3 should be kept for debugging later.

To package the file, use the following command:

```
xt-pkg-loadlib -e <library_name> -o <output_file_name>\
  <input_file_name> [-p]
```

Where *`input_file_name`* is the name of the file from Step 3, *`output_file_name`* is the name of the output file (which must end in `.o`), and *`library_name`* is a symbol that the main program will use to refer to the packaged library. When packaging a position-independent library, use the `-p` flag.

To run the tool for this example, use the following command for the fixed-location overlay:

```
xt-pkg-loadlib -e fixed_location_overlay -o packaged_flo.o \
  fixed_location.lib
```

Similarly for the position-independent library use the following command:

```
xt-pkg-loadlib -e pi_library -o packaged_pil.o \
  position_indepenent.lib -p
```

4.5.5 Step 5: Modifying the Main Program to Load the Library

The code to load the library is straightforward using the Library-Loader API and the library described in this chapter. Examples for both types follow.

Loading a Fixed-Location Overlay

The code to load a fixed location overlay is simple and requires a single call to `xtlib_load_overlay_ext()`. The complete code is in `overlay_test.c`, in the `examples/LibraryLoader` directory; following is an abbreviated version:

```
#include <xt_library_loader.h>

/* The library name from step 4. */
extern xtlib_packaged_library fixed_location_overlay;

/* A function pointer with the same type as the library's _start */
typedef void * (*flo_start_ptr_t)(void * printf_ptr);

flo_start_ptr_t flo_start;

/* A function pointer of same type as the function in the library. */
typedef char * (*foo_ptr_t) (void);

foo_ptr_t foo;

int main(int argc, char * argv[])
{
xtlib_ovl_info info;

  flo_start = (flo_start_ptr_t)
xtlib_load_overlay_ext(&fixed_location_overlay, &info);
```

```
  if (flo_start != NULL) {
    char * string_returned_from_foo;

    foo = flo_start(printf);
      /* We can now call "foo" from the library normally. */
    ...
  }
  else {
    printf("Loading library failed with error code: %d\n",
xtlib_error());
  }
}
```

Loading a Position-Independent Library

Loading a position-independent library is slightly more complicated than loading a fixed-location overlay. First, use the provided API to calculate the size of memory needed to hold the library. Then, allocate space to match those requirements. For a library linked with `piload` LSP this memory needs to be both writable and executable. For a library linked with `pisplitload` LSP you need two memory blocks - one writable and executable for code part and one writable for data part. You will also need to allocate an object of type `xtlib_pil_info`. This structure can be allocated on the stack or dynamically, and will be filled out when you call `xtlib_load_pi_library` or `xtlib_-load_split_pi_library.` The loader uses this structure to keep track of how the packaged library was loaded, and you will need it for subsequent calls to manipulate the library.

The complete code is in the example directory in the file `pil_test.c`; below is an abbreviated version which loading library linked using `piload` LSP:

```
#include <xt_library_loader.h>

/* The library name from step 4. */
extern xtlib_packaged_library pi_library;

/* A function pointer with the same type as the library's _start */
typedef void * (*pil_start_ptr_t)(void * printf_ptr);
pil_start_ptr_t pil_start;

/* A function pointer of same type as the function in the library. */
typedef char * (*foo_ptr_t) (void);
foo_ptr_t foo;

int main(int argc, char * argv[])
{
  xtlib_pil_info lib_info;
  unsigned int bytes = xtlib_pi_library_size(&pi_library);
  void * memory = malloc(bytes);
  pil_start = xtlib_load_pi_library(&pi_library, memory, &lib_info);
  if (pil_start != NULL) {
    foo = pil_start(printf);
    ... we can now call foo normally…
    xtlib_unload_pi_library(&lib_info);
  }
  else {
    printf("failed with error code: %d\n", xtlib_error());
  }
  return 0;
}
```

For a library linked using `pisplitload` LSP, you need to allocate two memory blocks. In the example below, system memory is used but you could use local memories as well. The complete code is in the example directory in the file `pilsl_test.c`. The following is an abbreviated version:

```
#include <xt_library_loader.h>

/* The library name from step 4. */
extern xtlib_packaged_library pi_library;

/* A function pointer with the same type as the library's _start */
typedef void * (*pil_start_ptr_t)(void * printf_ptr);
pil_start_ptr_t pil_start;

/* A function pointer of same type as the function in the library. */
typedef char * (*foo_ptr_t) (void);
foo_ptr_t foo;

int main(int argc, char * argv[])
{
  xtlib_pil_info lib_info;
  unsigned int code_bytes;
  unsigned int data_bytes;
  if (xtlib_split_pi_library_size (&pi_library, &code_bytes,
&data_bytes) != XTLIB_NO_ERR) {
    printf ("failed with error: %d\n", xtlib_error ());
    return 0;
  }
  void * data_memory = malloc(data_bytes);
  void * code_memory = malloc(code_bytes);
  pil_start = xtlib_load_split_pi_library(&pi_library, code_memory,
data_memory, &lib_info);
  if (pil_start != NULL) {
    foo = pil_start(printf);
    ... we can now call foo normally…
```

```
      xtlib_unload_pi_library(&lib_info);
    }
    else {
      printf("failed with error code: %d\n", xtlib_error());
    }
    return 0;
  }
```

Ensuring Correct Multithreading Behavior

If you are using multithreading, calling into multiple libraries that are dynamically loaded and unloaded in overlapping memory locations, and the calls into the libraries are not necessarily by the same thread that loaded them, you must take some extra care.

The HAL `xthal_clear_regcached_code()` function must be called when code that may have previously been executed in the current thread, is unloaded and replaced (in overlapping memory locations, by another thread) with new code that may also be called by the current thread. See Section 3.14 for details.

4.5.6 Step 6: Linking the Main Program

This final step provides instructions to link the main program against the packaged library and the loader library. You will need to link your application against the code that loads the packaged libraries, and against the packaged libraries themselves. The library `libloader.`a implements the library loading API and ships as part of your configuration.

This library includes code to load both fixed-location overlays and dynamic-location libraries, although only the code you actually use will be included in your final application.

Next, add both -lloader and the name of the object file produced in Step 4 to your link line. For fixed-location overlays, you will also need to use the proper LSP produced in Step 1. In this example, the link line looks like this for fixed-location overlays:

```
xt-xcc packaged_flo.o overlay_test.o -o flo_test.exe \
   -mlsp=../mainlsp -lloader
```

And the link line looks like this for position-independent libraries:

```
xt-xcc packaged_pil.o pil_test.o -o pil_loader.exe -lloader
```

Note that an application that uses a position-independent overlay does not need a special LSP.

4.6 Debugging

When linked into an application in its packaged form, a loadable library is not visible to the debugger, and thus the debugger cannot perform source level debugging without a few extra steps. These steps involve loading the symbol table for the loadable library by hand. The steps for loading debugging information for either type of library are the same.

After loading, you can enable source-level debugging with a fixed-location overlay with the following command to `xt-gdb,` or in the Xtensa Xplorer debug console:

```
add-symbol-file <filename> <address>
```

where <*filename*> is the path to the unpackaged loadable library produced in Step 3. The value to use for <*address*> depends on the type of library you load.

For fixed-location libraries, use `xt-objdump -h` to determine the address by using the following command with name of the library file, as given in Step 3. The appropriate address will be the one in the VMA column. For example, with the command below.

```
xt-objdump -h fixed_location.lib
...
1. .text  0000011a 4ffff854 4ffff854 00000288 2**2
...
```

For position-independent libraries, use `xt-gdb` to determine the address by calling `xtlib_pi_library_debug_addr` with the address of the `xt_pil_info` structure you allocated earlier:

```
(xt-gdb) p xtlib_pi_library_debug_addr (&lib_info)
$2 = (void *) 0x6000a850
```

Next, use the printed value for <*address*> in your add-symbol-file command. Note that you must type the address manually, rather than use an xt-gdb convenience variable.

`xt-gdb` will now be able to see symbols in the overlay and step through it line by line. This command adds the debugging information found in <*filename*> to the main program's debugging information.

If the library and the main program each have a symbol with the same name, and debugging information for both is loaded, then you might not be able to refer to either symbol properly. You can solve this problem by loading the debug information for only a single file:

```
symbol-file <filename> <address>
```

You will not be able to symbolically debug the code that does not have symbolic debugging information loaded, but you will be able to see all symbols in the other library correctly.

If the program subsequently unloads or stops referring to the library, you should unload the library's debugging information. As there is no way to do so for just the library, you must clear all debugging information and then reload the main program's debugging information.

To clear all debugging information and reload the main program’s information, use the symbol-file command with `xt-gdb`:

```
symbol-file <main-program-name>
```

You can then load any new libraries’ debugging information as above.

4.7 Using DMA or Custom Functions to Load a Library

As it is written, the Library Loader uses `xthal_memcopy` to move a loadable library’s data into place. However, some users will want to use, for example, a DMA engine, or some other mechanism. The library loader provides an easy mechanism to do this. Instead of loading your library with `xtlib_load_overlay`, you can call a very similar function that allows you to specify a function to copy the data. It has the following prototype:

```
void * xtlib_load_overlay_custom_fn(xtlib_packaged_library * library,
       memcpy_func copy_fn, memset_func set_fn);
```

Or in the case of position-independent libraries, instead of calling `xtlib_load_pi_library`, use:

```
void * xtlib_load_pi_library_custom_fn (xtlib_packaged_library *
       library, void * destination_address, xtlib_pil_info * lib_info,
       memcpy_func mcpy_fn, memset_func mset_fn)
```

and instead of `xtlib_load_split_pi_library`, use:

```
void * xtlib_load_split_pi_library_custom_fn (xtlib_packaged_library *
       library, void * destination_code_address, void *
       destination_data_address, xtlib_pil_info * lib_info,
       memcpy_func mcpy_fn, memset_func mset_fn);
```

Observe the last two parameters. The `mcpy_fn` parameter allows you to provide your own function to copy the data. It should have exactly the same prototype and semantics as the normal c-library `memcpy`:

```
void * memcpy(void * dest, const void * src, size_t n);
```

Your `memcpy` will be called with `dest`, `src` and `size` all set appropriately. You can then copy the data however you like, including via DMA. The Library Loader will call this function anytime needs to move data

Similarly, the `mset_fn` parameter allows you to provide a function to zero the data. It should have exactly the same prototype and semantics as the normal c-library `memset`.

Observe that functions which write or read from an instruction RAM must do it in aligned, four-byte chunks.

4.8 Loading a Library From a Host Processor

In a multiprocessor environment, you may want to load libraries for an Xtensa processor (target processor) from another processor (a host processor). This section describes the process of loading libraries from host to the target CPU. What initiates the load and a host-target communication protocol implementation is out of scope of this section.

Note: Maintaining memory consistency is the programmers' responsibility. Depending on system configuration you may need to flush/invalidate caches on the host and target processors to make sure that data is written out and is not corrupted.

Packaging the Loadable Library

To package the library, follow the steps described in Section 4.5.1, Section 4.5.2, and Section 4.5.3. Step 4 might be different:

- If both target and host are the same Xtensa processor, you can follow Step 4 in the Section 4.5.4. This will produce object file that you can link to the host application.
- If host and target processors are different Xtensa processors, you need to pass additional flags to `xt-pkg-loadlib` tool in Step 4 to specify Xtensa configuration for target and host processors:

  ```
  --xtensa-host-core=<core> --xtensa-host-params=<params> \
   --xtensa-host-system=<system> --xtensa-target-core=<core> \
   --xtensa-target-params,=<params> --xtensa-target-system=<system>
  ```
  These flags correspond to `--xtensa-core=<core>`, `--xtensa-params=<params>` and `--xtensa-system=<system>` Xtensa tools flags. Please see the *Xtensa Software Development Toolkit User's Guide* for their definition.
- If the host processor is not Xtensa, you must decide how to package the library in your application. To use the Library Loader API, the host application must have the content of the loadable library file in memory and should pass its address as `xtlib_packaged_library*` parameter in API calls. Use the following command to discard all unnecessary data from the library file:

  ```
  xt-pkg-loadlib -s <output_file_name> <input_file_name> [-p]
  ```

Where `input_file_name` is the name of the file from Step 3 and `output_-file_name` is the name of the output file. When input file is a position-independent library, use the `-p` flag. As symbol tables are discarded by this command, the output from Step 3 should be kept for later debugging.

Loading Library

The library loading functions discussed later in this section require memory copy and memory set callbacks provided by the programmer.

The `memcpy_func_ex` callback must copy *n* bytes starting from `src` address on the host processor to `dest` address on the target processor. The `memset_func_ex` call-back must set *n* bytes starting from `dest` address on the target processor to the value c interpreted as unsigned char. Both callbacks must return `dest`.

```
typedef xt_ptr (*memcpy_func_ex) (xt_ptr dest, const void * src,
        unsigned int n, void *user);
typedef xt_ptr (*memset_func_ex) (xt_ptr dest, int c, unsigned int n,
        void *user);
```

`user` is a user-defined context pointer. The library loading functions pass the unmodi-fied value of their `user` parameter. Note that the `dest` address is in the target processor address space, it is not translated into host processor address space.

Loading a Fixed-Location Overlay

The newer fixed-location overlay API is not usable from a host processor at this time.

To load a fixed-location overlay, call the `xtlib_host_load_overlay()` function on the host processor:

```
xt_ptr xtlib_host_load_overlay (xtlib_packaged_library * library,
        memcpy_func_ex mcpy_fn, memset_func_ex mset_fn, void *user);
```

with `library` pointing to the packaged overlay library in memory; memory copy and memory set callbacks and a context pointer `user` which is passed unmodified into call-backs. The function returns the pointer to the library `_start` function on success or zero if failed. The pointer returned is in the target processor's address space.

There is no need to link the target application with the Library Loader in this case. All the work is performed by the host processor.

Loading a Position-Independent Library

Loading a position-independent library is similar to the single processor case, but some steps are performed on the host processor and others are performed on the target processor. Therefore, both target and host applications should be linked with the Library Loader. Please review Section 4.5.5 to become familiar with the loading position-independent library on a single processor.

These are the steps to perform on the host and target processors:

1. Host: use `xtlib_pi_library_size (xtlib_split_pi_library_size` if it's a library linked with `pisplitload` LSP) to calculate the size of memory needed to hold the library;
2. Either Target or Host: allocate space on the target processor to match library requirements;
3. Either Target or Host: allocate the `xtlib_pil_info` structure; You can allocate it in a shared memory block. Alternatively, you can allocate it on the host and copy it as-is to the target processor memory after Step 4 below.
4. Host: call `xtlib_host_load_pi_library` function for a library built with the `piload` LSP or call `xtlib_host_load_split_pi_library` for a library linked with the `pisplitload` LSP. This step loads the library to the target processor and fills out the `xtlib_pil_info` structure;
5. Target: complete loading by calling `xtlib_target_init_pi_library` with the `xtlib_pil_info` structure filled out from the previous step;

When the library is no longer needed, call the following function on the target processor before releasing the memory allocated for the library:

```
void xtlib_unload_pi_library (xtlib_pil_info * lib_info);
```

where `lib_info` is the library `xtlib_pil_info` structure.

In Step 4 above, a loadable library linked with the `piload` LSP is loaded into contiguous memory block using:

```
xt_ptr xtlib_host_load_pi_library (xtlib_packaged_library *library,
       xt_ptr destination_address, xtlib_pil_info *lib_info,
       memcpy_func_ex mcpy_fn, memset_func_ex mset_fn, void *user);
```

where

- `library` points to the loadable library image in memory,
- `destination_address` is a pointer to the memory allocated in Step 2 in the target processor address space (*i.e.* not translated to the host address space),
- `mcpy_fn` and `mset_fn` are memory copy and memory set callbacks, and
- `user` is a context pointer passed into callbacks unmodified.

On success, it fills out `lib_info` structure and returns non zero value. On failure it returns zero.

A loadable library linked with the `pisplitload` LSP has code and data loaded into separate memory blocks:

```
xt_ptr xtlib_host_load_split_pi_library (
      xtlib_packaged_library *library, xt_ptr destination_code_address,
      xt_ptr destination_data_address, xtlib_pil_info * lib_info,
      memcpy_func_ex mcpy_fn, memset_func_ex mset_fn, void *user);
```

where:

- `library` points to the loadable library image in memory,
- `destination_code_address` and `destination_data_address` are pointers to the memory allocated in Step 2 for the library text section and data section. Both pointers in the target processor’s address space (*i.e.*, addresses are not translated to the host address space),
- `mcpy_fn` and `mset_fn` are memory copy and memory set callbacks, and
- `user` is a context pointer passed into callbacks unmodified.

On success, fills out `lib_info` structure and returns non zero value. On failure it returns zero.

On the target processor, loading is completed by a call to:

```
void * xtlib_target_init_pi_library (xtlib_pil_info * lib_info);
```

where `lib_info` is the `xtlib_pil_info` structure filled out on the host in Step 4.

It returns pointer to the library `_start` function on success or NULL on failure.

5. Automatic Xtensa Overlay Manager (AXOM)

5.1 Automatic Xtensa Overlay Manager (AXOM) Library

The Automatic Xtensa Overlay Manager (AXOM) is a library that manages the mapping and execution of code overlays. Overlays are code blocks in a program that are executed out of a common location in memory, but normally reside at different locations. Each overlay section has to be mapped (loaded) into this common area (“map area”) before the code in it can be executed. There can in principle be multiple such “map areas”, although at this time only one such area is being supported.

Unlike the loadable libraries described in Chapter 4, all overlays are contained within the same executable image. Thus, all overlay code shares the same namespace as the main program code. Also unlike the loadable libraries, functions placed in overlays can call other functions in non-overlay code, and access any data in the executable, without needing any special handling. It is even possible to call a function in one overlay from a function in another overlay, even though both are mapped to the same “map area”. All of this calling and mapping/unmapping is automatically handled by the overlay manager.

Since all overlays are part of one executable, they share symbols and data. Two overlays cannot define global symbols of the same name. The AXOM implementation does not support keeping data in overlays, only code and literals. The literals associated with the overlay code cannot be separated out to be placed elsewhere; they must reside alongside the code.

The primary motivation for using code overlays is to manage scarce program memory. In some circumstances, there is not enough program memory to hold all the executable code. In this case, overlays allow much larger program sizes by essentially paging code in and out of program memory on demand. Although the mechanism for moving overlays in and out is automatically handled by the overlay manager, the partitioning of the application code into overlays and permanently resident code must be done by the application designer manually.

Enabling overlays in an application is quite simple. It requires including one extra header file, tagging functions that will go into overlays with a special attribute, and linking against the overlay manager library using a specially modified LSP (Linker Support Package). If you are using Xtensa Xplorer, then many of these steps can be automated. All these steps are fully described in the following sections.

The complete source code for the overlay manager can be found in the following directory:

```
<xtensa_tools_root>/xtensa-elf/src/liboverlay
```

A simple example of overlay usage is provided with Xtensa Xplorer. Please refer to the Xplorer documentation for details.

5.1.1 Terminology

- *overlay* - A module or set of functions in a program, among many, only one of which may be mapped (loaded) at any given time into an *overlay area*.
- *overlay area* - An area of memory that can contain one of a set of *overlays*.
- *common area* - Area(s) of memory outside any *overlay area*. Code and data in the common area are always present, regardless of which overlays are mapped.
- *overlay region* - Alternate term for *overlay area*.
- *overlay manager* - Target code that maps and unmaps overlays as needed.
- *mapped overlay* - An *overlay* that has been loaded into its *overlay area*.
- *unmapped overlay* - An *overlay* that is NOT loaded into its *overlay area*.
- *loaded overlay* - Alternate term for *mapped overlay*.
- *load address* – Also called the *load memory address (**LMA**)*. This is the address at which an *overlay* is stored (but not used), from which it can be copied to its *overlay area* to map it. When the executable that contains overlays is loaded, all overlay sections are loaded at their *load addresses*, and the *overlay area* is initially empty. In some cases, the unmapped overlays may not reside in memory, e.g. they can reside in flash and be mapped into memory on demand.
- *mapped address* – Also called the *virtual memory address (**VMA**)*. This is the address of an *overlay* within its *overlay area*, when it is *mapped* (in use or ready to be used). This is often the same address as that of the entire *overlay area*, but need not be (could be any address within the overlay area, such that the entire overlay fits within that area). The name VMA is used simply to match the linker's use of this term, and does not have any relation to OS-supported virtual memory, or to an MMU.

5.1.2 Requirements

The overlay manager has the following requirements in this release:

- The overlay manager supports only XEA2 configurations. XEA3 configurations are not supported.
- The use of automatic code overlays requires the windowed calling ABI. The CALL0 ABI is not supported.
- The default implementation of the overlay manager works in the XTOS environment. For use with a preemptive OS, some OS-specific adaptation is required, as explained in Section 5.2 “Operating System Support”.
- There are no specific hardware requirements other than those mandated by the windowed calling ABI.

5.1.3 Features

The overlay manager supports automatic overlay management with minimal disruption to normal program development and execution. Some of the salient features are:

- An overlay is automatically mapped as needed when a function in that overlay is called, or returned to. The overlay manager is automatically invoked upon such calls or returns. An API call is provided to explicitly load an overlay, if such prefetching is found useful.
- A function in one overlay can call another function in a different overlay, even though both overlays execute out of the same memory area. The overlay manager handles the process to make this possible.
- It is possible to take the address of an overlay function and call it using this pointer. Upon calling the function through this pointer, the overlay is automatically mapped as needed.
- The overlay manager allows users to provide their own functions to load overlays (*i.e.*, copy them from their load address to their mapped address). This allows taking advantage of system-specific speedups, for example, DMA transfers.
- The normal application build flow is unaffected by the presence of overlays. The memory map file does need to be modified to specify the number of overlays required etc.
- Xtensa Xplorer is overlay aware and can support debugging overlay code. Xplorer also supports memory map editing and LSP generation for overlays. The command line debugger (xt-gdb) is also overlay aware.

5.1.4 Restrictions

There are some restrictions on the use of overlays in an application.

The most significant one is that the placement of data in overlays is not supported. Managing data in overlays can be extremely complex, as there is no simple way to automatically catch references to data in specific overlays. Nor is there any simple way to save changes to data in an overlay when that overlay is unmapped.

Some other restrictions are:

- Each overlay is contiguous (a single sequence of bytes). Literals and code are combined together, and their placement optionally controlled using the single `.overlay` section described in the *Modifying the Memory Map* subsection of Section 5.1.5. There is no option to split the literals and the code into separate sections to support, for instance, literals in data RAM and code in instruction RAM. Thus, if overlays are placed in IRAM, literals are placed there as well.
- Given the above, overlays are not compatible with the software-extended L32R option available on older Xtensa processors.

- Overlays cannot be named, only numbered. If N overlays are supported, the numbers range from 0 to N-1. It is possible to define names that map to these numbers.
- There can be only one overlay mapped area, *i.e.*, only one area where overlays can be copied to and executed from.
- The overlay manager itself, obviously, cannot reside in an overlay.
- Interrupt and exception handing code must not be resident in overlays. Also, interrupt and exception handling code must not call any overlay functions directly or indirectly, else unexpected behavior can occur.
- Functions that the compiler may invoke automatically (such as functions in libgcc or a few C library functions such as `memcpy()`) must not be placed into an overlay. There is no automatic enforcement or checking of this constraint.
- Note that in C++ code, class constructor and destructor functions can be called automatically by the compiler to handle unnamed temporary objects, in addition to the objects explicitly created by the programmer. If the constructor or destructor is an overlay function, there may be unexpected side effects, for example, on performance. Placing constructor/destructor code in overlays should be done with care.
- The C/C++ compiler command line option `-ipa` should not be used with files that contain overlay functions. This option currently conflicts with overlays and will generate errors. Source files in the project that do not contain overlay code can still be compiled with this option.

5.1.5 Adding Overlay Support to an Application

The following steps are required to add overlays to an application:

1. Include the file `xtensa/overlay.h`. This file defines overlay-related macros and functions.
2. Declare each desired overlay using the DECLARE_OVERLAY macro.
3. Declare overlay functions as such by using the OVERLAY macro.
4. Add `liboverlay.a` to the list of libraries to be linked against.
5. Modify the memory map and generate new LSPs using Xtensa Xplorer.

Using the Function Attribute

Each overlay to be used in the program must be declared at least once in the program using the DECLARE_OVERLAY macro, like so:

```
DECLARE_OVERLAY (0);
```

This declares an overlay with ID zero. For readability, it would be better to #define names for the overlays, for instance:

```
#define INIT_OVERLAY     0
DECLARE_OVERLAY (INIT_OVERLAY);
```

Remember that overlay IDs must be in the range of 0 to M-1, where M is the number of overlays specified in the `memory.xmm` file.

Declaring Overlay Functions

Overlay functions are declared using the macro OVERLAY (N), where N specifies the overlay number. For example,

```
int func1(int arg1, int arg2) OVERLAY(3);
```

This declares the function `func1` as residing in overlay number 3. It is sufficient to specify the OVERLAY() attribute in the prototype declaration. It is not necessary to change the definition of the function itself.

Modifying the Memory Map

If you are using Xtensa Xplorer, please refer to the Xplorer online documentation topic *Changing the Memory Map of a Configuration* to see how to modify the memory map and generate new LSPs for overlay support. If you are doing this using command line tools, then you should be familiar with memory maps and LSPs. Read the *Xtensa Linker Support Packages (LSPs) Reference Manual* for more information before proceeding.

There are two parameters that can be added to the memory map file to specify the overlay properties.

```
OVERLAYS = <number>
```

This specifies how many overlays are to be created in the executable. In principle, the number of overlays possible is only limited by the amount of storage available (although we have not tested with more than 100 overlays). If the number is zero or negative, no overlay support is generated. If this parameter is absent, no overlay support is generated.

```
OVERLAY_ALIGN = <alignment>
```

This specifies the load address (LMA) alignment for the overlay sections. The load addresses are assigned by the linker such that these addresses are aligned to the specified value. For instance, a value of 8192 will ensure that the load addresses all start on 8-KB boundaries. This is useful in aligning the overlay sections with block start addresses if they are stored in flash, for example. This parameter is optional, the minimum value being 16 and the maximum 2^10, *i.e.*, 1MB. The value specified must be a power of 2. The default is 16 if not specified.

The placement of the overlay VMA cannot be directly controlled. It is possible to specify the memory segment in which the VMA is to be placed, by using the special section name ".overlay" in the list of sections for that segment. For example:

```
BEGIN sram1
0x50000000: sysram : sram1 : 0x10000 : executable, writable ;
 sram1_0 : C : 0x50000000 - 0x5000ffff : .overlay .sram1.rodata
.sram1.literal .sram1.data .sram1.bss;
END sram1
```

Note that this must be placed in an executable segment, *i.e.*, a segment from which code is allowed to be run. If this is not specified, then by default the overlay VMA will be placed in the same segment that holds the `.text` section and the VMA will follow immediately after the `.text` location.

For more precise definition of the overlay VMA, it can be placed at the beginning of the sections list (as shown above) – this ensures that the overlay VMA is placed at the start of the memory segment. In the example above, this means that the overlay VMA would be placed at address 0x50000000. If necessary, a segment can be split up into multiple segments to allow exact placement of the VMA. The example above can be modified to:

```
BEGIN sram1
0x50000000: sysram : sram1 : 0x10000 : executable, writable ;
 sram1_0 : C : 0x50000000 - 0x50003fff : .sram1.rodata .sram1.literal
.sram1.data .sram1.bss;
sram1_1 : C : 0x50004000 - 0x5000ffff : .overlay;
END sram1
```

Now the overlay will be placed at address 0x50004000. Of course, it must be ensured that there is enough room to hold the largest of the overlay sections when it is mapped or linker errors will occur.

The overlay LMA cannot be specified. For non-romable LSPs, the LMAs are assigned directly following the VMA location. For romable LSPs, the LMAs are assigned following the `.text` section.

Instead of starting from scratch, it is recommended to start with a standard memory map file for the applicable Xtensa configuration and modify it. The memory map file is found inside each LSP defined for that core. For romable images, it is best to start with a romable LSP.

An example step-by-step process to create a new LSP is as follows:

1. Copy an LSP from the `<xtensa_root>/xtensa-elf/lib directory` (*e.g.*, the “sim” LSP).
2. Edit the `memmap.xmm` file inside the LSP directory as described above.
3. Run `xt-genldscripts -b <lspname>` to update the LSP with your changes.

Disabling Overlays

It may be desirable to disable overlay support during development or debugging. This is accomplished by defining *XT_DISABLE_OVERLAYS* in the project file, the make file or the compiler command line. If included in source code, this must be defined before `overlay.h` is included. Rebuilding the application will now turn all overlay functions into regular functions.

5.1.6 Use Models

The overlay mechanism is intended primarily for use in memory-constrained systems. It can be used in the following ways:

- In a standalone XTOS environment with no OS.
  Multitasking is not an issue in this case.
- With an OS that has no knowledge of the overlay mechanism.
  In this case, the overlay mechanism is managed at the application level, e.g. by using a shared mutex to ensure that when a task is using the overlay, no other task may take control of the overlay.
- With an OS that does manage the overlay.
  In this case, the OS tracks the use of the overlay by tasks, and makes sure that the correct overlays are mapped and unmapped as tasks are suspended and resumed. If the OS you intend to use does not already have overlay support, see Section 5.2 “Operating System Support” for details on how to add it.

The overlay mechanism is not designed for use in an OS environment with separately loadable applications, *e.g.*, Linux. It can only be used when everything (OS + application code) is linked together into a single executable.

5.1.7 Overlay Manager API

Normal overlay use from an application does not require any overlay manager functions to be called. The following API is provided for convenience and optimization if needed. To use the API, include the file `xtensa/overlay.h` in your code.

int xt_overlay_map_async (int ov_id);

This function starts mapping the specified overlay into the location where it will execute from, *i.e.*, the VMA. It does nothing if the overlay is already mapped. This function is non-blocking, *i.e.*, it returns as soon as the mapping process has started. Note that this function will not be truly non-blocking unless `xt_overlay_start_map()` and `xt_overlay_is_mapping()` are re-implemented to support non-blocking operation, using some kind of asynchronous mechanism such as DMA, *etc.*

This function returns 0 if the overlay has been mapped and is ready, 1 if the mapping process has started and is in progress, and -1 in case of an error.

IMPORTANT: This function currently does not use the overlay locking functions for thread safety (Section 5.1.8). Since there is no completion function that must be called, there is no place to perform the unlocking. So, this function must not be called from the application when running in a pre-emptive multitasking environment.

void xt_overlay_map (int ov_id);

This function is listed here since it is part of the public API, but it is meant to be called by the overlay manager code only. This function maps the specified overlay into the location where it will execute from, i.e. the VMA. It does nothing if the overlay is already mapped. This function is blocking, i.e. it only returns after the overlay has been mapped. In a multitasking environment, this function will be thread safe if the appropriate locking functions have been implemented (Section 5.1.8). Keep in mind that though this function is blocking, internally it can use asynchronous operations like DMA for the actual data movement.

int xt_overlay_get_id (void);

This function returns the ID of the currently mapped overlay. It returns -1 if no overlay is mapped or an overlay is in the process of being mapped.

int xt_overlay_map_in_progress (void);

If an overlay is currently being mapped, this function returns the overlay ID. It returns -1 if no mapping is in progress.

5.1.8 Customizing the Overlay Manager

The overlay manager can be modified by overriding some of its default functions, and providing alternate implementations for its OS hooks.

User-Provided Functions

The following functions can be overridden to customize the mapping (copy) process and handle errors. To override the functions, simply define your own version of these functions with the same signature, and your versions will be used instead of the ones provided by the library.

IMPORTANT: These functions are called by the overlay manager. Do not call these directly from your application code, as bypassing the overlay manager could result in incorrect operation and undefined behavior.

IMPORTANT: While you can implement locking and unlocking in these functions for thread safety, this can get complicated in a preemptive multitasking environment. See the discussion in Section 5.2 “Operating System Support”.

IMPORTANT: These functions must never use floating point operations or any other coprocessor functions. Since the coprocessor state is not saved before these functions are invoked, using such operations could corrupt the state of a coprocessor that is being used by the application code. These functions should generally follow the same rules as for writing interrupt handlers. In particular, the compiler options `-LNO:simd and` `-mcoproc` must not be used. It is recommended that user override functions be compiled with the option `-mno-coproc.`

int xt_overlay_start_map (void * dst, void * src, unsigned len, int ov_id);

The `xt_overlay_start_map()` function is called by the overlay manager to copy an overlay of *len* bytes from its load address (LMA, given here as *src*) to its mapped address (VMA, given here as *dst*). It should return 0 on success and -1 on errors.

If this function only supports blocking (synchronous) operation, then it completes the entire copy. The default implementation is simply `memcpy()`.

If this function supports non-blocking (asynchronous) operation, it starts the overlay mapping (copying) process and returns immediately. However, it must also handle the possibility of being called while a previously started mapping was already started asynchronously (note: this can only happen if the application calls `xt_overlay_map_async()`). If that happens, this function must handle the situation gracefully, by either canceling the previously started mapping, or waiting for it to complete, before starting the new request.

int xt_overlay_is_mapping (int ov_id);

The `xt_overlay_is_mapping()` function returns an indication of whether an overlay mapping is in progress. For blocking (synchronous) implementations of `xt_overlay_start_map()`, this function can simply return zero. For non-blocking (asynchronous) implementations, this function checks progress and performs any housekeeping required after the mapping has completed; it returns 1 if the overlay specified is still mapping, and zero if the mapping is completed.

Note: After a map operation, the corresponding range of addresses may have to be invalidated in the processor's instruction cache, to make sure that the mapped code is fetched correctly. If the map operation copies data through the processor's data cache (e.g. memcpy) then the data cache may need to be written back before the instruction cache is invalidated.

void xt_overlay_fatal_error (int ov_id);

This function is called by the overlay manager if the mapping process encounters an error. It is up to the user to decide how to handle the error. The default version provided with the overlay library will call `exit()`.

OS-Provided Functions

These functions are specific to the underlying OS, if any. They are used by the overlay manager to protect against concurrent accesses to the overlay area.

void xt_overlay_lock (void);

Lock access to shared overlay resources. This typically acquires a mutex.

void xt_overlay_unlock (void);

Unlock access to shared overlay resources. This typically releases a mutex.

void xt_overlay_init_os (void);

Called by the overlay manager to initialize any OS specific data structures related to overlays. Specifically, it should set up the mutex used by the above two functions. The mutex should be set up with priority inheritance to prevent priority inversion.

5.2 Operating System Support

The overlay manager does not support any specific RTOS, but it has hooks built into it for multitasking RTOS support. Some changes must also be made to the RTOS context switching code to handle overlays. If your RTOS already has overlay support, or if you do not require RTOS overlay support, then you can safely skip this section.

In a pre-emptive multitasking environment, the simplest use option is to allow the overlay manager's automatic mapping mechanism to handle overlay management. If you want to do your own custom implementation, you may want to override `xt_overlay_map_async().` This function does not support locking. You may want to support locking and unlocking in your custom implementations of the mapping functions. But supporting locking can lead to situations like this. Say task A starts mapping overlay 1. Now it is preempted by task B, which tries to map overlay 2. Since A owns the mutex lock for overlay mapping, B is now blocked. If the mutex is not set up for priority inheritance, this could lead to a deadlock. If the mutex is set up properly, A will be resumed at a higher priority and allowed to run until it releases the mutex. Now once overlay 1 is mapped, A releases the mutex and is promptly suspended so that B can resume. Now B loads overlay 2, so all the work done in loading overlay 1 goes to waste. So, in this case, the preloading is no better than relying on the default mechanism. If multiple tasks at different priorities need to use different overlays, then some application-level locking may be required to ensure that the lower priority tasks are not perpetually starved of their chance to use the overlay.

5.2.1 Required RTOS Changes

If the RTOS is to be overlay aware and manage the overlay area as a shared resource, changes must be made to the context switch code. The overlay library provides some helper routines to help manage the overlay state. These are described in the following sections. Context switch code is often highly optimized assembly, and cannot easily call C functions without a penalty in performance. For this reason, the relevant overlay library functions are available as assembly macros which can be used in such code with minimal overhead. The library also provides C callable wrappers for these. To examine the code for these macros, refer to the following files in the overlay manager sources: `overlay_os_asm.h`, `overlay_os_asm.S` and `overlay_os_asm_1.S`.

The overlay mechanisms occasionally store data below the stack pointer, at offsets -20 .. -17 that is, in addition to the Windowed ABI storage of caller registers at offsets -16 .. -1. So when an interrupt is processed, the RTOS must be careful not to touch these bytes below sp (a1). This may require increasing some stack frame sizes, but in general, most RTOS code already handles this since these 4 bytes are also used by nested functions.

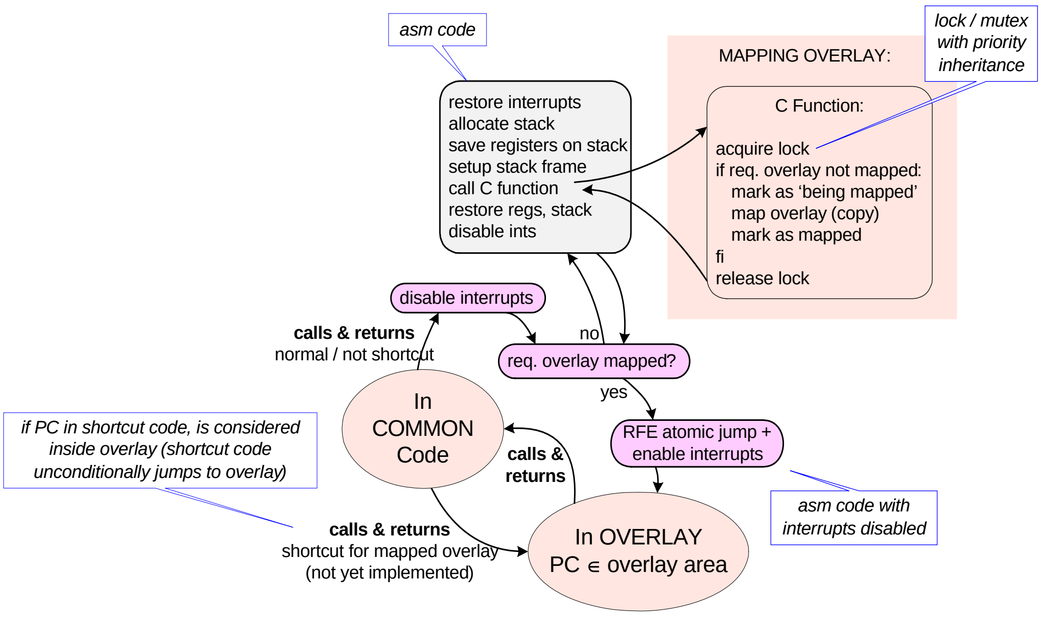

Figure 5–4. AXOM Execution Flow Diagram

5.2.2 Switching Out of a Task

When a task is being switched out, the current overlay state must be obtained and stored in the task's context. This is done using the function `xt_overlay_get_state()` or the macro *_xt_overlay_get_state*.

This is true at least for a task switched out due to an interrupt. For a task being switched out on an OS function call, the simplest is to adopt (and document for that OS) the convention that such OS functions may not be put in an overlay, and not do anything special on save or restore. Alternatives exist that involve more considerations, and are not discussed here.

5.2.3 Switching Into a Task

When a task is being restored (that was switched out due to an interrupt, see above), the RTOS needs to take some steps to ensure the proper overlay is mapped if necessary prior to resuming, if the task PC points within an overlay.

To avoid affecting real-time behavior of higher priority tasks and interrupts, any required mapping of the overlay must occur in the context of a task, rather than doing it in the context-switch code itself with interrupts or pre-emption disabled. The suggested implementation is that it be done in the context of the task being resumed, by creating an appropriate call frame on the task stack, making the task call an appropriate C function at whatever point it was running. This is essentially the same thing as a signal in Unix and some RTOS, with the signal handler taking care of mapping the overlay. See Figure 5–4 above for details.

The RTOS invokes the *_xt_overlay_check_map* macro (or C function of the same name) to determine whether it is necessary to create such a stack frame. Given both the task's PC and its saved overlay state, the macro determines whether the PC is in an overlay, and if so, whether that overlay (as recorded in the overlay state) is already mapped. If the PC was in an overlay that isn't currently mapped, the OS must create a proper stack frame to cause the task to map the required overlay.

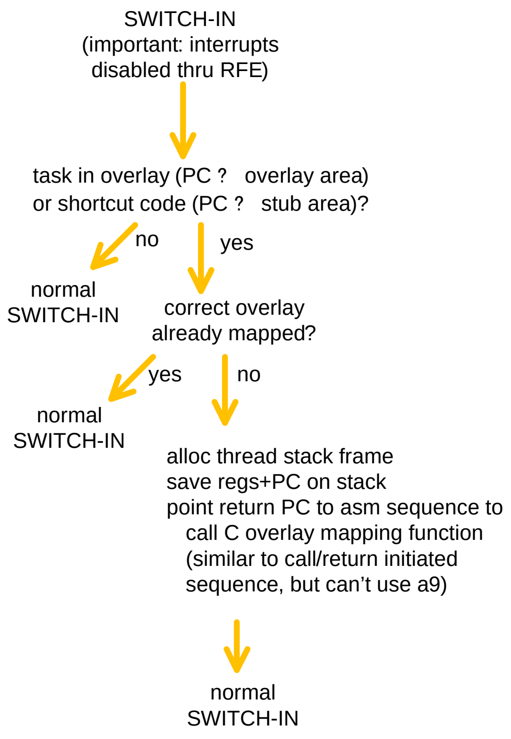
SWITCH-IN
(important: interrupts
disabled thru RFE)
task in overlay (PC ? overlay area)
or shortcut code (PC ? stub area)?
no
yes
normal
SWITCH-IN
correct overlay
already mapped?
yes
no
normal
SWITCH-IN
alloc thread stack frame
save regs+PC on stack
point return PC to asm sequence to
call C overlay mapping function
(similar to call/return initiated
sequence, but can't use a9)
normal
SWITCH-IN

Figure 5–5. Logic Flow of _xt_overlay_check_map() Macro

The *_xt_overlay_check_map* macro / function also restores the overlay state that was obtained (and then saved) on switch-out using *`_xt_overlay_get_state`*. It also takes as argument the task's stack pointer, and stores the task's original PC and PS below that stack pointer, where it can be retrieved to resume original task execution once the overlay has been mapped. The updated PC and PS are returned to the calling code. See Figure 5–5 above for details. The overlay manager provides an assembler sequence to create this stack frame and call the appropriate C routine, so that all the RTOS has to do is change the task's PC and PS.

IMPORTANT: Interrupts and/or pre-emption must be disabled when *`_xt_overlay_check_map`* is called, all the way through return to the task. The actual instruction used to return to the task must be done with interrupts disabled and must atomically re-enable interrupts and jump to the task PC and restore the task PS. This is to avoid any race conditions that arise if interrupts and pre-emption are enabled and task PC is temporarily outside any overlay while proceeding towards the task PC that is in an overlay, assuming the overlay to be mapped. (An interrupt during that sequence might cause a context-switch and cause another overlay to be mapped.) The instruction typically used at the end of this sequence is RFE (where PS.EXCM is set to disable interrupts prior to invoking RFE, which clears PS.EXCM), though it may possibly be one of the RFI n instructions (where either PS.EXCM and/or PS.INTLEVEL can be used to disable interrupts prior to invoking RFI, as RFI restores the entire PS).

5.2.4 OS-Provided Functions

See Section 5.1.8 for a list of OS-specific functions to be provided (either by the OS vendor, or by the user).

5.2.5 Overlay Mapping C Code

The C code that maps an overlay has RTOS dependencies. In particular, because it may be called concurrently by multiple tasks, it must protect access to the single overlay area shared among all tasks. It does this by calling `xt_overlay_lock()` and `xt_overlay_unlock()` (see Section 5.1.8), which typically use a lock or mutex, preferably one that supports priority inheritance.

The C code that handles the overlay mapping operation is part of the overlay manager library. It calls `xt_overlay_start_map()` and `xt_overlay_is_mapping()` to actually map (copy) the overlay.

5.2.6 API Calls for OS Support

The following functions are provided by the overlay manager to support OS operations. These are meant to be called only by OS code and not application code.

xt_overlay_get_state – Get Overlay State on Switch-Out

C synopsis:

```
#include <xtensa/overlay.h>
unsigned  xt_overlay_get_state (unsigned pc)
```

ASM synopsis:

```
#include <xtensa/overlay_os_asm.h>
_xt_overlay_get_state    pcreg, scratchreg1, scratchreg2
```

Returns the current overlay state (what's relevant to the current task). This state is a 32-bit opaque quantity that typically includes:

- (*e.g.*, 16-bits) ID of the overlay currently mapped, or in the process of being mapped, if any (in current implementation, this is 0 thru number of overlays – 1), or -1 if none.
- Indication of whether an overlay is fully mapped, or in the process of being mapped.
- Other flags as needed.

This routine is usually called when switching out from a pre-empted task (*i.e.,* due to an interrupt), and the overlay state (return value) so obtained is saved as part of the task context. This overlay state is normally restored by passing it to `xt_overlay_check_-map`.

Inline assembly arguments are all address registers (a0-a15):

`pcreg` contains the task's PC (*i.e.*, the point at which the task got interrupted). This register is clobbered and the return value is returned here. Scratch registers *sr1* and *sr2* must be distinct from *`pcreg`*.

Depending on implementation, `xt_overlay_get_state` may look at the PC to determine whether the task was in an overlay. (This check may also, or instead, be done in `xt_overlay_check_map`.)

Example:

```
overlay_get_state    a2, a3, a4       // get overlay state in a2, given
PC in a2; a3, a4 clobbered
```

xt_overlay_check_map

C synopsis:

```
    #include <xtensa/overlay.h>
        unsigned  xt_overlay_check_map (unsigned * pc,
        unsigned * ps, unsigned * ovstate, unsigned sp)
```

ASM synopsis:

```
    #include <xtensa/overlay_os_asm.h>
    _xt_overlay_check_map    pc_reg, ps_reg, ovstate_reg, sp_reg,
scratch_reg
```

Restores overlay state, and adjusts task return PC if needed to cause task to call the overlay mapping function. This routine is usually called from the scheduler when resuming a task.

It is passed the task's saved PC/PS, and returns either that same PC/PS, or an adjusted PC (to code that arranges to safely call the overlay mapping C function) and PS. In the latter case, the original saved PC/PS is written to the task's stack, where it can be picked up by the code at the adjusted PC.

Note: May store data under the task's stack pointer, at offsets -32 .. -21 relative to the `sp` provided.

On entry to asm macro:

pc_reg address register contains the task's return PC

ps_reg address register contains the task's saved PS

ovstate_reg address register contains the task's saved overlay state

sp_reg address register contains the task's stack pointer (its value in the task at the time of the interrupt, *not* an exception stack frame relative to it)

scratch_reg address register is undefined

On exit from asm macro:

pc_reg address register is either unchanged, or contains an adjusted return PC

ps_reg address register is either unchanged, or contains an updated PS value

ovstate_reg address register is clobbered

sp_reg address register is unchanged

scratch_reg address register is undefined

Example:

```
overlay_check_map   a3, a4, a2, a5, a6
```

5.2.7 Debugging Support

Xtensa Xplorer has full support for debugging overlays. Refer to the Xplorer help documentation for more details.

The Xtensa debugger (`xt-gdb`) has support for overlays. This is based on standard GDB overlay support, as described in Section 10.1 of the *GNU Debugger User's Guide*.

The GDB command `overlay auto` is used to set GDB into auto overlay mode. If you are using xt-gdb directly, then you must issue this command right at the beginning, as soon as you start xt-gdb. This will enable overlay support. If you are using Xtensa Xplorer, then this is automatically taken care of for you.

We recommend that you run xt-gdb with the command line option `--readnow` when debugging programs that use overlays. This forces xt-gdb to read in all symbols at startup. Normally, xt-gdb reads symbols only as needed, which can lead to problems in dealing with overlay code.

Setting breakpoints by address in the overlay area may lead to unexpected results. This happens because in the presence of overlays, there is not a unique mapping from address to source information, so xt-gdb may be unable to determine which overlay function you intended when you specified the address.

Using xt-gdb's `call` command to call overlay functions from the debugger prompt will not work correctly. The automatic overlay management was not designed to work under this condition.

If execution is stopped inside an overlay function called from another overlay function, the call stack may not be correctly displayed. Local variables from the calling function may not display correctly.

5.2.8 Other Tools

Some other tools are affected by the presence of overlays:

The C/C++ compiler command line option `-ipa` should not be used with files that contain overlay functions. This option currently conflicts with overlays and will generate errors. Source files in the project that do not contain overlay code can still be compiled with this option.

The tool xt-link-order (which performs profile-based section reordering and placement) will exclude all overlay sections from its analysis. This is automatic, no user action is required.

6. The Integrated DMA Library API

The IDMA library (iDMAlib) manages the Integrated DMA Controller (iDMA), which transfers data between local memories and the PIF. The processor can continue performing other tasks at the same time the DMA transfer is in progress. To know when the transfer completes, the iDMAlib can either poll the transfer status information, or it can use an iDMA transfer completion interrupt. Using the interrupt method enables usage of callbacks to process completed data and enables waiting in low-power mode (only XTOS support is provided).

The iDMA hardware logic receives a copy request in the form of an iDMA descriptor, which is of fixed size.The hardware fetches descriptors consecutively, one after another, as long as its control registers indicate there are more descriptors to process. There are three types of descriptors:

- One-dimensional (1D) descriptors contain the source address, the destination address, and the transfer size.
- Two-dimensional descriptors (2D) are used to copy matrices (tiles) and contain parameters such as row size, source pitch and destination pitch, which are used to specify a matrix row size and the distance between rows.
- A special JUMP command (descriptor) redirects the flow to another consecutive list of descriptors.

All three descriptor types are of different sizes. The iDMA hardware knows the location of the next descriptor to fetch after decoding the currently executing descriptor. More information on descriptors and the iDMA hardware is provided in the Integrated DMA chapter of the *Xtensa Microprocessor Data Book*.

Note: Refer to Appendix A for a list of deprecated, renamed, and dropped features.

6.1 Task-Execution Versus Fixed-Buffer Mode of Operation

The iDMAlib introduces two modes of operation:

- The *fixed-buffer* mode executes descriptors in turns from a single, pre-set looping buffer, which holds the array of descriptors. Descriptors are added to this buffer, existing ones are updated, and they are scheduled for execution.
- The *task-execution* mode executes independent copy requests from the array of descriptors by linking buffers to each other during their scheduling. Each buffer, or copy request, contains a number of descriptors added to it the same way they are added to the fixed-buffer.

Details on both modes are given in subsequent sections, and examples are given in Section 6.7.

An iDMA task is a structure containing one or more copy requests (descriptors) that the iDMAlib schedules for execution on behalf of an application or a thread. The task also contains control fields (for maintaining the task during its creation and execution), and the task status fields. A task is prepared in four steps:

1. Allocate memory for a task. This can be done by using the provided `IDMA_BUFFER_DEFINE()` macro, to allocate the correct memory size for a desired number of descriptors.
2. Initialize a task (shown as Step 1 for the task named *New Task*, in Figure 6–6). This step is needed for iDMAlib correct operation.
3. Add copy request to the task (shows as Step 2 in Figure 6–6) Repeat this step to add more requests (descriptors).
4. Schedule the task for execution by linking it to the previously added task using a special `JUMP` command (shown as Step 3 in Figure 6–6) followed by incrementing the number of descriptors to execute by writing to a hardware control register (shown as Step 4 in Figure 6–6). The task scheduling process links independent tasks and the iDMA hardware executes them one after another. This is because the hardware and iDMAlib do not have multi-channel capabilities, priority queues, and priority scheduling.

Note that tasks cannot be rescheduled without following all four steps again; for example, an application wanting to send the same task for execution again will not be able to execute Step 4 only.

Figure 6–6 also shows the `STATUS` field, which tracks the task progress. Other iDMAlib Task fields are not shown for simplicity.

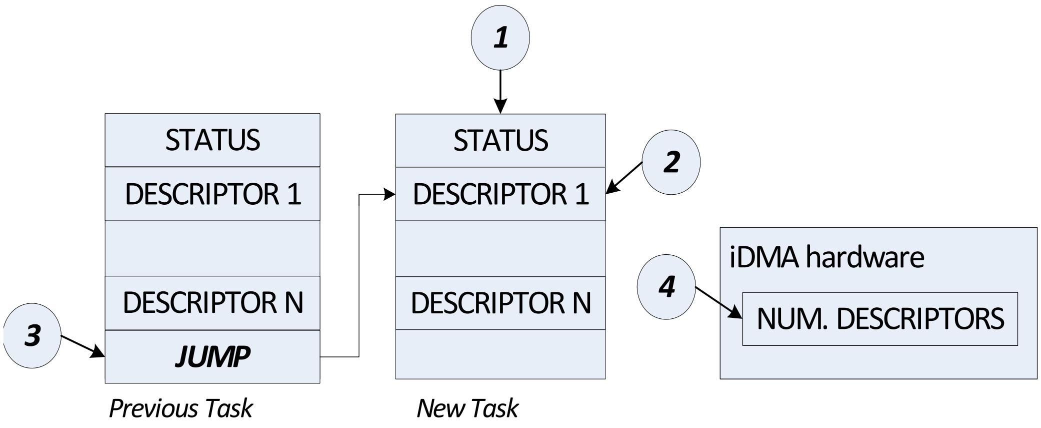

Figure 6–6. The iDMAlib Task Creation

Using the task structure allows independent applications, threads, or code segments to submit their copy requests to the iDMA logic independently of each other. Possible failure in one task execution does not affect other tasks as the failure is detected by iDMAlib followed by the optional recovery and automatic rescheduling of all tasks queued after the failing one.

6.1.1 Task Execution Completion

The iDMA logic can be set to raise two interrupts: the completion interrupt is raised on each descriptor completion (when the copy request is done), and the error interrupt is raised when the iDMA logic encounters an error during execution.

The application decides if the interrupts are to be raised by providing callbacks for each interrupt and setting descriptor options. For example, a callback is invoked on an interrupt only if a descriptor control field indicates that the interrupt is to be raised. For errors, the interrupt (and subsequently the callback) is invoked if the error callback is provided.

When an error or a completion interrupt occurs, or the application pushes the iDMAlib to process the API calls, the iDMAlib checks for completed descriptors and marks them as done. Each pending (scheduled) task status field is updated to reflect the number of outstanding descriptors. If the error is detected, the status field for the erroneous task is set to indicate the error including the error details as well. The iDMAlib can stop task execution on an error, or it can automatically reschedule the tasks past the failing one. Note that the iDMAlib drops all non-completed descriptors from the erroneous task.

The application can choose not to raise interrupts if they are adding significantly to the overall execution processing time. Instead, status calls can be used. Using these calls, the application is able to check the status of each task. For example, not using the error callback will completely drop the error interrupt. However, if the completion interrupt is enabled in a descriptor, the interrupt will occur although allowing the iDMAlib to continue internal processing. Without the interrupt, the user must push the iDMAlib internal task processing mechanism.

6.2 Fixed-Buffer Mode

An application can pre-set a fixed circular buffer of descriptors that is not going to be changed afterwards. For example, when a CPU processes an image and stores each processed chunk to the local memory, the iDMA hardware can be responsible for moving these chunks further to the main memory where the whole image is stored. Another example is when an I/O brings data into a circular buffer in main memory while it also transfers data to the circular buffer in local memory, for further processing.

Following is an example of image processing where the CPU stores processed chunks (of size `ChunkSize`) to the buffers in the local memory (`LB1` and `LB2` in Figure 6–7) while the iDMAlib transfers them further to the main memory and fills the whole image (`IMG` in Figure 6–7).

Double-buffering is used for performance so the CPU fills `LB1` and `LB2` in turns. To support such a system, the application creates a single task with two descriptors with their source addresses pointing to `LB1` and `LB2` while their destination addresses point to the image `IMG`, in the main memory. The `JUMP` command is added from `LB2` to `LB1`. The destination addresses are changed before each descriptor execution, in order to populate the whole image. The execution starts with the CPU creating the first chunk in `LB1` and scheduling the first iDMA transfer from `LB1` to `IMG[0]`. In parallel with the iDMA transfer, the CPU works on the second chunk and stores it to `LB2`. When `LB2` becomes ready, the application schedules another iDMA transfer from `LB2` to `IMG[ChunkSize]`. When the first iDMA transfer completes, the CPU can again store a chunk to `LB1` and initiate another iDMA transfer from `LB1` to memory. However, this time the destination address must be changed to `IMG[2*ChunkSize]`before scheduling the descriptor. This processes continues in turns - as soon as a descriptor completes, the CPU stores the next processed image chunk to the next available local buffer, updates the destination address of the next descriptor and schedules the next descriptor.

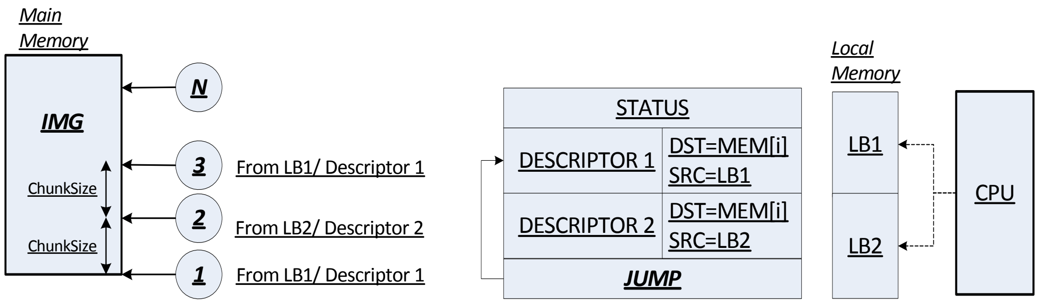

Figure 6–7. Fixed-Buffer Setup Example

As shown in the previous example, some use models require a single buffer with descriptors that loop during the execution, with possible update to descriptors based on a synchronization mechanism. The iDMAlib has a special API that sets a newly created task into this special looping mode. The mode is called *Fixed-Buffer* mode. After setting this mode, scheduling the task is not possible anymore but the application can only schedule a number of descriptors from the current task, that is, it can schedule the number of descriptors that follow the last scheduled one. Similar to the task-execution mode, the memory for a buffer that keeps descriptors must be allocated. During the iDMAlib initialization, the library receives a pointer to that buffer and continues to manage it onwards. Buffer can be allocated using the provided `IDMA_BUFFER_DEFINE()` macro that allocates the correct memory size for a desired number of descriptors.

6.2.1 Descriptors: First Add then Schedule Versus Add and Schedule

The iDMAlib has two incompatible use models while in the fixed-buffer mode:

- The *First Add then Schedule* use model assumes multiple descriptors are to be added to the loop buffer and then scheduled. This mode of operation requires iDMAlib to track both the location to which the next descriptor is to be added, and the location from which the next descriptor is scheduled.
- b) The *Add and Schedule* use model immediately schedules the added descriptor requiring iDMAlib to track only one pointer. While it may seem as a minor overhead to maintain two pointers, high performance applications can suffer significantly from such overhead.

The differences between these two models are shown in the examples in Section 6.7. In addition, the API described in Section 6.5.3 lists which functions in the fixed-buffer mode are incompatible with each other.

6.2.2 iDMA Execution Completion Using Interrupts

Similar to the task-execution mode, the iDMA logic can be set to raise the completion and error interrupts using callbacks and the descriptor control field. Alternatively, polling can be performed for each descriptor completion or execution errors. However, in comparison to the task-execution mode, if the interrupts are not enabled, there is no need to push the iDMAlib internal processing. Instead, the calls to obtain status can perform the iDMAlib housekeeping.

If an error is detected, the status for the buffer is set to indicate the error, including the error details as well. The iDMAlib stops the execution on an error, and the descriptors past the erroneous one are not scheduled. When an error occurs, the iDMAlib must be re-initialized.

6.3 Fixed-Buffer Versus Task-Execution Mode Comparisons

The fixed-buffer mode differs from the task-execution mode in the task prepare function, scheduling function, and iDMA error processing behavior, as described in Table 6–25.

Table 6–25. Fixed-Buffer Mode Versus Task-Execution Mode

	Fixed-Buffer Mode	Task-Execution Mode
Task prepare function	1. Task initialization in Fixed-Buffer mode 2. Add descriptors to the buffer (optional). Descriptors added in-order.	1. Task initialization in Task-Execution mode. 2. Add descriptors to the task.
Scheduling API function	1. Schedule a number of pre-set descriptors. 2. Schedule a new desc. by providing all copy args. *Descriptors are added/scheduled in-order.*	Schedule the task (in order, after the last scheduled one).
iDMA Error processing	The iDMAlib stops and requires the user to restart the iDMA hardware and iDMAlib.	The iDMAlib stops requiring the user to restart the iDMA hardware and iDMAlib.
iDMA Completion Processing	1. Check if a descriptor with an ID completed 2. Check the number of outstanding descriptors 3. Set callback on each completion interrupt	1. Check if any outstanding desc. in a task. 2. Set callback on each completion interrupt.

6.4 OS Integration

The iDMAlib depends on certain OS provided functions, which relate to interrupts (enabling and disabling them, and registering interrupt handlers) and waiting for descriptor completion in an OS dependent way—e.g. by utilizing thread blocking or low-power sleep. The iDMAlib offers a default implementation of the waiting functions for XTOS and XOS, resulting in the iDMAlib package to include three libraries; an XTOS default library, an XOS default library, and a library without any OS functions compiled in. See Section 6.5.4 for details on these functions and their default implementation.

6.5 Support for Two iDMA Channels

The processor may be configured to instantiate two iDMA channels. These channels are operated independently. The iDMAlib supports two-channel configurations by allowing the user to specify the channel number in each API call. Note that the iDMAlib function names are the same regardless of the number of channels—only the function signature changes. For the API to provide multichannel support (i.e., provide the channel argument), the processor must instantiate two iDMA channels, and the user program must define `IDMA_USE_MULTICHANNEL` before including `<xtensa/idma.h>`. If these conditions are not met, the produced API is kept compatible with older iDMAlib releases, and in case of multichannel configurations it will be accessing channel 0 only. Refer to Section 6.5.1 for details on `IDMA_USE_MULTICHANNEL`.

The iDMAlib API can be divided into five parts: task-execution related API, fixed-buffer related API, OS-related functions, common API (shared between task-execution and fixed-buffer modes of operation), and compile-time defines. All five are listed below. See the iDMAlib header file `idma.h` for more details about each function's arguments and return codes. Note that the majority of the functions return `idma_status_t` type, which indicates whether the function succeeded, or the error type if an error occurred.

6.5.1 The Common API

Following functions and defines are shared between the task-execution and fixed-buffer modes of operation. Section 6.9 gives details on each data type or structure used by the API described below.

IDMA_BUFFER_DEFINE

Allocate a memory for the buffer (task) containing descriptors to execute.

```
    IDMA_BUFFER_DEFINE(buffername, n, type);
```

Allocate memory for `n` either 1D or 2D descriptors depending on the `type` argument, and also set `buffername` as a pointer to the buffer structure.

IDMA_USE_MULTICHANNEL

A define whose presence will instruct iDMA to provide the multi-channel API; if the hardware implements multiple iDMA channels, and if this define is present, the iDMAlib will make API functions to provide an additional argument that indicates the channel to which the call is issued. This additional argument is always the first argument of a function. Below, the argument's presence in each function call is indicated with a comment.

iDMA channels numbering starts at zero. That is, to access the first iDMA channel (at channel zero) the channel argument to the API calls must have a value of zero.

idma_init

Initialize iDMAlib and iDMA hardware. Performs iDMA logic soft reset, sets iDMAlib parameters and sets iDMA logic registers with given parameters. Returns possible error during initialization.

```
idma_status_t idma_init (
          int ch,  // present only when multi-channel support is enabled
          unsigned             init_flags,
          idma_max_block_t     max_block_sz,
          unsigned             max_pif_req,
          idma_ticks_cyc_t     ticks_per_cyc,
          unsigned             timeout_ticks,
          idma_err_callback_fn err_callback);
```

Possible `init_flag` values are given in Table 6–26. Arguments `max_block_sz`, `max_pif_req`, `ticks_per_cyc` and `timeout_ticks` are settings for the iDMA hardware `Settings` and `Timeout` register. The argument `err_callbacks` assigns the callback to the iDMA error interrupt.

Table 6–26. Possible Values for `init_flags` Argument

Values for desc_flags	Description
`OCD_HALT_ON`	Enables iDMA halt on an OCD interrupt.

idma_stop

Stop iDMA operations and disable the idma hardware.

```
void idma_stop(void);
void idma_stop(int channel); // when multi-channel support is enabled
```

This function has no effect on the iDMAlib. If the iDMA logic is to be used again, it is expected that the iDMAlib be re-initialized using `idma_init()`.

idma_log_handler

Register log handler. All messages from the iDMAlib are sent to the application using the `log_handler()` calls. To use the log handler, the debug version of the iDMAlib must be used (refer to Section 6.8 on page 232).

```
void idma_log_handler(idma_log_h log);
```

When multi-channel support is enabled, the log messages will indicate the iDMAlib channel from which they originate.

Note: To enable logs from the functions that are implemented in `idma.h`, the application must be compiled with the `IDMA_DEBUG` compile flag. You must use the same compile flag as the one iDMAlib uses internally to enable logging, because many functions are provided in the iDMAlib header file.

idma_sleep

Attempt to put the core into the `WAITI` mode. The function returns immediately if there are no outstanding descriptors or if iDMA hardware is in error. The implementation uses the OS provided wait functionality (e.g. sleep or thread block) when the sleep argument to this function is set (see Section 6.5.4).

```
int idma_sleep (void);
int idma_sleep (int ch); // when multi-channel support is enabled
```

Note: This function requires the user to properly set the completion interrupt for each descriptor. Failure to generate an interrupt can cause the sleeping thread to never wake-up. Or, interrupts on each descriptor completion can cause unnecessary wake-ups.

Note: After returning from `WAITI` mode and until `idma_sleep()` returns, the processor executes with all interrupts enabled (interrupt level is set to zero).

idma_error_details

Obtain the error details. The data pointed to by the return value pointer provides the correct information only if the IDMA hardware is in error, thus this function should be used only when an error is indicated. (Note that the status reading functions are different for fixed-buffer and task-execution modes.)

```
idma_error_details_t* idma_error_details();
idma_error_details_t* idma_error_details(int ch); // when multi-channel
                                                    support is enabled
```

idma_add_desc

Add an iDMA 1D copy request to a buffer (fixed-buffer or the task-execution mode). Adds a descriptor to a consecutive location from the previously added one. Possible `desc_flags` values are given in Table 6–27.

```
idma_status_t idma_add_desc (
               idma_buffer_t* buffer,
               void* dst,
               void* src,
               int size,
               unsigned flags);
```

Table 6–27. Possible Values for `desc_flags` Argument

Values for desc_flags	Description
`DESC_IDMA_TRANSPOSE`	Transpose the matrix copied using 2D copy transfer.
`DESC_IDMA_SIZE_`*n*`B`	Granularity of iDMA transfers, in byte units. For transpose operation only. Regular 1D and 2D transfers are always in unit of one byte. *n* = 1,2,4,8
`DESC_IDMA_NOPRIV_SRC` `DESC_IDMA_NOPRIV_DST`	Privilege for source/destination access. By default, accesses are privileged. This flag sets non-privileged accesses.
`DESC_IDMA_PRIOR_H` `DESC_IDMA_PRIOR_L`	Priority of the iDMA accesses to PIF: high or low priority.
`DESC_IDMA_TRIG_OUT`	When set, iDMA hardware sends a pulse to *udma_trig_out* pin on descriptor completion.
`DESC_IDMA_TRIG_WAIT`	When set, current descriptor is not processed until *HaveTrig* control register becomes asserted.
`DESC_NOTIFY_W_INT`	On descriptor completion, generate the iDMA completion interrupt.

More information on the descriptor control word fields is provided in the Integrated DMA chapter of the *Xtensa Microprocessor Data Book*.

Note: In the fixed-buffer mode, this call is incompatible with `idma_copy_desc` and `idma_copy_2d_desc` functions.

idma_add_2d_desc

Add an iDMA 2D copy request to the buffer (either fixed-buffer or task mode) - adds a descriptor to a consecutive location from the previously added one. Possible `desc_flags` values are given in Table 6–27.

```
idma_status_t idma_add_2d_desc(
               idma_buffer_t* buffer,
               void* dst,
               void* src,
               int row_size,
               unsigned flags,
               unsigned num_rows,
               unsigned src_pitch,
               unsigned dst_pitch);
```

Note: In the fixed-buffer mode, this call is incompatible with `idma_copy_desc` and `idma_copy_2d_desc` functions.

idma_hw_num_outstanding

```
uint32_t idma_hw_num_outstanding (void);
uint32_t idma_hw_num_outstanding (int ch); // when multi-channel support is enabled
```

Check the number of outstanding DMA requests directly from the iDMA hardware, which bypasses any iDMAlib internal structure’s updates and housekeeping.

idma_hw_wait_all

```
void idma_hw_wait_all (void);
void idma_hw_wait_all (int ch); // when multi-channel support is enabledWait for
all the descriptors to finish by directly polling the iDMA hardware and
bypassing any iDMAlib internal structure’s housekeeping.
```

idma_hw_schedule

```
void idma_hw_schedule (uint32_t count);
void idma_hw_schedule (int32_t ch, uint32_t count); // when multi-channel
                                                   // support is enabled
```

Schedule a number of descriptors for execution by directly accessing the iDMA hardware and bypassing any iDMAlib internal structure’s housekeeping.

6.5.2 Task-Execution Mode API

idma_init_task

Initialize a task that contains `num` 1D or 2D descriptors. Also sets iDMAlib to the task-execution mode. This function should be called before adding descriptors to the task.

If multi-channel iDMA hardware support is enabled, the first argument specifies the iDMA channel which is initialized. It also links `task` (second argument) to that channel so later references to `task` will apply to the specified channel.

Note: When descriptors are added later to the task (using `idma_add_desc` and `idma_add_2d_desc`), exactly `num` descriptors need to be added. The need to match the task initialization arguments and the task preparation arguments exits to speedup iDMAlib internal processing.

```
idma_status_t idma_init_task (
          int ch, // present only when multi-channel support is enabled
          idma_buffer_t* task,
          idma_type_t type,
          int num, idma_callback_fn cb_func, void *cb_data;
```

idma_schedule_task

Schedule a task for execution. If the iDMA hardware is busy, the last descriptor from the last scheduled task is linked with the first descriptor from this task using a `JUMP` command. If the iDMA is not busy, the copy request starts with execution of this task.

```
idma_status_t idma_schedule_task(idma_buffer_t* task);
```

idma_copy_task

Create and schedule a 1D copy request. This is a convenience function that combines initializing the task, adding one descriptor to it, and scheduling the task. Possible `desc_flags` values are given in Table 6–27. The call will apply to the iDMA channel `ch`, if multi-channel support is enabled.

```
idma_status_t idma_copy_task (
          int ch, // present only when multi-channel support is enabled
          idma_buffer_t* task,
          void* dst,
          void* src,
          unsigned size,
          unsigned flags,
          void* cb_data,
          idma_callback_fn cb_func);
```

idma_copy_2d_task

Create and schedule a 2D copy request. This is a convenience function that combines initializing the task, adding one descriptor to it and scheduling the task. Possible `flags` values are given in Table 6–27.

```
idma_status_t idma_copy_2d_task (
          int ch, // present only when multi-channel support is enabled
          idma_buffer_t* task,
          void* dst,
          void* src,
          unsigned row_size,
          unsigned flags,
          uint32_t num_rows,
                unsigned src_pitch,
                unsigned dst_pitch,
                void* cb_data,
                idma_callback_fn cb_func);
```

idma_process_tasks

Trigger the iDMAlib internal processing. Necessary when no interrupts are available, or if interrupts are not enabled in each descriptor control settings. **Note:** The function will invoke callback function assigned to a task.

```
idma_status_t idma_process_tasks (void);
idma_status_t idma_process_tasks (int ch); // when multi-channel support is enabled
```

idma_task_status

Get the task execution status by reading the number of outstanding descriptors (not yet completed ones). When using the iDMAlib in the polling mode, the call to this function must be preceded by `idma_process_tasks()` in order for the iDMAlib to update status of each scheduled task. A negative value indicates an error.

```
task_status_t idma_task_status (idma_buffer_t* task);
```

idma_abort_tasks

Reset iDMA hardware and forcefully abort tasks in progress. All tasks in progress are marked as aborted. **Note:** Task may be in progress even if the iDMA hardware completed all its descriptors. To ensure all the completed tasks are not seen as in “progress“, enable the iDMA Done interrupt for at least the last descriptor in a task, or call `idma_process_tasks()` before calling this function. **Note:** The function will invoke the callback function assigned to any that gets processed.

```
void idma_abort_tasks ();
void idma_abort_tasks (int ch); // when multi-channel support is enabled
```

6.5.3 Fixed-Buffer Mode API

idma_init_loop

Initialize a buffer and set the iDMAlib to fixed-buffer mode. The function initializes the task and organizes it as a circular buffer.

If multi-channel iDMA hardware support is enabled, the first argument specifies the iDMA channel the loop buffer `buffer` is assigned to; later references to that buffer will apply to the specified channel.

```
idma_status_t idma_init_loop (
          int ch,  // present only when multi-channel support is enabled
          idma_buffer_t* buffer,
          idma_type_t type,
          int num,
          void *cb_data,
          idma_callback_fn cb_func);
```

The `type` (1D or 2D descriptor) and `num` (number of descriptors in the buffer) arguments must reflect values passed to the `IDMA_BUFFER_DEFINE()` macro.

Note: Functions to schedule copy request differ depending on the iDMAlib mode.

idma_schedule_desc

Schedule a number of consecutive descriptors for execution. Scheduled descriptors are those that follow the last scheduled one, with wrap-around. The number of descriptors to schedule should not be larger than the actual number of descriptors in the buffer.

```
int idma_schedule_desc(
          int ch,  // present only when multi-channel support is enabled
          unsigned count);
```

The return value indicates either an error, if it is less than zero (error type is `idma_status_t`), or the absolute index of the scheduled descriptor, counting from iDMAlib initialization (an unique ID) with a starting value of one. If `count` is more than one, the index indicates the last scheduled descriptor. E.g., if the buffer has eight descriptors total, the `count` is two, and the returned value is 17, this indicates the last descriptor in the buffer is scheduled, followed with a jump, followed with the first descriptor, and the iDMAlib has scheduled total of 17 descriptors from initialization.If the iDMAlib can not schedule a new descriptor (current ones are all in progress), a busy error status is returned.

Note: This call is incompatible with `idma_copy_desc` and `idma_copy_2d_desc` functions.

idma_schedule_desc_fast

Schedule a number of consecutive descriptors for execution. Scheduled descriptors are those that follow the last scheduled one, with wrap-around. The function assumes that additional descriptors will neither be added nor changed.

```
int idma_schedule_desc_fast(
                        int ch,  // present only when multi-channel support is enabled
                        unsigned count);
```

The return value indicates either an error, if it is less than zero (error type is `idma_status_t`), or the absolute index of the scheduled descriptor, counting from iDMAlib initialization (an unique ID) with a starting value of one. If `count` is more than one, the index indicates the last scheduled descriptor. If the iDMAlib can not schedule a new descriptor (current ones are all in progress), a busy error status is returned.

For example, if the buffer has eight descriptors total, the `count` is two, and the returned value is 17, this indicates the last descriptor in the buffer is scheduled, followed with a jump, followed with the first descriptor, and the iDMAlib has scheduled a total of 17 descriptors from initialization.

Note: This call is incompatible with the following functions: `idma_copy_desc`, `idma_copy_2d_desc`, `idma_update_desc_src`, `idma_update_desc_dst` and `idma_update_desc_size`.

idma_desc_done

Check if the descriptor is done using its unique index (made available as the return value from `idma_schedule_desc`).

```
int idma_desc_done (
                  int ch,  // present only when multi-channel support is enabled
                  int index);
```

The return value of one indicates the descriptor has completed, return value of zero indicates the descriptor has not yet completed, and a return value that is negative indicates an error in iDMAlib or iDMA hardware.

Note that this call triggers iDMAlib internal processing. Thus, it can be called in an empty loop while waiting for the given descriptor to complete.

idma_copy_desc

Update the next in line 1D descriptor and schedule it.

```
int idma_copy_desc (
          int ch, // present only when multi-channel support is enabled
          void* dst,
          void* src,
          size_t size,
          unsigned flags);
```

The return value indicates either an error, if it is less than zero (error type is `idma_status_t`), or the absolute index of the scheduled descriptor, counting from iDMAlib initialization (an unique ID) with a starting value of one. E.g., if the buffer has eight descriptors total and returned value is 16, this indicates the last descriptor in the buffer is scheduled, and the iDMAlib has scheduled total of 16 descriptors from initialization.If the iDMAlib can not schedule a new descriptor (current ones are all in progress), a busy error status is returned.

Note: This call is incompatible with `idma_add_desc`, `idma_add_2d_desc` and `idma_schedule_desc` calls.

idma_copy_2d_desc

Update the next in line 2D descriptor and schedule it.

```
int idma_copy_2d_desc (
               int ch, // present only when multi-channel support is enabled
               void* dst,
               void* src,
               int row_size,
               unsigned flags,
               unsigned num_rows,
               unsigned src_pitch,
               unsigned dst_pitch);
```

The return value indicates either an error, if it is less than zero (error type is `idma_status_t`) or the absolute index of the scheduled descriptor, counting from iDMAlib initialization (an unique ID). E.g., if the buffer has eight descriptors total and the returned value is 17, this indicates the first descriptor in the buffer is scheduled, and the iDMAlib has scheduled a total of 17 descriptors from initialization.

If the iDMAlib can not schedule a new descriptor (current ones are all in progress), a busy error status is returned.

Note: This call is incompatible with `idma_add_desc`, `idma_add_2d_desc` and `idma_schedule_desc` calls.

idma_update_desc_dst

Update the destination field of the next in line descriptor.

```
idma_status_t idma_update_desc_dst (
               idma_buffer_t* buf // present only when multi-channel support is enabled
               void* dst);
```

idma_update_desc_src

Update the source field of the next in line descriptor.

```
idma_status_t idma_update_desc_src (
             idma_buffer_t* buf // present only when multi-channel support is enabled
             void* src);
```

idma_update_desc_size

Update the source field of the next in line descriptor.

```
idma_status_t idma_update_desc_size (
               idma_buffer_t* buf // present only when multi-channel support is enabled
               unsigned size);
```

idma_buffer_status

Get the buffer execution status by reading the number of outstanding descriptors (not yet completed ones). If return value is less than zero, an error of type `idma_status_t*` occurred. Note that this call triggers iDMAlib internal processing.

```
int idma_buffer_status();
int idma_buffer_status(int ch); // present only when multi-channel support is enabled
```

idma_buffer_check_errors

Check if the iDMA hardware is in error. If yes, the function sets the error in the buffer status and returns the error code.

```
idma_errcodes_t idma_buffer_check_errors();
idma_errcodes_t idma_buffer_check_errors(int ch); // present only when
                                                   multi-channel support is enabled
```

6.5.4 OS-Provided Functions

The iDMAlib expects the target OS or application software to provide a way to deal with interrupts.

idma_thread_block

OS provided wait function, `idma_sleep`, will call this function. The default XTOS implementation is to simply execute `WAITI` and wait in low power mode. The default XOS implementation is to block the thread on an XOS event.

idma_thread_unblock

OS provided function, iDMAlib calls on each interrupt. It provides the OS with the ability to unblock threads set to sleep using a call to `idma_thread_block`. The default XTOS implementation does not implement this function as the interrupt itself is sufficient to wake up the core from `WAITI`. The default XOS implementation is to unblock the thread that waits for an XOS event.

idma_disable_interrupts

Disable the iDMA Error and Completion interrupts.

```
    extern void idma_disable_interrupts(void);
```

idma_restore_interrupts

Restore the iDMA Error and Completion interrupts.

```
    extern void idma_restore_interrupts(void);
```

6.5.5 Compile-time Defines

An application using the iDMAlib can be compiled with different compile-time defines. Some of the defines also affect the iDMAlib API execution since many of the library functions are implemented in the header file. Seven defines are provided:

IDMA_USE_DRAM1

Use DRAM1 memory to store iDMAlib code objects, such as buffers generated using `IDMA_BUFFER_DEFINE`, including the internal structures. The default is DRAM0.

IDMA_USE_INTR

Use interrupts in iDMAlib functions defined in the header file. Default is to use interrupts. This define incurs overhead of enabling/disabling interrupts to guard iDMAlib internal structures.

IDMA_DEBUG

Enable the debugging mode. Refers mainly to the debug prints that are not available without this macro—even if the log handler is assigned. Note that the iDMAlib deliverables include both the regular library and the one used for debugging (see Section 6.8). The debugging one is produced by compiling the iDMAlib with `IDMA_DEBUG` defined. The functions that exist in the header file, not in the library itself, need this define to operate in the debugging mode.

IDMA_APP_USE_XTOS and IDMA_APP_USE_XOS

These two defines automatically select XTOS or XOS functions that implement user provided functions for enabling and disabling interrupts. In addition, these defines automatically inline the related functions, if possible. If these macros are not used, the code to disable and enable an interrupt is executed by calling related functions from the respective iDMAlib libraries, which incurs an execution overhead. In the code, these macros need to be defined before the iDMAlib header file is included.

ALIGNDCACHE and IDMA_SIZE

Use these macros when allocating a copy buffer to avoid sharing cache lines between the buffer and the other data. The macro works by modifying the buffer alignment and its size. Refer to Section 6.6 for the use details.

6.6 Cache Coherency Issues

The iDMA hardware does not lookup or modify the caches, and so does not ensure cache coherency. For example, if the iDMA transfers a buffer B from local memory to main memory, accessing B in main memory may return previously cached data. Another example is sharing the same cache line between buffer B and an independent variable X (known as *false sharing*): a write to X may cause a reload of the whole cache line, bringing into the cache data that belongs to B, which may become stale with subsequent DMA operations to B. Moreover, a later cast out of that line will corrupt any DMA done to B in the meantime.

To avoid coherency issues, the application can disable caching all together. Since coherency issues are most likely only to appear during the application testing, the performance impact from disabling can be seen as unimportant.

Otherwise, correctly managing these coherency issues requires three things:

- **Allocation**. To avoid false sharing, allocate DMA buffers located in external memory in their own cache lines. One way to do this is to allocate such buffers so as to be aligned to the data cache line size, and sized as a multiple of the data cache line size.
- **Synchronize Writes**. If any cache lines covering a DMA buffer might be dirty, write-back and/or invalidate them before initiating DMA. You can use, for example, `xthal_dcache_region_writeback()`, `xthal_dcache_region_writeback_inv()`, or if the dirty data can be thrown away, `xthal_dcache_region_invalidate()`, Otherwise, DMA from this buffer will not see the dirty data, and cast outs (which can occur at any time) may overwrite DMA to this buffer.
  Note that where an application never writes to an external DMA buffer (other than using DMA), the corresponding cache lines are never dirty, and thus this step is not required. This is preferable for performance reasons.
- **Synchronize Reads**. Before reading from an external DMA buffer in cached memory, invalidate the corresponding data cache lines. This is usually done after the DMA rather than before, so that any spurious reads of the DMA buffer during DMA (for example, by a debugger or other process) do not cache the old data and cause stale data to be read later. There is normally no dirty data at this point, so you can use either `xthal_dcache_region_invalidate()` or `xthal_dcache_region_writeback_inv()`, for example.
  Note that where an application never reads from an external DMA buffer (other than using DMA), this step is not required. This is preferable for performance reasons.

For example, if you are transferring 100 bytes of data in the system where the cache line is 64bytes wide, instead of using:

```
char [100];
```

you may need to use:

```
char buffer [128] __attribute__ ((aligned(64)));
```

or more generally:

```
char buffer [IDMA_SIZE(100)] ALIGNDCACHE;
```

The `IDMA_SIZE()` and `ALIGNDCACHE` macros are available in the `idma.h` file.

6.7 iDMA Examples

The iDMAlib usage examples are in the `examples/libidma` directory within the installed configuration space. Each individual test describes its setup and purpose.

- Examples for the task-execution mode begin with `test_task*`. These examples set up multiple tasks to be executed at once, with possible callbacks, looping for tasks to end, etc.
- Examples for the fixed-buffer mode begin with `test_Dbuf*` and `test_fifo*`. The former creates the iDMAlib buffer as a simple double buffer where the descriptors are updated before scheduling, while the latter creates a deeper FIFO, and each descriptor schedule adds a whole new copy request.

The provided Makefile.example actually provides three sets of the compiled programs; two set utilize XTOS or XOS default iDMAlib support, while the third one (files `*-os.test`) is the iDMAlib implementation for XTOS with the OS-defendant functions being provided locally so they serve as an custom OS support example.

The next sections give the basic API usage on how to setup a task for execution in both the *task-execution* and *fixed-buffer* modes.

6.7.1 Task-Execution Setup Example

The following code sets two tasks each with a single 1D descriptor. *Task2* is scheduled first so the code must wait for *task1* to finish before proceeding.

```
IDMA_BUFFER_DEFINE(task1, IDMA_1D_DESC, 1);
IDMA_BUFFER_DEFINE(task2, IDMA_1D_DESC, 1);

ALIGNDCACHE char mem1 [IDMA_XFER_SIZE];
ALIGNDCACHE char mem2 [IDMA_XFER_SIZE];
ALIGNDCACHE char lbuf1 [IDMA_XFER_SIZE] IDMA_DRAM;
ALIGNDCACHE char lbuf2 [IDMA_XFER_SIZE] IDMA_DRAM;

int main () {

       idma_init(0, MAX_BLOCK_2, MAX_PIF_4, TICK_CYCLES_2, 0, idma_err_cb);
       idma_init_task(task1, IDMA_1D_DESC, 1, NULL, NULL);
       idma_add_desc(task1, _lbuf1, _mem1, size, DESC_NOTIFY_W_INT);

       /* task2 is prepared and schedule using a single call, before task1 */
       idma_copy_task(task2, _lbuf2, _mem2, size, 0, task2, cb_func);

       /* Now schedule task1 as well */
       idma_schedule_task(task1);

       /* Wait for task completion - can sleep if interrupts are setup or can
        poll while triggering iDMAlib internal processing */
       while (idma_task_status(task1) > 0) {
         idma_sleep() OR idma_process_tasks();
        }
        /* If completed due to an error, process the error */
        if (idma_task_status(task1) < 0) {
         idma_error_details_t* error = idma_error_details();
         ...
        }
}
```

6.7.2 Fixed-Buffer Mode Setup Example, Descriptor Add Then Schedule

The following code sets up a circular buffer with two 1D tasks. The descriptors are added and then a call is repeatedly made to schedule the next in line descriptor. This executes two descriptors in turns. For example, such a code can be used with a synchronization mechanism where the source double-buffers (`mem1/mem2` in the main memory) are updated with the new data coming from the I/O, and the CPU consumes arriving data from buffers in the local memory (`lbuf1/lbuf2`).

```
#define NUM_DESCS 2;
IDMA_BUFFER_DEFINE(buffer, NUM_DESCS, IDMA_1D_DESC);

int main () {
        idma_init(0, MAX_BLOCK_2, MAX_PIF_4, TICK_CYCLES_2, 0, err_cb);
        idma_init_loop(buffer, IDMA_1D_DESC, NUM_DESCS, cb_data, cb_func);
        idma_add_desc(buffer, lbuf1, mem1, size, DESC_NOTIFY_W_INT);
        idma_add_desc(buffer, lbuf2, mem2, size, DESC_NOTIFY_W_INT);while (1)
{

         /* Wait for at least one free descriptor (not checking for error) */
         if (idma_buffer_status() == NUM_DESCS)
               continue;

         /* Schedule next in line descriptor */
         idma_schedule_desc(1);

         /* Optionally update descriptors and data in the memory buffers */
         E.g. idma_update_desc_src(new_src);
        }
}
```

Note that Section 6.2.1 introduced two different and incompatible ways to add and schedule descriptors. The example above matches the first approach, where multiple descriptors are added first and then are scheduled at once (only one descriptor is scheduled in this example).

6.7.3 Fixed-Buffer Mode Setup Example, Descriptor Add And Schedule

The following code sets up a circular buffer with two 1D tasks. The descriptors are added and scheduled at once. This executes descriptors sequentially. For example, such code can be used for independent scheduling of different descriptors where return value gives an unique ID that can be used later to check on a particular descriptor status.

```
#define NUM_DESCS 2;
IDMA_BUFFER_DEFINE(buffer, NUM_DESCS, IDMA_1D_DESC);

int main () {
       idma_init(0, MAX_BLOCK_2, MAX_PIF_4, TICK_CYCLES_2, 0, err_cb);
       idma_init_loop(buffer, IDMA_1D_DESC, NUM_DESCS, cb_data, cb_func);

       while (1) {
        /* Repeatedly keep scheduling, overflow returned if busy */
        int ret = idma_copy_desc(dst_ptr, src_ptr, size, 0);

        printf("Added descriptor @ index:%d\n", ret);

        /* Wait until Nth descriptor is completed to process data. Can either
        sleep waiting for interrupts or trigger processing repeatedly. */
        while(!(idma_desc_done(N)) {
              idma_sleep OR idma_buffer_status();
        }

        // Process Nth data chunk here, update src_ptr, dst_ptr, size;
       }
}
```

The example above matches the second approach in Section 6.2.1, where a descriptor is added and scheduled at once.

6.7.4 Example for Two-channels IDMA

The following code issues two iDMA copy requests, using two channels. It also waits for their completion.

```
IDMA_BUFFER_DEFINE(buf0, 2, IDMA_1D_DESC);
IDMA_BUFFER_DEFINE(buf1, 2, IDMA_1D_DESC);

int main () {
       idma_init(IDMA_CHANNEL_0, 0, MAX_BLOCK_2, 16, 0, 0, NULL);
       idma_init(IDMA_CHANNEL_1, 0, MAX_BLOCK_2, 16, 0, 0, NULL);

       idma_init_loop(IDMA_CHANNEL_0, buf0, IDMA_1D_DESC, 64, NULL, NULL);
       idma_init_loop(IDMA_CHANNEL_1, buf1, IDMA_1D_DESC, 64, NULL, NULL);

       idma_copy_desc(IDMA_CHANNEL_0, dst0, src0, 64, 0);
       idma_copy_desc(IDMA_CHANNEL_1, dst1, src1, 64, 0);
```

```
        while (idma_hw_num_outstanding(IDMA_CHANNEL_0) > 0) {}
        while (idma_hw_num_outstanding(IDMA_CHANNEL_1) > 0) {}
}
```

6.8 Compilation and Linking

To use the iDMA library, user code must include the library header file (`idma.h`) and must be linked with the iDMA library. The iDMA library package includes six iDMA libraries:

libidma.a:	iDMAlib without any OS integration. The main application must provide OS required functions (see Section 6.5.4).
libidma-xtos.a:	iDMAlib with the default XTOS implementation, as described in Section 6.5.4.
libidma-xos.a:	iDMAlib with the default XOS implementation, as described in Section 6.5.4.
*libidma-debug-*a:*	Three variants of the iDMAlib above also come in the debug version; a logger (see “idma_log_handler” on page 216) can be registered to receive debug logs.

Users must specify the library explicitly when invoking the linker using `xt-ld`, `xt-gcc`, or `xt-xcc`. This is usually done by specifying the library using the `-l` option, e.g. `lidma`, to one of these commands (assuming the default linker paths). In Xplorer, simply add a library entry in the Libraries tab in your Build Target, then select the desired library by checking the box.

6.9 iDMAlib Structures

IDMA descriptor completion callback:

```
     typedef void (*idma_callback_fn)(void* arg);
```

IDMA hardware error callback:

```
     typedef void (*idma_err_callback_fn)(
                    const idma_error_details_t* error);
```

Descriptor buffer structure:

```
typedef struct {
      char buffer[4];
} idma_buffer_t;
```

Type of descriptor:

```
typedef enum {
    IDMA_1D_DESC  = 1, // use 1D descriptors
    IDMA_2D_DESC  = 2  // use 2D descriptors
} idma_type_t;
```

Maximum allowed PIF request block size:

```
typedef enum {
    MAX_BLOCK_2  = 0,   // use max PIF block size of 2
    MAX_BLOCK_4  = 1,   // use max PIF block size of 4
    MAX_BLOCK_8  = 2,   // use max PIF block size of 8
    MAX_BLOCK_16 = 3    // use max PIF block size of 16
} idma_max_block_t;
```

IDMA hardware internal timer tick period:

```
typedef enum {
    TICK_CYCLES_1  =  0,   // Timer ticks every 1 cycle
    TICK_CYCLES_2  =  1,   // Timer ticks every 2 cycles
    TICK_CYCLES_4  =  2,   // Timer ticks every 4 cycles
    TICK_CYCLES_8  =  3, // Timer ticks every 8 cycles
    TICK_CYCLES_16  = 4, // Timer ticks every 16 cycles
    TICK_CYCLES_32  = 5, // Timer ticks every 32 cycles
    TICK_CYCLES_64  = 6, // Timer ticks every 64 cycles
    TICK_CYCLES_128 = 7  // Timer ticks every 128 cycles
} idma_ticks_cyc_t;
```

Return values for iDMA API calls that return `idma_status_t` type:

```
typedef enum {
  IDMA_ERR_BAD_DESC = -20, // Descriptor not correct
  IDMA_ERR_NOT_INIT ,      // iDMAlib and HW not initialized
  IDMA_ERR_TASK_NOT_INIT,  // Cannot scheduled uninitialized task
  IDMA_ERR_BAD_TASK,       // Task not correct
  IDMA_ERR_BUSY,           // iDMA busy when not expected
  IDMA_ERR_IN_SPEC_MODE,   // iDMAlib in unexpected mode
  IDMA_ERR_NOT_SPEC_MODE,  // iDMAlib in unexpected mode
  IDMA_ERR_TASK_EMPTY,     // No descs in the task/buffer
  IDMA_ERR_TASK_OUTSTAND_NEG, // Number of outstanding descs is negative
  IDMA_ERR_TASK_IN_ERROR,   // Task in error
  IDMA_ERR_BUFFER_IN_ERROR, // Buffer in error
  IDMA_ERR_NO_NEXT_TASK,    // Next task to process is missing
  IDMA_ERR_BUF_OVFL,        // Attempt to schedule too many descriptors
  IDMA_ERR_HW_ERROR,        // HW error detected
  IDMA_ERR_BAD_INIT,        // Bad idma_init args
  IDMA_OK = 0               // No error
} idma_status_t;
```

Return values for iDMA API calls that return `task_status_t` type:

```
typedef enum {
  IDMA_TASK_ABORTED = -3, // Task aborted (a preceding task caused error)
  IDMA_TASK_ERROR   = -2, // Task in error.
  IDMA_TASK_NOINIT  = -1, // Task not initialized before schedule
  IDMA_TASK_DONE    =  0, // Task completed (valid after been scheduled)
  IDMA_TASK_EMPTY   =  0  // No descriptors in a task
} task_status_t;
```

Error details structure:

```
typedef struct
  idma_hw_error_t err_type; // ErrorCodes field of the iDMA Status
  uint32_t currDesc;        // Descriptor that caused error
  uint32_t srcAddr;         // AXI source address causing error
  uint32_t dstAddr;         // AXI destination address causing error
} idma_error_details_t;
```

Possible value for the `idma_hw_error_t` field of `idma_error_details_t`:

```
#define IDMA_ERR_FETCH_ADDR      0x2000
#define IDMA_ERR_FETCH_DATA      0x1000
#define IDMA_ERR_READ_ADDR       0x800
#define IDMA_ERR_READ_DATA       0x400
#define IDMA_ERR_WRITE_ADDR      0x200
#define IDMA_ERR_WRITE_DATA      0x100
#define IDMA_ERR_REG_TIMEOUT     0x80
#define IDMA_ERR_TRIG_OVFL       0x40
#define IDMA_ERR_DESC_OVFL       0x20
#define IDMA_ERR_DESC_UNKNW      0x10
#define IDMA_ERR_DESC_UNSUP_DIR  0x8
#define IDMA_ERR_DESC_BAD_PARAMS 0x4
#define IDMA_ERR_DESC_NULL_ADDR  0x2
#define IDMA_ERR_DESC_PRIVILEGE  0x1
#define IDMA_NO_ERR              0x0
```

IDMA hardware state `idma_state_t` returns one of the idma states below, further described in the Xtensa Databook:

```
#define IDMA_STATE_IDLE 0x0u    // idma disabled, no copy requests
#define IDMA_STATE_STANDBY 0x1u // transient state
#define IDMA_STATE_BUSY 0x2u    // idma busy copying
#define IDMA_STATE_DONE 0x3u    // idma done but enabled
#define IDMA_STATE_HALT 0x4u    // idma disabled while copying
#define IDMA_STATE_ERROR 0x5u   // idma hardware in error
```

7. Building Multi-core Executables with XTOS

This feature allows the user to build and run a single executable on a multi-core system. The application runs in parallel on all the cores, somewhat like a multi-threaded application on a single core (except that the timing is different because the cores truly run concurrently). This style of application is useful in many cases. For example, there may be too much data to process with one core and still meet timing constraints. In such a case the same code could run on multiple cores and each core would process a portion of the data. Or, several different operations might be needed on the same data. In this case the cores would run different processing functions, but process the same (shared) data.

XTOS supports simple multi-core applications out of the box on a restricted subset of system configurations. Such an application can be compiled and linked in the same way as a regular single-core application, except that it has to be linked with a special linker script (LSP). Two prepackaged LSPs are provided to get started, and these can be used as the base for creating custom LSPs.

XTOS multi-core applications have one copy of the code and global data, with each core having its own copy of private data, and each core having its own stack. XTOS internal state is per-core, so for example interrupt and exception handlers may be installed differently on each core, and interrupts may be enabled independently on each core as well. The PRID register can be read to determine the core ID. C library functions may be freely used as long as the locking is properly supported (discussed in Section 7.7.1).

This framework does not provide any software cache coherency. Access to shared data within the application must be guarded with mutual exclusion, and the data cache must be invalidated and written back explicitly as needed if hardware cache coherence is not available. Alternatively, shared data may be accessed uncached if system characteristics and performance allow it.

7.1 Requirements

The use model for the current single-image multi-core support imposes the following requirements on the system:

- All the Xtensa cores in the multi-core cluster must be of identical configuration.
- All the cores must have the same memory map, i.e. have the same types of memories located at the same addresses.
- All the cores must have a data RAM configured. The data RAM is necessary to support per-core private data, since there is no other mechanism for thread-local (core-local) data.

- The cores must be configured with the Processor ID option so that the PRID register is present. This is the only way for software to discover which core it is running on. The processor with PRID 0 (zero) is designated as the master.
- The software build must be configured with the Xtensa C Library (xclib).
- If shared access to data or resources is to be controlled via mutual exclusion, then the Exclusive Store option must be configured, together with the necessary hardware support for some region of shared memory.

7.2 Program Model

The simple program model has the following characteristics:

- The code in the executable image is shared by all the processors. This includes the reset handler. Some of the code may be placed in local instruction memory, in which case each core has its own copy, but this code is still identical across all of them. After system startup, different cores may load different libraries / overlays.
- All data, read-only data, and BSS sections not explicitly placed in data RAM are shared; that is, there is only one copy shared by all the processors. There is a single shared heap. This assumes there is shared memory. If the system has been configured such that there is only per-core data RAM available, then each core will get a copy of all data/rodata/BSS sections, and there will be no data sharing at all. Global variables will not work in this situation because effectively all data becomes core-local.
- Processor-local data is located in data RAM for each core. There is currently no other support for processor-local data.
- The stack may be placed in shared memory or in local data RAM. The default stack location and size can be overridden by the application developer, as described in Section 7.6.
- XTOS internal data is placed in data RAM since this is per-core and cannot be shared.
- The C library's internal context structure and the pointer to it are both placed in data RAM since there must be one copy per core. However, all data related to file I/O is global, since files are treated as global and handles are shared across all cores. All other C library data is global and placed in shared memory.
- Since the C library file I/O and memory allocator are both global, locks are needed to provide mutually exclusive access. These locks are implemented by XTOS, but the locks must be placed in specific memory areas to work properly. This is described in Section 7.7.1.

7.3 System Startup

It is assumed that only the master (core 0) is running when the system comes out of reset. The master initializes system-level hardware (such as the L2 cache controller etc.), unpacks program sections from ROM if needed, clears all shared BSS sections and initializes the C library. Once it gets to the main() function, application code is expected to release the other cores from reset in some system-specific manner.

Each core initializes its local BSS sections and unpacks local memory code and data sections from ROM if needed. Every core other than core 0 performs a limited initialization of the C library for its own context.

7.4 Predefined Linker Scripts

Two predefined linker scripts (LSPs) are provided for building single-image multi-core applications. These are adequate for getting started, but will likely need to be customized for final use.

The `sim-mc` LSP is very similar to the `sim` LSP, except that it places certain data and BSS sections in data RAM to make them private per-core, as described in Section 7.2. The `sim-mc-stacklocal` LSP is very similar, the only difference being that the stack is placed in data RAM instead of in external shared memory.

If you wish to create LSPs for your own custom targets based on standard LSPs (e.g. `min-rt`) then examine the difference between the `sim` and `sim-mc` LSPs (the `memmap.xmm` and `specs` files). Make a copy of the LSP you wish to start from (e.g. `min-rt`) and then apply the same set of changes to this LSP, and then use `xt-genld-scripts` to update the LSP in place.

The details of the memory sections and how they can be placed is described in Section 7.7.

7.5 Initializing BSS Sections

BSS sections can be in shared memory (global) or private memory. All global BSS sections are initialized by core 0. Private memory sections (which can only be in data RAM at this time) are initialized by each core. This is implemented in the `__bss_init` user hook (`bss-init-simc.S`) that gets called from the `_start` routine. This user hook can be overridden if you want to provide your own custom BSS init procedure.

7.6 Setting up the Stack

The stack can be located either in data RAM or in shared memory. If the stack is in data RAM, all cores can use the same address for the initial stack pointer. If the stack is not in data RAM, it is assumed to be in shared memory. The default stack setup code calculates the stack pointer for each core (except core 0) by reserving `__stack_size` bytes for each core, top down from the value of `__stack`. That is, the stack for core 0 begins at `__stack` and grows down for `__stack_size` bytes. The stack for core 1 begins here and grows further down for another `__stack_size` bytes, and so on. As an example, in a 4-core system with `__stack = 0x64000000`, and `__stack_size = 0x1000`, the initial stack pointers are assigned as follows:

```
Core 0: 0x64000000

Core 1: 0x63FFF000

Core 2: 0x63FFE000

Core 3: 0x63FFD000
```

The default stack assignment is implemented by `__stack_init` (`stack-init-simc.S`) and can be overridden if so desired. The default stack size is 4 Kbytes, and this also can be overridden by redefining the value of the symbol `__stack_size`.

7.7 Placing Data and BSS Sections

These sections need special placement for proper multi-core operation.

Section Name	Description
.rtos.percpu.data .rtos.percpu.bss	XTOS internal state, private per-core. Must be placed in private memory (dataram).
.clib.data .clib.bss	Xclib global data. Only one copy must exist. Must be placed in shared memory. May require special alignment if hardware cache coherence is not available.
.clib.percpu.data .clib.percpu.bss	Xclib context data, private per-core. Must be placed in private memory (dataram).
.xtos.lock	XTOS lock section. Must be placed in shared memory that supports exclusive access.

The `sim-mc` and `sim-mc-stacklocal` LSPs take care of proper placement for these sections. The `.xtos.lock` section is placed along side the `.data` section, making the assumption that the memory containing the `.data` section supports exclusive store or conditional store. If your system memory layout is different, then you must create your own LSPs and assign these sections appropriately. Refer to the *Xtensa Linker Support Package Reference Manual* for information on how to customize LSPs.

7.7.1 Placing the Lock Variables

Proper placement of the lock variables is necessary for proper operation of the multi-core framework, especially the C library shared-resource functions such as file I/O and dynamic memory allocation. The section containing the lock variables must be placed in memory that supports exclusive stores. For proper operation, this memory must be mapped uncached or cached with hardware coherence. Mapping the memory cached without hardware coherence will not work.

7.7.2 Using the C Library Without Locks

If file I/O and dynamic memory allocation are not used at all, then locking is not required for the C library. If you provide a definition for `xtos_locks` in your application code, then the XTOS implementation will not be linked in.

7.8 A Simple Example

An example multicore application is provided in the XTOS source tree in the `<xtensa_tools_root>/xtensa-elf/src/xtos/examples/multicore` directory. This example is set up to run on a 4-core system modeled in XTSC (Xtensa SystemC simulator). This example will only work if the Xtensa configuration you selected has all the properties specified in the Requirements section. The example is set up to work with the `sample_controller` configuration provided with Xtensa Xplorer.

To build the example, open a command shell from Xtensa Xplorer and navigate to the `xtos/examples/multicore` directory. Xplorer will already have set the tool paths and configuration parameters in the shell environment. Now type "`xt-make clean`" followed by "`xt-make run`". This will build and run the example.

See the README file included with the example for detailed directions on how to run and modify the example.

A. Deprecated, Dropped, and Renamed Features

Table A–28 lists the status for features that are deprecated, dropped, and renamed in the RI-2018.0 release.

Table A–28. Feature and Status

Feature	Status	Substitute	Reference
_xtos_ints_on	Dropped	xtos_interrupt_enable	Section 2.2.1
_xtos_ints_off	Dropped	xtos_interrupt_disable	Section 2.2.1
_xtos_read_ints	Dropped	xtos_interrupt_enabled xthal_get_interrupt	Section 2.2.4 Section 2.2.1
_xtos_clear_ints	Dropped	xtos_interrupt_clear xthal_set_intclear	Section 2.2.4 Section 2.2.1
_xtos_timer_<N>_delta	Dropped	xthal_set_ccompare	Section
XCHAL_OP0_FORMAT_LENGTHS	Deprecated	XCHAL_BYTE0_FORMAT_LENGTHS	Section 3.13.1
xthal_op0_format_lengths	Deprecated	xthal_byte0_format_lengths	Section 3.13.1
xthal_get_intenable	Deprecated	xthal_interrupt_enabled	Section 3.7.3
xthal_set_intenable	Deprecated	xthal_interrupt_enable or xthal_interrupt_disable	Section 3.7.3
xthal_get_interrupt	Deprecated	xthal_interrupt_pending	Section 3.7.3
xthal_set_intset	Deprecated	xthal_interrupt_trigger	Section 3.7.3
xthal_set_intclear	Deprecated	xthal_interrupt_clear	Section 3.7.3
xthal_dcache_block_ prefetch_for_modify	Renamed	xthal_dcache_block_ prefetch_modify	Section 3.11.4
idma_add_copy	Dropped		Chapter 6
idma_add_2d_copy	Dropped		Chapter 6
idma_task_error_details	Dropped		Chapter 6
idma_wait_desc	Dropped		Chapter 6
idma_task_t	Dropped	idma_buffer_t	Chapter 6
IDMA_TASK_SIZE	Dropped	IDMA_BUFFER_SIZE	Chapter 6
IDMA_BUSY_CHECKING	Dropped		Chapter 6
idma_copy	Renamed	idma_copy_task	Section 6.5.2
idma_2d_copy	Renamed	idma_2d_copy_task	Section 6.5.2
IDMA_HW_NUM_OUTSTANDING	Renamed	idma_hw_num_outstanding	Section 6.5.2
IDMA_HW_WAIT_ALL	Renamed	idma_hw_wait_all	Section 6.5.2
IDMA_HW_SCHEDULE	Renamed	idma_hw_schedule	Section 6.5.2

Index