

Leon

Leon Drivers

*Make sure to update all fields on Title page from File->Properties Menu.*

*Once updated right click and select update field to force update*

*\*Note: This text will not print. Do not delete.*

|  |  |
| --- | --- |
|  |  |
| **Authors:** | David Meggy |
| **Project:** | Lionsgate/Golden Ears |
| **Date Saved:** |  |

|  |  |
| --- | --- |
| **Reviewed by:** |  |
| **Approved by:** |  |

CONFIDENTIAL

The information contained herein is confidential and shall not be revealed to any third party without the expressed written consent of Icron Technologies Corporation. The document and all copies shall be returned to Icron promptly upon request by Icron.

Contents

1 Introduction 1

1.1 Purpose 1

1.2 Scope 1

1.3 Terminology References 1

1.4 Document References 1

2 Design Overview 2

2.1 Background Information 2

2.2 System Evolution Description 2

2.3 Current Process 2

2.4 User Characteristics 2

2.4.1 User Problem Statement 2

2.4.2 User Objectives 2

2.5 Proposed Process 2

2.6 Constraints 2

2.7 Design Trade-offs 2

3 Design Architecture 3

3.1 UART 3

3.2 Timers 3

3.3 Serial Flash Interface 3

3.4 Interrupts and Traps 3

3.5 CPU 3

3.6 Startup 3

4 External Module interface 4

4.1 Use model 4

4.2 API/Data Structures 4

4.2.1 UART 4

4.2.2 Timers 4

4.2.3 Serial Flash Interface 4

4.2.4 Interrupts and Traps 4

4.2.5 CPU 4

4.2.6 Startup 4

5 Detailed Design 5

5.1 UART 5

5.1.1 Processing 5

5.1.2 Local data structures 5

5.2 Timers 5

5.2.1 Processing 5

5.2.2 Local data structures 6

5.3 Serial Flash Interface 6

5.3.1 Processing 6

5.3.2 Local data structures 6

5.4 Interrupts and Traps 6

5.4.1 Processing 6

5.5 CPU 6

5.5.1 Processing 6

5.5.2 Local data structures 6

5.6 Startup 7

6 Testing 8

6.1 Test Plans 8

6.1.1 UART 8

6.1.2 Timers 8

6.1.3 Serial Flash Interface 8

6.1.4 Interrupts and Traps 8

6.1.5 CPU 8

6.1.6 Startup 8

6.2 Integration Plan 8

# Introduction

This document describes the low level Leon drivers. The Leon is RTL block that contains the Leon2 SPARC processor, and the accompanying peripherals: UART, timer, interrupt controller, and SFI. Within Icron the Leon block has being modified from the original Leon2 from Gaisler to meet the requirements of the Icron projects.

## Purpose

Exposed API to users of the Leon driver is the main purpose in this document. The secondary purpose is describing the design for future maintenance, and describing the implemented test cases which are used to verify the design and implementation.

## Scope

This document contains all aspects on the Leon drivers, including the export API, the internal design, and the test methods.

## Terminology References

SFI – Serial Flash Interface

KLEON – Keith’s modified Leon

usec – microsecond

msec – millisecond

ibuild – Icron build system

ilog – Icron logging system

ibase – the base Icron C component

## Document References

Leon2 Processor Users’ Manual: <https://shamrock.icron.local/twiki/pub/Main/Structured1Project/leon2-1.0.30-xst.pdf>

U:\Projects\Structured-1\90-00362 SFI Serial Flash Interface Design Specifcation\90-00362-A08.docx

U:\Projects\Structured-1\90-00364 Processor Design Specification\90-00364-A04.doc

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

# Design Overview

## Background Information

The Leon RTL block encompasses everything needed by the firmware running on the Lionsgate1 chip. The design of the RTL is to allow the possibility of a future replacement of the entire Leon RTL block with another processor and its supporting basic peripherals. This Leon driver is an attempt to split out all low level code related to the Leon RTL into this component where a future RTL update could be restricted to replacing just this module. Emphasis is then placed on to ensuring that the exposed API is as portable as possible.

## System Evolution Description

One of basic design criteria is allowing the system to change under the hood while ensuring that the exposed API stays constant. Previously all the details on the Leon implementation and register definitions were exposed, and this new design attempts to correct that.

## Current Process

The previous process exposed all register definitions so any piece of code in any module could write to any Leon register.

## User Characteristics

### User Problem Statement

A portable low level driver infrastructure is needed for future projects that are based on the Lionsgate ASIC.

### User Objectives

1. Portable low level drivers that encapsulate their internals
2. Reduce the code footprint of the previous drivers
3. Maintain deterministic behavior or document otherwise when not possible

## Proposed Process

The new process starts with using ibuild and ensuring that the exposed API is very controlled through the exposed header files. This is to ensure the 1st objective of creating a portable system that is well encapsulated. The remaining objectives are reflected in the many internal design decisions made in the implementation of the component.

## Constraints

The Leon drivers are the close to the lowest level code in the system. The only dependency is on ibase, which provides common C types, common C macros, common C structures, and basic C functions.

## Design Trade-offs

The main emphasis’s were on encapsulation and determinism. The encapsulation in some cases may restrict opportunity inlining function optimizations. Also a choice had to be made as to which functions are in .ftext, and which functions are just in .text.

# Design Architecture

## UART

The UART module of the Leon drivers provides the caller a mechanism to use UART at any baud rate, and in both polling and interrupt driven modes. It also exposes API for the multi-byte atomic transmits that match the requirements for the iLog backend.

To ensure a basic level of performance transmitted bytes are buffered up until ready to be sent out. As the majority of users of the Leon drivers only care about transmitting performance, the same is not done for the receiver.

## Timers

The timer module of the Leon drivers provides 3 basic purposes: measuring passed time, delay for set amount of time, and managing timers with function callbacks. Exposed data types are all abstract, and operations are provided to convert abstract types into microseconds. The timer callbacks must be setup at program startup with calls to LEON\_TimerRegisterHandler, which allocates memory for the timer. This should only be done at startup to ensure that runtime operation will always be deterministic.

## Serial Flash Interface

The SFI module of the Leon drivers has no knowledge of the specific flash chip or any SFI commands, so it only provides a mechanism to transport SFI commands to the flash device. This is broken down into 1) send instruction without data 2) send instruction with data 3) read status and 4) send a read instruction. The Leon RTL provides 2 mechanisms to access flash. One through register commands and one read only interface through the AHB. The read status will send the instruction and read back through the registers, while the read instruction will send the command, which will allow future access to use the AHB interface.

## Interrupts and Traps

The Leon driver provide C APIs to enable/disable/clear/set interrupt and trap handlers, as well as locking out interrupts for critical sections, or setting the interrupt priority high to provide a critical section for lower priority interrupts.

An assembly API is provided to create a trap table at build time, to reduce the need to set interrupt and trap handlers at startup time. If a trap table is not defined a default is used, which contains a jump to start for the reset trap and all other traps will be uninitialized.

## CPU

During an assert the Leon can provide useful information to the assert handler on the state of the CPU. The Leon provides this in a C API in the CPU module.

## Startup

Program startup is provided by the Leon driver. This is done by providing the assembly start function, as well as the sample linker files which place the start function at the beginning of flash. This code will call the function void \* imain(void) which is similar to C int main(int argc, char \* argv[]), except no arguments are provided, and the return code is another function to call with the same type as imain.

# External Module interface

## Use model

Users of the leon module are intended to use ibuild, and include the leon as the 2nd last dependency to be used as a library. Only ibase is a lower dependency than leon. The sample linker files will properly setup the start function, so the project only has to define imain, and it will be called. Optionally a trap table could be setup ahead of time so the interrupt handlers are defined at build time.

Each of the modules inside the leon driver, have their own initialization requirements and use models. Described below

## API/Data Structures

### UART

leon\_uart.h contains the APIs for this module. To use the UART in any mode the baudrate must first be configured.

A design decision must be made whether or not the UART is going to be used in a interrupt or polled mode. If it is interrupt driven then its interrupt handler must be registered and enabled, and a UART receive handler must be registered with the UART interrupt handler. Otherwise every time that it is wished to pause and flush the uart buffer LEON\_UartWaitForTx must be run, and LEON\_UartRx must be constantly called to check for any incoming data.

### Timers

leon\_timers.h contains the APIs for this module. LEON\_TimerInit must be called before using any timers at all. Timer interrupt callbacks must be registered at startup with LEON\_TimerRegisterHandler, and all other APIs are design to be called directly, with quick (or set delay) deterministic results.

### Serial Flash Interface

leon\_flash.h contains the APIs for this module. The flash module contains the basic transport API to send instructions to the flash, and wait for the result for that command. This module has no knowledge of SFI commands, and only ensures commands are transported to the external flash. To read from the flash, a read command does need to be sent, and then reads can happen directly over the AHB interface.

### Interrupts and Traps

leon\_traps.h contains the APIs for this module. This defines the interrupt types, and provides all of the API for acting on interrupts enable/disabling/clearing, as well as setting their handlers. There is also API for locking out interrupts for critical section, and only locking out lower priority interrupts.

### CPU

leon\_cpu.h contains the APIs for this module. A number of functions are provided to query the state of the internals of the CPU. This is intended for assert handlers, to assist in debugging.

### Startup

The startup code only exports the function start which is referenced in the sample linker scripts, default trap table for the reset vector, and needed nowhere else.

# Detailed Design

## UART

### Processing

The uart module is fairly straightforward. Before using the LEON\_UartSetBaudRate must be set, as well as externally setting the interrupt handler to LEON\_UartInterruptHandler and enabling it if desired. There are then 2 different parts to the uart module, the receiving and transmitting of characters over the uart.

The transmitter maintains an internal buffer to feed to the uart hardware that grows with every transmit software call, and shrinks on every interrupt, or while polling. The transmitter software uses the fact that the interrupts are edge-triggered so anytime the module is not active, the transmit function will write a byte directly to hardware to jump start the process. The transmitter has 2 functions to write data, LEON\_UartAtomicTx and LEON\_UartByteTx. The latter is for backwards compatibility, it only writes a single bytes and blocks to guarantee that that byte is written, while the former is for the newer ilog, and it will not block to guarantee determinism, which means that it can’t guarantee transmission.

The receiver will call its function pointer handler on every byte received, and maintains no state. If the module is running in polling mode, then the user must constantly call the polling function to check for incoming data.

### Local data structures

The uart contains an internal software fifo, which could be updated in the future to a hardware fifo, and the uart module also contains a function pointer for calling after every byte is received.

## Timers

### Processing

The timer module must be initialized with LEON\_TimerInit, which only initializes hardware, as no data structures with the module need initialization other than .bss zeroing.

The timer is then split into 3 different parts: blocking delays, measuring time, and timer callbacks.

LEON\_TimerWaitMicroSec is the only blocking delay function, and it merely constantly polls the hardware waiting for the specified delay time.

Time is read with LEON\_TimerRead, which returns an abstract type. The only way to use the abstract time type is on a call to LEON\_TimerCalcUsecDiff which knows how to convert to microseconds for 2 different time values, and find the difference. So one function reads the hardware, and the other function interprets the results.

The timer callbacks are much more involved. To ensure deterministic behavior all timer callbacks must call LEON\_TimerRegisterHandler at program initialization to ensure there is enough memory to handle the timers. Since the Leon component is such a low level component, there is no way to assert on an out of resource error, so instead the user must check the return value, where NULL is an error condition. Also on initialization the user must enable the timer interrupt, and register the LEON\_TimerInterruptHandler as the handler.

The timers can then be started and stopped at anytime with LEON\_TimerStart and LEON\_TimerStop. The behaviour of LEON\_TimerStart is defined as restarting the count if the timer is already running. Currently LEON\_TimerStart takes in 2 additional arguments for information on the timer. In future this could be moved to the initialization function to increase the runtime speed, and decrease the amount of data the caller has to supply.

The LEON\_TimerInterruptHandler is run on every timer interrupt. It currently searches through every timer to see which timer has expired. This could easily be optimized.

### Local data structures

The timer callbacks are stored as an array of structures. Each structure contains a function callback pointer, the count of time of which the timer is set, the timer tick count when the timer is active, and the timer’s state (disabled, periodic, one shot)

## Serial Flash Interface

### Processing

The SFI module is only a transport mechanism for flash commands. Each exported function just transports an SFI command to the flash chip, and optionally reads back the result

### Local data structures

LEON\_FlashDataLengthT is an abstract enum type, with names representative of its intended use, but the bit values set to match the hardware bitfields.

## Interrupts and Traps

### Processing

The interrupt and trap processing module contains a few different parts. For entering & leaving critical section a lock, unlock, and unlock priority function are available to lock out all interrupts, unlock the old state, or set the lock to a specified priority. This works by inline assembly taking a software trap, and a having a dedicated software trap handler to adjust the interrupt priority register that returns with the old value. The enabling, disabling and clearing of interrupts just modify the hardware registers directly related to the interrupt in question.

Trap and interrupt handlers are set by two different mechanisms. The build time handlers can be set with an assembly file that calls the macros LEON\_TRAPTABLESTART, LEON\_TRAPHANDLER, and LEON\_TRAPTABLEEND, which can populate the entire table at build time. If this isn’t used a default version exists. At runtime preferable start up time a different handler can be set with LEON\_InstallIrqHandler and LEON\_InstallTrapHandler which rewrite the assembly opcodes at the trap table entry point for that trap.

## CPU

### Processing

The CPU module is purely intended to assist in debugging after an assert. All of the functions exported are very small and contained, nothing is called from this module. A user should call LEON\_CPUDisableIRQ() to disable interrupts, and allow the debugging to occur uninterrupted. The entire rest of the API just return CPU internals.

### Local data structures

struct LEON\_CPURegs is used to represent a snapshot of a CPU register window.

## Startup

The startup code works in conjunction with example linker files, which set the start address as the leon startup code.

The startup code is very linear. The following is done

1. Initialize SPARC registers PSR, TBR & WIM
2. Enable/Disable cache
3. Disable Leon interrupts
4. Setup global registers G5, G6 & G7
5. memcpy .ftext into IRAM (possibly call with 0 bytes to copy)
6. memcpy .data into DRAM (possible call with 0 bytes to copy)
7. memset .bss to 0x0 (possible call with 0 bytes to set)
8. Enable SPARC traps
9. Set main function as imain()
10. call main function
11. set return value from main function as new main function
12. goto 10)

# Testing

## Test Plans

### UART

The UART is easily tested as a test harness. Each different exposed API is black box tested. Test harnesses exist in the test directory for polling and interrupt mode, for string and ilog transmitting, for receiving and echoing back. Since each test is a test harness running on the target hardware, and outputting back to the debug computer, passes and fails need to be determined by the human operator of the test.

### Timers

The timers module has a complete set of test harnesses testing every API. As with the UART module the test harnesses are all run on the device and must be verified by the human operator.

### Serial Flash Interface

The SFI module only transports flash commands. The only testing needed on this module is ensuring the upper layer of code that knows what flash commands are, and what type of flash commands to send to a device, can actually run. No testing is provided directly on this code. Upper layers need to do test based on a known flash device. Even a tiny application like the flash writer uses every one of the 4 exported functions.

### Interrupts and Traps

Interrupt handling is every interrupt based test harness. The most basic uart test harnesses test every almost every API this module exports. The build time interrupt table is tested implicitly by all projects that use it, and the interrupt priority locking is tested only by the Rex scheduler. This could be tested by a simulation based test harness in the future if needed.

### CPU

The CPU module is purely for assert behaviour. Any project which implements an assert handler should test their handler as they write it. The code here is small enough and low enough priority that it is only need for verification to test the project using this code.

### Startup

The start up code is always the same and is implicitly tested with every test harness.

## Integration Plan

All of the test harnesses run successfully before any attempt is made to push the Leon component into mainstream use, however the Leon component is designed to be easily dropped into the existing LG1 software, and in a branch in our revision control system Git, this is demonstrated long before this Leon component is ready to be pushed for general use. There is an attempt to update some of the old designs, so it isn’t an exact drop in, but the LG1 software can still be easily migrated. The flash writer can also be easily changed to use the Leon component as well. From this point on new software can use the Leon component from the start.