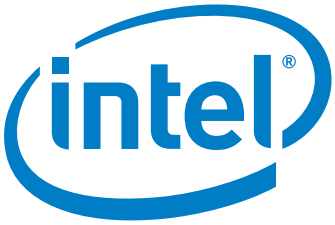


Intel® Quark SoC X1000 Board Support Package (BSP)

Programmer's Reference Manual (PRM)

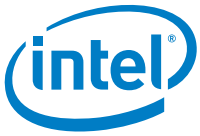
November 2013

Document Number: 521233, Revision: 0.8.0

Revision History

Revision	Date	Author	Description
0.8.0	November 2013	Bryan O'Donoghue	Added the following sections: • Section 7.3, "STMicroelectronics* LIS331DLH Accelerometer" on page 29 • Section 7.4, "Audio Control Driver" on page 30 • Section 7.5, "Bluetooth" on page 31 • Section 7.6, "WiFi" on page 32 Updated from Clanton codename to product name: Intel® Quark SoC X1000 (no changebars).
0.6.1	August 2013	Bryan O'Donoghue	Updated with feedback from first internal review. Modifications are indicated with changebars.
0.6	July 2013	Bryan O'Donoghue	Initial document detailing SOC, Bridge, Core and BSP driver and secure boot flow in Clanton A0 reference OS delivery.

Contents



Figures

Tables

1.0 Introduction

1.1 About this Manual

Intel® Quark SoC X1000 is a next generation secure, low-power Intel Architecture (IA) System on a Chip (SoC) for deeply embedded applications. The Intel® Quark SoC X1000 integrates the Intel® Quark Core plus all the required hardware components to run off-the-shelf operating systems and to leverage the vast x86 software ecosystem.

This document describes the architecture and usage of the Intel® Quark SoC X1000 Board Support Package (BSP) for Linux*.

1.2 Introduction

The Intel® Quark SoC X1000 BSP is a set of silicon enabling software that exposes silicon features to a run-time kernel and user-space in a convenient manner. Drivers that have been extended to enable Intel® Quark SoC X1000 are described in terms of standard driver interfaces. Drivers that have been created to expose a particular silicon feature are detailed in terms of their specific in-kernel and/or user-space API.

Intel® Quark SoC X1000 has standard x86 environment enumeration with legacy block and PCI enumeration mechanisms that are highly compatible with previous silicon configurations. Where possible, commercial off-the-shelf (COTS) drivers have been used and/or modified to achieve maximum compatability with minimum software code churn.

1.3 Related Documentation

Table 1 lists the documentation supporting this release.

Table 1. Product Documentation

Title	Number
Clanton Secure Boot Programmer's Reference Manual (PRM), revision 0.6	521232
Intel® Quark SoC X1000 Datasheet, revision 001	329676

1.4 Terminology

Table 2. Terminology (Sheet 1 of 2)

Term	Description
ADC	Analogue to Digital Converter
BSP	Board Support Package - a set of silicon enabling software which enables and enhances a run-time operating system kernel, such as Linux*.
DMA	Direct Memory Access

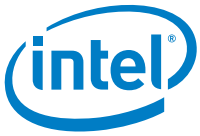


Table 2. Terminology (Sheet 2 of 2)

Term	Description
EHCI	Enhanced Host Controller Interface
eSRAM	embedded SRAM
GIP	Gateway Internet Protocol
GPIO	General Purpose Input/Output
i2c	I-squared-C - a type of two wire communications bus
IMR	Isolated Memory Region
LAN	Local Area Network
MMC	Multi Media Card
NC-GPIO	North Cluster GPIO
OHCI	Open Host Controller Interface
PCH	Platform Control Hub
RS232	A standard protocol to transmit serial data between computing systems.
SC-GPIO	South Cluster GPIO
SD	Secure Digital Flash
SoC	System on Chip
SPI	Serial Peripheral Interconnect
STMMAC	STMicroelectronics Media Access Controller
USB	Universal Serial Bus
VLAN	Virtual LAN

1.5 Conventions

The following conventions are used in this manual:

- `Courier` font - code examples, command line entries, API names, parameters, filenames, directory paths, and executables
- **Bold** text - graphical user interface entries and buttons

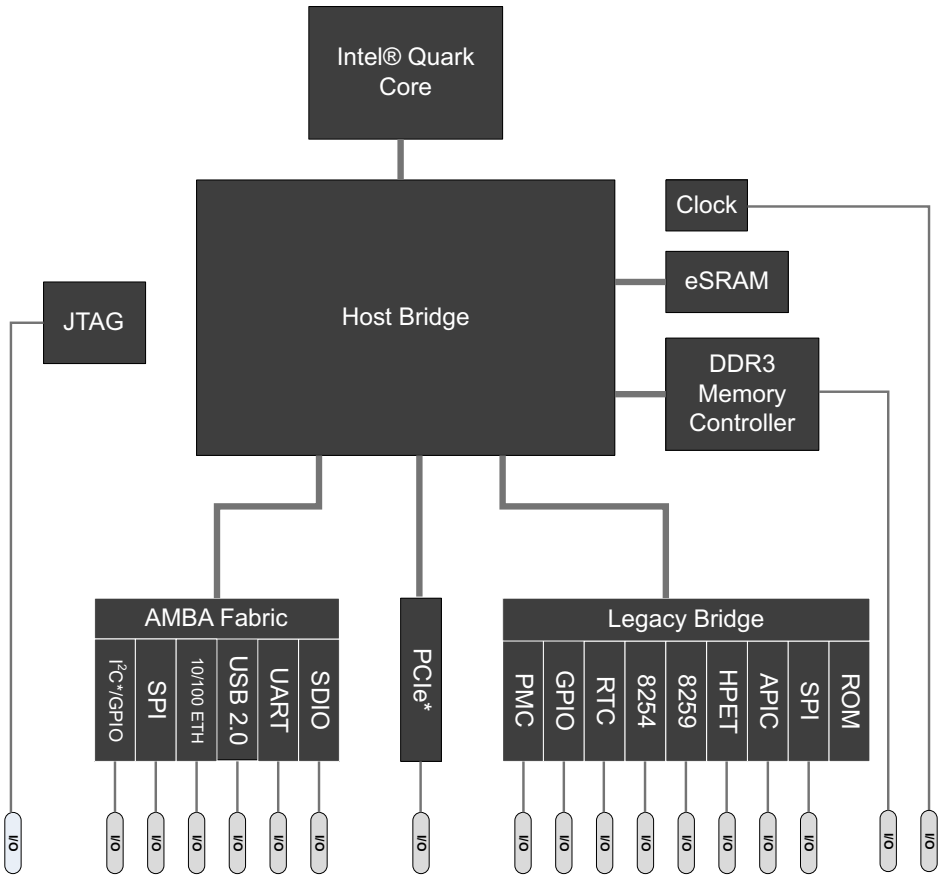
§ §

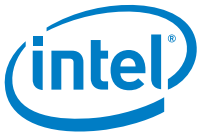
2.0 Platform Overview

2.1 Platform Synopsis

Intel® Quark SoC X1000 is a next generation, secure, low-power Intel Architecture System on Chip (SoC) for deeply embedded applications. As shown in Figure 1, Intel® Quark SoC X1000 is comprised of a Intel® Quark Core processor with a host bridge, PCIe expansion, a range of I/O interfaces, DDR3 controller, and an eSRAM block.

Figure 1. Intel® Quark SoC X1000 Block Diagram





2.2 SoC Features

The main features relevant to a BSP on Intel® Quark SoC X1000 are as follows:

- Intel® Quark Core
  - Intel® Pentium® compatible instruction set architecture (ISA)
  - Time stamp counter register (TSC)
  - Local APIC (LAPIC)
  - MSR compatability CPUID family = 0x5 revision = 0x09
- Host Bridge
  - 512k of fast access embedded SRAM (eSRAM)
  - 8 x memory protection regions, called *Isolated Memory Regions* (IMRs)
  - Thermal Sensor
- Legacy block
  - 8254 Programmable Interval Timer (PIT)
  - 2 cascaded 8259 Programmable Interrupt Controllers (PIC)
  - High Precision Event Timer (HPET)
  - IO-APIC
  - Real Time Clock (RTC)
  - GPIO x 8 - 6 in suspend well - driving NMI, SCI, or SMI
  - Legacy SPI and Boot ROM
- Intel® Quark SoC X1000
  - OCHI USB Host controller
  - EHCI USB Host controller
  - USB Device controller
  - 2 x 16550 RS232 with DMA enhancements
  - 2 x SPI Master interface
  - I2C Master interface
  - 8 x SC-GPIO
  - 2 x 100 Mbit Ethernet
  - eMMC/MMC controller interface

§ §

3.0 Software Overview

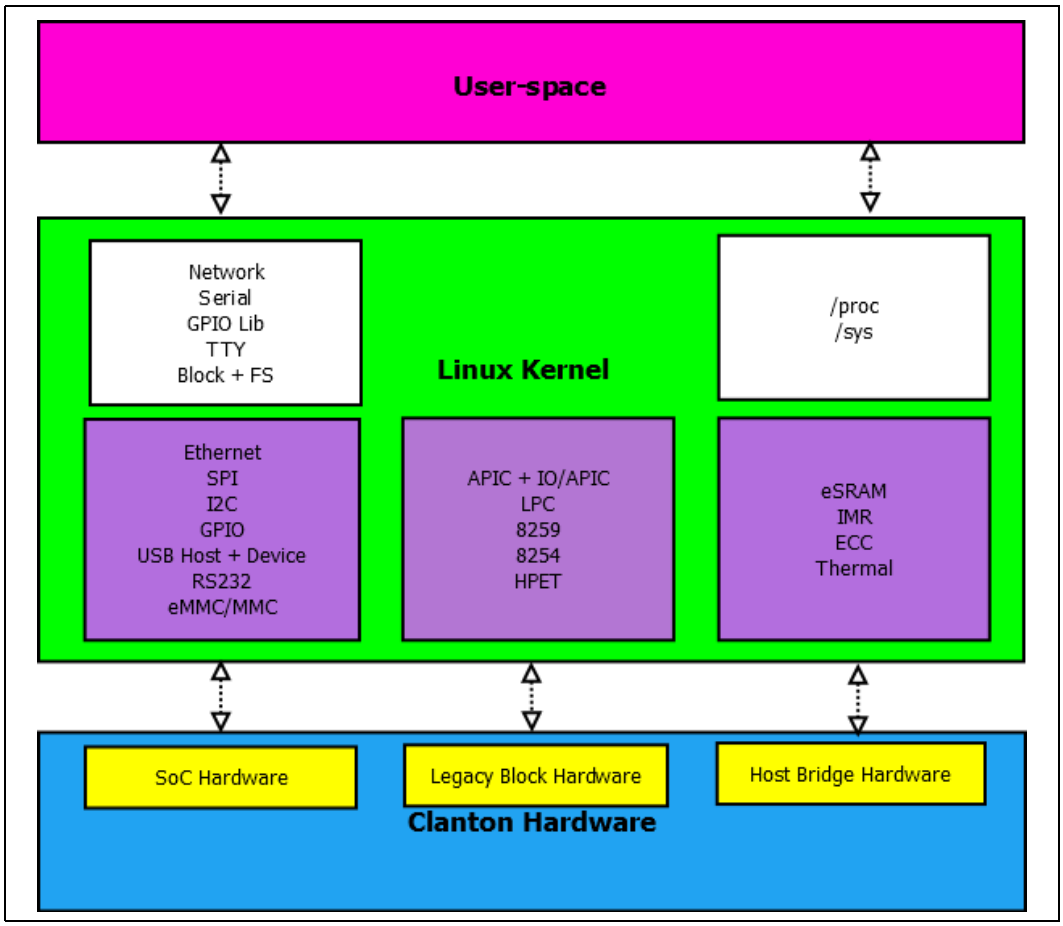
3.1 High-Level Software Architecture Overview

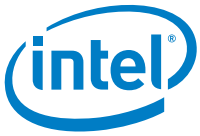
The Intel® Quark SoC X1000 uses many off-the-shelf software components to enable product features. This aim is pervasive throughout the system in terms of Intel® Quark Core, Host Bridge, and SoC components.

Intel® Quark SoC X1000 has two key categories of software deliverables in the BSP:

- Extensions to existing Linux* device drivers to enable the Intel® Quark SoC X1000
- Creation of entirely new drivers for Host Bridge-related functions

Figure 2. Software Architecture Overview





3.2 Operating System Support

3.2.1 Standard OS Drivers

The software delivery supports Linux. Many of the I/O drivers, including USB, Ethernet, RS232, I2C, and SPI, are derived from existing upstream kernel components. (The SC-GPIO driver was created for Intel® Quark SoC X1000.) Driver modifications maintain compatibility with existing software while enabling Intel® Quark SoC X1000 specific features.

3.2.2 Host Bridge OS Drivers

Host Bridge silicon enabling software is specific to the Intel® Quark SoC X1000 and as such has no formal operating system interface that exactly matches the conceptual paradigms. For this reason, Intel® Quark SoC X1000 specific APIs and user-space interfaces via `sysfs` and `proc` have been developed for the IMR, ECC scrub control, and eSRAM interface.

Details on the interfaces for IMR, ECC, and eSRAM configuration are provided later in this document.

The capability exists to add a kernel subsystem to control dynamic allocation of IMRs based on DMA mappings, however, this is not currently planned.

3.2.3 Bootloader Host Bridge Drivers

In order to facilitate secure boot, the reference bootloader `grub` has been modified to support setup and teardown of IMRs as appropriate to transition from UEFI to run-time OS. Section 8.0, “Secure Boot Implementation” on page 34 describes this flow.

3.3 User-Space Software Dependencies

To facilitate exposure of silicon features, the user-space component of the runtime reference OS requires the following utilities:

- `ethtool` - customized version of ethtool updated to include registers exported by the Intel® Quark SoC X1000
- `ptpd` - Precision Time Protocol Daemon - required to implement 1588 PTP sync

§ §

4.0 Intel® Quark SoC X1000 Drivers

System on a Chip in the context of Intel® Quark SoC X1000 refers to peripheral hardware south of the host bridge interface. SoC software drivers bind the hardware interfaces into standard Linux sub-systems. Linux kernel baseline of 3.8.7 (or higher) is required to ensure proper integration and compatibility of upstream reused kernel drivers.

4.1 Overview

Table 3 lists the hardware interfaces implemented on Intel® Quark SoC X1000 and identifies whether the associated driver is one of the following:

- standard (unmodified) off-the-shelf driver
- modified version of off-the-shelf driver, enhanced to enable Intel® Quark SoC X1000 specific features
- created to be Intel® Quark SoC X1000 specific

Table 3. Intel® Quark SoC X1000 Hardware Interfaces and Drivers

Hardware Interface	Standard Linux Driver	Modified Linux Driver	Intel® Quark SoC X1000 Specific Driver
USB OHCI Controller Interface	X		
USB 2.0 EHCI Controller Interface	X		
USB Device Interface		X†	
SD/MMC Controller Interface	X		
RS232 + DMA Interface		X†	
SPI Master Interface		X	
I2C Master Interface	X		
SC-GPIO Interface			X
Ethernet Interface		X	
† PCI vendor/device identifiers added for Intel® Quark SoC X1000.			

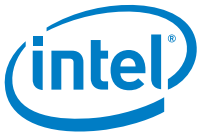
4.2 USB OHCI Controller Interface Driver

The standard Linux OHCI driver is 100% compatible with Intel® Quark SoC X1000. This driver provides full USB host control and arbitration of the USB in EHCI mode.

To load this driver in Linux as root, type:

```
modprobe ohci_hcd
```

Once loaded, the OHCI driver provides access to USB 1.1 devices through either of the USB host ports, thus enabling host controller interface with full speed and low speed USB devices.



A given USB port can be OHCI mode or EHCI mode, but not both.

4.3 USB 2.0 EHCI Controller Interface Driver

The standard Linux EHCI driver is 100% compatible with Intel® Quark SoC X1000. This driver has a prerequisite for the OHCI to be loaded before the EHCI driver is loaded. Once loaded, the EHCI driver provides full host control and arbitration of the USB in EHCI mode.
To load this driver in Linux as root, type:

```
modprobe ohci_hdc
modprobe ehci_hcd
```

Once loaded, the EHCI driver provides access to High speed USB devices through either of the Intel® Quark SoC X1000 host controller ports.

A given USB port can be OHCI mode or EHCI mode, but not both.

4.4 USB Device Interface Driver

The standard PCH UDC driver (with the addition of Intel® Quark SoC X1000 PCI vendor/device identifiers) is 100% compatible with Intel® Quark SoC X1000.

Using the reference driver released in the BSP, type:

```
modprobe pch_udc
```

This will load the hardware driver.

To have Intel® Quark SoC X1000 appear as a USB key, and assuming a suitable file exists at
`/media/mmc1/floppy.img`, type:

```
modprobe g_mass_storage file=/media/mmc1/floppy.img
```

Intel® Quark SoC X1000 should then present to the USB host machine as a standard USB key.

4.5 SD/MMC Controller Interface Driver

The standard Linux MMC/SD driver is 100% compatible with Intel® Quark SoC X1000. Once loaded, an MMC or SD storage device will appear as a standard Linux block interface, upon which a file system can be formatted and mounted.

This example loads the SDHCIPCI driver and MMC block device driver:

```
modprobe sdhci-pci
modprobe mmc_block
```

Once loaded, assuming the MMC card is partitioned and formatted, device entries will appear in `/dev` representing the partitions found on the MMC device.

4.6 RS232 + DMA Interface Driver

The standard upstream 16550 PCI UART will work with Intel® Quark SoC X1000, with the addition of the relevant PCI vendor/device strings. The Intel® Quark SoC X1000 RS232 interface is 100% compatible with the standard 16550 register interface. The FIFO depth is 32 bytes and hardware flow control is included. Intel® Quark SoC X1000 has two RS232 ports which operate in PCI mode only.

Note: There is no support supplied for legacy I/O port access at addresses 0x3F8, 0x2F8, 0x3E8 or 0x2E8.

Inside the PCI configuration space of each UART, a second PCI BAR exists containing DMA registers that can be used with each of the UARTs to provide high data throughput.

A custom driver called `intel_cln_uart` is provided to take advantage of these DMA registers. This driver can be loaded in MSI mode or non-MSI mode, where MSI mode is the default, see sample commands below:

```
modprobe intel_cln_uart
modprobe intel_cln_uart enable_msi=0
```

This driver registers:

```
/dev/ttyCLN0
/dev/ttyCLN1
```

These two UART interfaces provide higher data throughput with lower IRQ count when used in place of the standard 8250_pci driver.

4.7 SPI Interface Driver

The Intel® Quark SoC X1000 SPI interface exports a standard SPI interface from kernel-space to user-space. Two SPI master interfaces are available on Intel® Quark SoC X1000. To increase the number of devices that Intel® Quark SoC X1000 can communicate with simultaneously, GPIOs are used to achieve *multiplexing* (also called *muxing*) of the SPI master interface.

This muxing approach allows Intel® Quark SoC X1000 to communicate with up to four SPI slave interfaces, with a maximum of two slave devices at any one time as shown in Figure 3.

To load Intel® Quark SoC X1000 SPI driver, type:

```
modprobe spi-pxa2xx.ko
modprobe spi-pxa2xx-pci
modprobe spidev.ko
```

Note: For non-MSI, type: `modprobe spi-pxa2xx.ko enable_msi=0`

GPIO pin selection is achieved by providing board-specific data in the file:

```
arch/x86/platform/cln/boardname.c
```

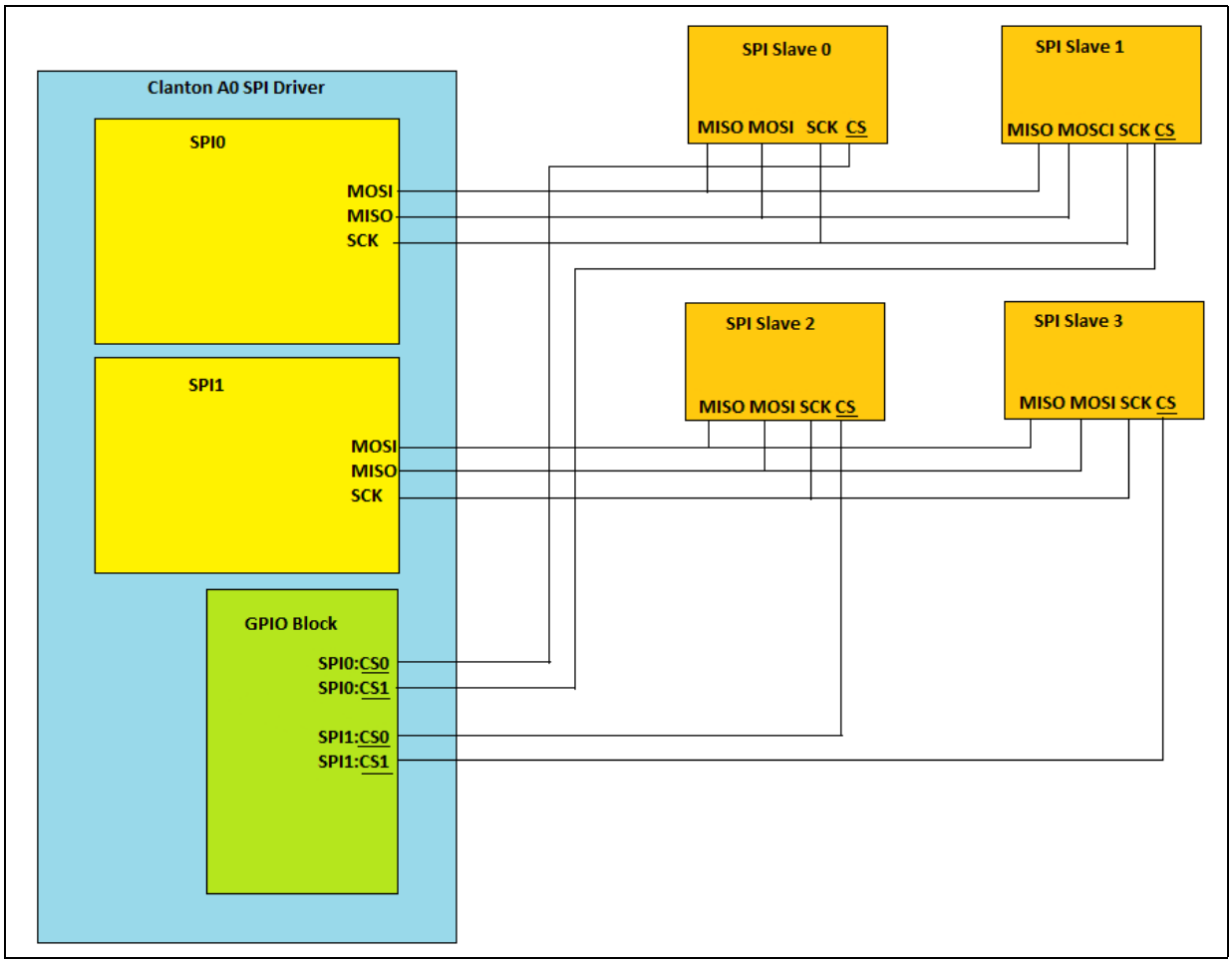
Once loaded, the master SPI driver will populate entries in `/dev` as follows:

```
/dev/spidev0.0
/dev/spidev0.1
/dev/spidev1.0
/dev/spidev1.1
```

The format is `/dev/spidevX.Y` where:

- `X` indicates the master interface
- `Y` indicates the slave interface

Figure 3. Multiplexing using Intel® Quark SoC X1000 SPI Driver



4.8 I2C Interface Driver

The I2C and SC-GPIO components are contained within the same PCI function and share resources as a consequence. The I2C register interface is 100% compatible with the upstream `i2c-designware-core` driver.

This register interface is incorporated in the `intel_cln_gip` driver, which provides a standard I2C interface when loaded. The GIP interface can be loaded in either MSI or non-MSI mode using the commands:

```
modprobe intel_cln_gip
modprobe intel_cln_gip enable_msi=0
```

In either case, loading this driver and using the command `modprobe i2c-dev` populates:

```
/dev/i2c-0
```

Once populated, it is possible to communicate with downstream I2C devices using the standard Linux API to interact with the I2C bus.

To load the I2C driver in isolation (that is, without the GPIO enabling logic contained in the GIP block), type.

```
modprobe intel_cln_gip gpio=0
modprobe intel_cln_gip gpio=0 enable_msi=0
```

4.9 SC-GPIO Interface Driver

The name *SC-GPIO* is used to differentiate the South Cluster GPIO driver from the legacy block or North Cluster GPIO components. The SC-GPIO and I2C components are contained within the same PCI function and share resources as a consequence. The SC-GPIO interface is a new register interface and is enabled by the GPIO section of the `intel_cln_gip` device driver module.

To load the GPIO driver in isolation (that is, without the I2C enabling logic contained in the GIP block) type:

```
modprobe intel_cln_gip i2c=0
modprobe intel_cln_gip i2c=0 enable_msi=0
```

Note: Enabling MSIs is recommended for improved performance.

4.10 Ethernet Interface Driver (STMMAC)

The STMMAC driver upstream in the Linux kernel is nearly entirely compatible with Intel® Quark SoC X1000, with some minor updates to the DMA component of the STMMAC driver.

In addition to the necessary DMA enumerating descriptors in STMMAC, additional Intel® Quark SoC X1000 specific silicon-enabling enhancements have been made to the standard STMMAC. The enhancements include:

- VLAN
  - — Hardware filtering has been added
  - — Maximum number of hardware filtered VLAN tags is 16
  - — Tag ID range 0 - 15
- 1588 version 2 time-stamping has been added

The following commands demonstrate how to load the STMMAC in either MSI or non-MSI mode.

```
modprobe stmmac
modprobe stmmac enable_msi=0
```

Note: MSI mode is enabled by default.

4.10.1 VLAN

The standard Linux commands `ip` or `vconfig` can be used to add or remove hardware accelerated VLAN tag filtering entries in STMMAC.

The following commands demonstrate how to add VLAN # 5:

```
vconfig add eth0 5
ifconfig eth0.5 xxx.yyy.zzz.qqq
```

Once setup is complete, VLAN frames with tag ID 5 will be processed by Intel® Quark SoC X1000 while other ethernet frames with different tags are not being processed by hardware and not raising interrupts to the core.

To remove a hardware filtered VLAN interface, enter the command:

```
vconfig rem eth0.5
```

4.10.2 IEEE 1588

The Intel® Quark SoC X1000 STMMAC driver registers a Linux Precision Time Protocol (PTP) device with the appropriate Linux sub-system.

The driver provides bindings for the following PTP events:

- HWTSTAMP_FILTER_PTP_V1_L4_EVENT
- HWTSTAMP_FILTER_PTP_V1_L4_DELAY_REQ
- HWTSTAMP_FILTER_PTP_V1_L4_SYNC
- HWTSTAMP_FILTER_PTP_V2_L4_EVENT
- HWTSTAMP_FILTER_PTP_V2_L4_DELAY_REQ
- HWTSTAMP_FILTER_PTP_V2_L4_SYNC
- HWTSTAMP_FILTER_PTP_V2_L2_EVENT
- HWTSTAMP_FILTER_PTP_V2_L2_DELAY_REQ
- HWTSTAMP_FILTER_PTP_V2_L2_SYNC
- HWTSTAMP_FILTER_PTP_V2_EVENT
- HWTSTAMP_FILTER-PTP_V2_DELAY_REQ
- HWTSTAMP_FILTER_PTP_V2_SYNC
- HWTSTAMP_FILTER_ALL
- HWTSTAMP_FILTER_NONE

With this set of PTP events enabled in the driver and bound into the Linux PTP sub-system, interface functionality consistent with the kernel PTP documentation is enabled.

For more information, refer to: linux-kernel/Documentation/ptp/ptp.txt
at or around Linux kernel version 3.8.7 for the appropriate reference.

5.0 Intel® Quark SoC X1000 Host Bridge Drivers

Host Bridge Drivers in the context of Intel® Quark SoC X1000 refer to drivers for silicon functionality that are part of the Host Bridge interface on Intel® Quark SoC X1000. This functionality is exposed via a *side-band* driver that arbitrates access to the various components using the Host Bridge interface.

The side-band driver provides access to the following blocks of functionality:

- eSRAM
- DRAM ECC Memory Scrub Control
- Isolated Memory Regions
- Thermal

5.1 eSRAM Configuration Driver

Intel® Quark SoC X1000 contains a set of embedded SRAM (eSRAM). There is 512 kilobytes of eSRAM sub-divided into 128 pages of four kilobytes each. eSRAM can be configured in “block” mode or in a per-page manner and eSRAM can exist in an ‘overlay’ or as a contiguous chunk of memory in the address space.

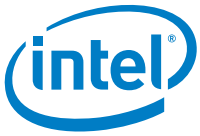
eSRAM is a fast access low-latency memory with similar performance to cache RAM, in terms of CPU wait-states and access times.

For Linux BSP enabling purposes, eSRAM has been configured in a per-page overlay mode. This approach allows overlay of specific regions of memory. For example, the interrupt descriptor table or arbitrary interrupt service routines (ISRs) can be locked into eSRAM.

Any kernel symbol visible in `/proc/kallsyms` can be mapped into eSRAM. The minimum granularity for any map operation is 4 kilobytes, hence any other data within the same 4 kilobyte address range will also be mapped.

A `sysfs` interface has been provided to configure and eSRAM mappings.

- `/sys/class/esram/map`
  - Allows mapping of a given kernel symbol
  - Allows unmapping of a given kernel symbol
  - Allows viewing of all current eSRAM mappings
- `/sys/class/esram/stats`
  - Gives a status overview of current eSRAM state
  - Number of free pages
  - Next eSRAM ECC scrub
  - Other miscellaneous data



5.1.1 Example showing eSRAM stat usage

```
root@clanton:~# cat /sys/class/esram/stats
esram-pgpool : 0x19f8fc00
esram-pgpool.free               : 127
esram-pgpool.flushing           : 127
esram-ctrl                      : 0x047f3f91
esram-ctrl.ecc                  : enabled
esram-ctrl.ecc-theshold         : 63
esram-ctrl.pages                : 128
esram-ctrl.dram-flush-priorityi : 2
esram-block                     : 0x00000000
free page                       : 127
used page                       : 1
refresh                         : 675000ms
page enable retries             : 0
page disable retries    : 0
ecc next page                   : 126
```

5.1.2 Example of mapping `printk` into eSRAM from user-space

```
root@clanton:~# echo printk on > /sys/class/esram/map
root@clanton:~# cat /sys/class/esram/map
printk+0x0/0x3a
        Page virt 0xc12ab000 phys 0x012ab000
        Refcount 1
We can easily verify the mapping is correct by viewing /proc/kallsyms
root@clanton:~# cat /proc/kallsyms | grep printk
c1004ea0 T printk_address
c101cd00 T early_printk
c12ab110 T printk
```

5.1.3 Kernel API Reference

An API to map known kernel symbols and arbitrary kernel address ranges is available.

Note: Unmapping is neither supported nor advised due to potential coherency issues when flushing eSRAM back to DRAM. Unmap code is provided for reference purposes only.

5.1.3.1 intel_cln_esram_map_range

Map 4k increments at given address to eSRAM. Maps any arbitrary virtual address from vaddr to vaddr + size bytes. This mapping will then be named `mapname`.

```
int intel_cln_esram_map_range(void * vaddr, u32 size, char * mapname);
```

- vaddr: Virtual address to start mapping (must be 4k aligned)
- size: Size to map from
- mapname: Mapping name - must be a valid kernel symbol name
- return 0 success < 0 failure

5.1.3.2 intel_cln_esram_unmap_range

Unmaps an address range from a given base address vaddr to vaddr+size.

```
int intel_cln_esram_unmap_range(void * vaddr, u32 size, char * mapname);
```

- vaddr: Virtual address to start mapping (must be 4k aligned)
- size: Size to map from
- mapname: Mapping name - must be a valid kernel symbol name
- return 0 success < 0 failure

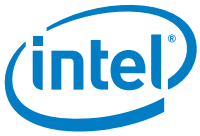
5.1.3.3 intel_cln_esram_map_symbol

Maps a series of 4k chunks starting at vaddr&0xFFFFF000. vaddr shall be a kernel text section symbol (kernel or loaded module)

We get the size of the symbol from kallsyms. We guarantee to map the entire size of the symbol - plus whatever padding is necessary to get alignment to eSRAM_PAGE_SIZE. Other code/data inside the mapped pages will get a performance boost for free.

```
int intel_cln_esram_map_symbol(void * vaddr);
```

- vaddr: Virtual address of the symbol
- return 0 success < 0 failure



5.1.3.4 intel_cln_esram_unmap_symbol

Logical corollary to `intel_cln_esram_map_symbol`. Removes mapping of pages starting at the address of the symbol vaddr. Reference counting for individual pages means that an eSRAM page will only become unmapped once all mapping references have been removed. If `printk()` and `malloc()` both live in the same four kilobyte physical address range and both have been mapped into eSRAM, then only when both mapping references have been removed will the physical mapping reference also be removed.

```
int intel_cln_esram_unmap_symbol(void * vaddr);
```

- vaddr: Virtual address of the symbol
- return 0 success < 0 failure

5.2 DRAM ECC Memory Scrub Control Driver

The Intel® Quark SoC X1000 has a highly configurable ECC refresh logic, which allows the end-user to tweak the size scope and frequency of DRAM ECC refresh.

By default, the ECC scrub logic:

- Starts at address : 0x00000000
- Ends at address : 0x800000000
- Has a scrub block size : 512 bytes
- Has a scrub interval : 255 seconds
- ECC scrub status : on

Each value can be over-ridden on the kernel command line:

- `intel_cln_ecc_scrub.ecc_scrub_config_override`
  - — Allows the user force scrubbing to the on/off state depending on the boolean value of ecc_scub_config_override
- `intel_cln_ecc_scrub.ecc_scrub_start_override`
  - — Allows setting the start address of scrubbing to an arbitrary address
- `intel_cln_ecc_scrub.ecc_scrub_end_override`
  - — Allows for setting the maximum end address of ECC scrubbing
- `intel_cln_ecc_scrub.ecc_scrub_next_override`
  - — Allows for setting the next address or begging address of ECC override

Each override variable has a similar entry in sysfs which can be directly influenced by writing to the relevant sysfs entry directly:

- `/sys/class/ecc_scrub/interval`
- `/sys/class/ecc_scrub/block_size`
- `/sys/class/ecc_scrub/control`
- `/sys/class/ecc_scrub/status`

5.2.1 /sys/class/ecc_scrub/interval

Writing a value into this entry will cause the interval specified to be used in ECC scrubbing.

Valid values are in the range 1 - 255

To configure:

```
echo 20 > /sys/class/ecc_scrub/interval
```

To verify current value:

```
cat /sys/class/ecc_scrub/interval
```

5.2.2 /sys/class/ecc_scrub/block_size

Writing a value into this entry will cause the interval specified to be used in ECC scrubbing.

Valid values are in the range 64 - 512.

To configure:

```
echo 128 > /sys/class/ecc_scrub/block_size
```

To verify current value:

```
cat /sys/class/ecc_scrub/block_size
```

5.2.3 /sys/class/ecc_scrub/control

The ECC control entry allows for writing of an integer value to switch ECC scrubbing on or off.

To switch ECC scrubbing on:

```
root@clanton# echo 1 > /sys/class/ecc_scrub/control
```

To switch ECC scrubbing off:

```
root@clanton# echo 0 > /sys/class/ecc_scrub/control
```

To view the current state of ECC scrub either 0 for off or 1 for on:

```
root@clanton# cat /sys/class/ecc_scrub/control
```

5.2.4 /sys/class/ecc_scrub/status

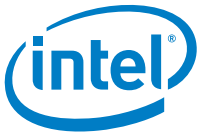
This entry gives a synopsis of current ECC scrubbing status.

```
root@clanton# cat /sys/class/ecc_scrub/status
ecc scrub mem start: 0x00000000
ecc scrub mem end: 0x80000000
ecc scrub next read: 0x00025800
ecc scrub interval: 255
ecc scrub block_size: 512
ecc scrub status: on
```

5.3 Isolated Memory Region Driver

Isolated Memory Region (IMR) allocation and assignments are detailed in the *Clanton Secure Boot Programmer's Reference Manual (PRM)*. In Linux a run-time interface provides an convienient method to view IMR allocations.

This interface shows the IMR allocations provided as part of the secure boot reference code on the Intel® Quark SoC X1000.



5.3.1 IMR run-time kernel protection

```
root@clanton:~# cat /proc/driver/imr/status
imr - id : 0
info     : System Reserved Region
occupied : yes
locked   : yes
size     : 4344 kb
hi addr (phy): 0x0143dc00
lo addr (phy): 0x01000000
hi addr (vir): 0xc143dc00
lo addr (vir): 0xc1000000
read mask  : 0x80000001
write mask : 0xc0000001
```

5.4 Thermal Driver

Linux provides a standard thermal driver interface. Intel® Quark SoC X1000 hooks its particular thermal silicon into this Linux sub-system. Since Intel® Quark SoC X1000 does not require external cooling, the thermal driver is minimalistic in design, with no associated thermal cooling device attached to the one and only thermal zone.

Intel® Quark SoC X1000 hardware can be configured to automatically shutdown on critical temperature detection. Two types of temperature ranges are supported by the thermal driver.

Linux provides an entire sub-system dedicated to triggering events based on hot and critical events. The task of the thermal driver is to provide the minimum level of silicon support to drive these events.

Selecting between normal temperature range and extended temperature range is done via a kernel command line parameter to the thermal driver:

```
intel_cln_thermal.extended_temp=0
intel_cln_thermal.extended_temp=1
```

By default, the thermal driver is in normal temperature mode.

5.4.1 Normal Temperature Range

Hot trip point in the normal temperature range is 70 degrees Celsius. The thermal driver is incrementally polling the thermal sensor and when the 70 degree theshold is exceeded a hot trip event is propagated into the thermal sub-system.

Critical trip point in the normal temperature range is 85 degrees Celsius. The Linux thermal sub-system will trigger a graceful system shutdown if 85 degrees celsius is reached.

As a precautionary measure, Intel® Quark SoC X1000 silicon is configured to drive a shutdown signal at 87 degrees Celsius. Assumption is that software polling should catch an over-temperature situation when temperature meets or exceeds 85 degrees celsius. A two degree over-limit from the maximum specified critical temperature will then lead embedded hardware to take preventative action and drive a shutdown signal directly.

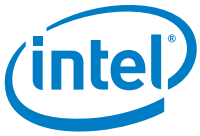
- Hot trip point: 70 degrees Celsius
- Critical trip point: 85 degrees Celsius
- Hardware failover critical temperature: 87 degrees Celsius

5.4.2 Extended Temperature Range

Driver behavior is precisely the same in the extended temperature range as it is in the normal temperature range, with the exception that the upper limits for hot, critical, and hardware failover critical temperatures are higher.

- Hot trip point: 85 degrees Celsius
- Critical trip point: 95 degrees Celsius
- Hardware failover critical temperature: 97 degrees Celsius

§ §



6.0 Legacy Block Drivers

The LPC address space contained within Intel® Quark SoC X1000 legacy block has two important components that have been brought out and enabled in the Linux run-time:

- NC-GPIO (North Cluster GPIO)
- Watchdog Timer

In order to enable both of these pieces of silicon functionality, a small modification is necessary to LPC enabling software in Linux, adding appropriate PCI vendor/device.

6.1 NC-GPIO

Intel® Quark SoC X1000 contains eight GPIOs within the legacy bridge. These North Cluster (NC) GPIOs provide the ability to drive GPE events and hence to remove a Intel® Quark SoC X1000 device in a low-power state.

The name *NC-GPIO* is used to differentiate the *North Cluster* GPIO driver from the SoC or *South Cluster* GPIO components.

There are:

- 6 GPIO pins in the resume power well
- 2 GPIO pins in the core well

The six GPIOs in the resume well can be used to drive a General Purpose Event (GPE) through the ACPI sub-system that will subsequently take Intel® Quark SoC X1000 out of a low-power state.

The eight SC-GPIO are indexed in the range [0,7] and can be accessed from user-space through sysfs interface.

The commands below demonstrate how to drive a signal to the first NC-GPIO:

```
root@clanton# echo 0 > /sys/class/gpio/export # Reserve first SC GPIO
root@clanton# echo "out" > /sys/class/gpio/gpio0/direction # Set as output
root@clanton# echo "1" > /sys/class/gpio/gpio0/value # Drive logical one
```

6.2 Watchdog Timer

Intel® Quark SoC X1000 has watchdog functionality embedded in the LPC bus address space.

Upstream driver code is 100% compatible with this WDT silicon.

Upstream driver is `drivers/watchdog/ie6xx_wdt.c`

Using the watchdog API, it is possible for user-space to periodically notify the kernel watchdog infrastructure that user-space is still alive.

```
open /dev/watchdog
Periodically execute the following:
while (1) {
                ioctl(fd, WDIOC_KEEPALIVE, 0);
                sleep(10);
}
```

Using this simple incremental notification pass, it is possible to preempt reset of the system by the watchdog silicon. Failure to execute the notification loop within the defined time-out will result in system reset.

§ §

7.0 Board Support Drivers

Intel® Quark SoC X1000 supports the following boards:

- Intel® Quark SoC X1000 Customer Reference Board, "Kips Bay"
- Intel® Quark SoC X1000 Indu/Energy Reference Design, "Cross Hill"
- Intel® Quark SoC X1000 Transportation Reference Design, "Clanton Hill"
- Intel® Galileo board

A number of drivers are included with the reference package that enable a range of board specific functionality.

For Cross Hill and Kips Bay:

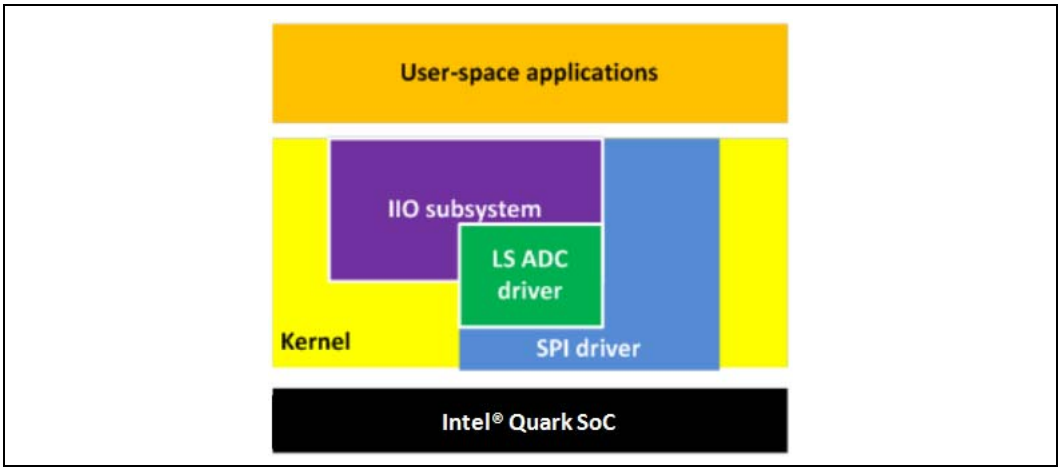
- Analog Devices* AD7298 is included with board FFRD designs
- BSP enabling driver is included with the reference OS deliverable

Cross Hill also includes an additional optional daughterboard interface:
Maxim Integrated* 78M6610+LMU energy ADC attaches to Cross Hill via the Blackburn Peak daughterboard.

7.1 AD7298

The Analog Devices* AD7298 is a 12-bit, low power, 8-channel, successive approximation ADC with an internal temperature sensor. The LS-ADC does not provide a user-space interface directly. This is provided by the IIO subsystem in the Linux kernel. The ADC registers with the IIO subsystem as an IIO ADC device driver. As such, it makes calls to functions on the IIO kernel API and provides callbacks which can be used by the IIO subsystem to invoke driver operations.

Figure 4. ADC Location in Software Stack



To load the drivers for the AD7298, perform the following sequence:

- Enable GPIO driver:
  ```
  modprobe intel_cln_gip
  modprobe gpio_sch
  ```
- Enable IIO support:
  ```
  modprobe industrialio
  ```
- Enable SPI driver:
  ```
  modprobe spi-pxa2xx-pci
  modprobe spi-pxa2xx
  ```
- Enable AD7298 driver:
  ```
  modprobe ad7298
  ```

After the driver loading sequence is complete, the AD7298 driver enables the following data points via the Industrial I/O (IIO) kernel API directly read from the ADC chip.

- Provide the RAW voltage at the input in the range 0 - 4095 representing the voltage range 0 to +5 Volts
  ```
  /sys/bus/iio/devices/iio:device0/in_voltage[0-7]_raw
  /sys/bus/iio/devices/iio:device0/in_voltage0_raw
  /sys/bus/iio/devices/iio:device0/in_voltage1_raw
  etc
  ```
- Scaling value to apply to the raw voltage input
  ```
  /sys/bus/iio/devices/iio:device0/in_voltage_scale
  ```
- Temperature offset
  ```
  /sys/bus/iio/devices/iio:device0/in_temp0_offset
  ```
- Raw instantaneous temperature of the ADC die
  ```
  /sys/bus/iio/devices/iio:device0/in_temp0_raw
  ```
- Temperature scaling factor
  ```
  /sys/bus/iio/devices/iio:device0/in_temp0_scale
  ```

Other data points are provided by the Linux IIO API but are out of scope for this document.

Using the above values, it is possible to calculate the real instantaneous voltage at a given voltage input using the following formula:

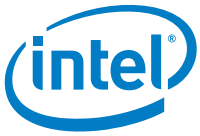
(Raw value * scale value) / 1000 = actual input voltage

7.2 Maxim Integrated* 78M6610+LMU

The Maxim Integrated* 78M6610+LMU is an energy measurement processor (EMP) for load monitoring on single or split-phase AC loads. It supports various interface configuration protocols through I/O pins. With 3 wire serial input/output interfaces provided by Intel® Quark SoC X1000, the 78M6610+LMU can be connected directly as an SPI slave device.

To load the drivers for the EMP, perform the following sequence:

- Enable GPIO driver:
  ```
  modprobe intel_cln_gip
  modprobe gpio_sch
  ```
- Enable SPI driver:
  ```
  modprobe spi-pxa2xx-pci
  modprobe spi-pxa2xx
  ```



- Enable EMP driver:
  ```
  modprobe max78m6610_lmu
  ```
- Enable IIO support:
  ```
  modprobe industrialio-triggered-buffer
  modprobe kfifo_buf
  modprobe iio-trig-sysfs
  ```

After the loading sequence is complete, the driver enables the following data points via the industrial I/O kernel API directly read from the 78M6610+LMU chip.

- Instantaneous Current Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_current[0/1]_inst_en`
- Instantaneous Voltage Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_voltage[0/1]_inst_en`
- RMS Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_current[0/1]_rms_en`
- Apparent Power Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_power[0/1]_apparent_en`
- Average Active Power Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_power[0/1]_avg_act_en`
- Average Reactive Power Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_power[0/1]_avg_react_en`
- Power Factor Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_power[0/1]_factor_en`
- Instantaneous Active Power Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_power[0/1]_inst_act_en`
- Instantaneous Reactive Power Phase [0/1]
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_power[0/1]_inst_react_en`
- Timestamp Counter
  `/sys/bus/iio/devices/iio:device1/scan_elements/in_timestamp_en`

7.3 STMicroelectronics* LIS331DLH Accelerometer

The STMicroelectronics* LIS331DLH is a low-power three axes linear accelerometer which is presented as an I^2C device at bus address 0x18 on the Clanton Hill CRB. The accelerometer provides data on the x, y and z axes, for the raw and scaled values detailed below.

The following drivers must be loaded to enable the LIS331DLH driver:

```
modprobe intel_cln_gip
modprobe gpio_sch
modprobe industrialio
modprobe lis331dlh
```

After the driver loading sequence is complete, the LIS331DLH driver enables data points via the industrial I/O kernel API directly read from the chip.

- Instantaneous raw acceleration value for axes x, y and z
  `cat /sys/bus/iio/devices/iio:device0/in_accel_<axis>_raw`
- Scaling value for each of the axes x, y and z
  `cat /sys/bus/iio/devices/iio:device0/in_accel_scale_available`
- Instantaneous scaled acceleration value for axes x, y and z
  `cat /sys/bus/iio/devices/iio:device0/in_accel_<axis>_scale`

- Scaling factor for axes x, y and z

  ```
  echo <scale> > /sys/bus/iio/devices/iio:device0/in_accel_<axis>_scale
  ```

- Event threshold value for axes x, y and z

  ```
  echo <0...127> > /sys/bus/iio/devices/iio:device0/events/
  in_accel_<axis>_thresh_rising_value
  ```

- Threshold enable for axes x, y and z

  ```
  echo 1 > /sys/bus/iio/devices/iio:device0/events/
  in_accel_<axis>_thresh_rising_en
  ```

Example code to capture IIO events from user-space is provided in:

```
drivers/staging/iio/Documentation/iio_event_monitor.c
```

7.4 Audio Control Driver

The Clanton Hill audio control driver (`intel_cln_audio_ctrl`) provides a user-space interface via sysfs to allow selecting one of the supported audio switch configurations to interconnect the interfaces. For details, see Section 7.4.1.

Load the following drivers to enable the audio components on Clanton Hill:

```
modprobe ehci-hcd
modprobe ohci-hcd
modprobe ehci-pci
modprobe intel_cln_audio_ctrl
modprobe snd-usb-audio
modprobe snd-usbmidi-lib
```

After the drivers are loaded, perform a basic sanity test on the audio hardware using the command:

```
aplay -l
```

The sound card displays the following on Clanton Hill:

```
**** List of PLAYBACK Hardware Devices ****
card 0: CODEC [USB audio CODEC], device 0: USB Audio [USB Audio]
Subdevices: 1/1
Subdevice #0: subdevice #0
```

7.4.1 Select audio input/output mode

The audio subsystem has three interfaces which can be interconnected in a fixed set of configurations. These are:

- USB soundcard interface, connected to the CPU
- External speaker/microphone jacks
- Maxim Integrated* 9867 I2S codec that interfaces with a Telit* HE910 GSM modem

The Clanton Hill audio control driver (`intel_cln_audio_ctrl`) provides a user-space interface via sysfs to allow a user to select one of the supported audio switch configurations to interconnect the interfaces:

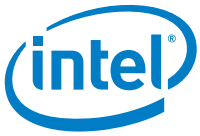
- Set an audio path between the CPU and the GSM modem for cellular voice calls:

  ```
  echo gsm > /sys/bus/i2c/devices/0-0018/audio_ctrl_mode
  ```

  This option is suitable for playback/record of wav file or other audio format with called party.

- Set a uni-directional audio path between the CPU and external speaker output port:

  ```
  echo spkr > /sys/bus/i2c/devices/0-0018/audio_ctrl_mode
  ```



This option is suitable for playback of wav file or other audio format to external (e.g. vehicle) speakers.

- Set a bi-directional audio path between the CPU and external speaker output port:
  ```
  echo spkr_mic > /sys/bus/i2c/devices/0-0018/audio_ctrl_mode
  ```
  This option is suitable for playback/record of wav file or other audio format to external (e.g. vehicle) speaker and microphone.
- Set an audio path between the external speaker/microphone and the GSM modem for "hands-free" cellular voice calls:
  ```
  echo gsm_spkr_mic > /sys/bus/i2c/devices/0-0018/audio_ctrl_mode
  ```
  This option is suitable for 2-way voice call between vehicle occupant and called party.

7.4.2 Audio playback

Configure the audio configuration as described above, using one of the following modes:

```
spkr
spkr_mic
```

Play an audio file to the speakers (e.g. 44KHz 16-bit PCM WAV file):

```
aplay test_wave_44k_16b_stero_pcm_5min.wav
```

7.4.3 Audio record

Configure the audio configuration as described above, using the following mode:

```
spkr_mic
```

Record the audio from the external microphone and save as a WAV file:

```
arecord recording.wav
```

7.5 Bluetooth

Bluetooth functionality is provided by a mini-pcie card connected to the mini-pcie slot on Kips Bay and Clanton Hill platforms. The Intel® Centrino® Wireless-N 135 has been validated on these platforms.

A requirement exists to include the firmware for the Intel® Centrino® Wireless-N 135 in the root filesystem at the following path:

```
/lib/firmware/iwlwifi-135-6.ucode
```

The following drivers must be loaded to enable the audio components on Clanton Hill:

```
modprobe ehci-hcd
modprobe ohci-hcd
modprobe ehci-pci
modprobe btusbl
```

Once loaded, the sysfs entry below should appear:

```
/sys/module/bluetooth
```

The following user-space components are required:

```
bluetoothd
hciconfig
hcitool
```

7.5.1 Device discovery

```
hciconfig <BT DEVICE NAME> noscan
hciconfig <BT DEVICE NAME>
    Expected UP_RUNNING
hcitool scan --flush
hciconfig <BT DEVICE NAME> piscan
```

7.5.2 Service discovery

```
sdptool browse <BT_2_BD_ADDR>
```

7.5.3 Establish connection

```
hcitool dc <BT_ADDR>
hcitool cc <BT_ADDR>
hcitool con
hcitool dc <BT_ADDR>
```

7.5.4 Ping

```
l2ping -c 5 <BT_ADDR>
```

7.6 WiFi

WiFi functionality is provided by a mini-pcie card connected to the mini-pcie slot on Kips Bay and Clanton Hill platforms. The Intel® Centrino® Advanced-N 6205 WiFi Radio Module (Dual Band WiFi, 2.4 and 5 GHz) has been validated on these platforms.

A requirement exists to include the firmware for the Intel® Centrino® Advanced-N 6205 WiFi Radio Module in the root filesystem at the following path:

```
/lib/firmware/iwlwifi-6000g2a-6.ucode
```

Latest firmware for this card can be downloaded from:

http://wireless.kernel.org/en/users/Drivers/iwlwifi/?n=downloads#Firmware

To load a driver for the Intel® Centrino® Advanced-N 6205 WiFi Radio Module, type the following command:

```
modprobe iwlwifi
```

After a successful load of this driver, the following sysfs path is available:

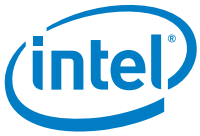
```
/sys/class/net/wlan0
```

7.6.1 Enable / Disable wlan radio

- Get the index of the device
  ```
  rfkill list
  ```
- Disable radio
  ```
  rfkill block 0
  ```
- Enable radio
  ```
  rfkill unblock 0
  ```

7.6.2 Scan for WiFi networks

```
wlist wlan0 scan
```



7.6.3 Configuring a WiFi device

Enter the command:

```
edit  /etc/network/interfaces
```

Add the following:

```
auto wlan0
iface wlan0 inet static
    address <IP ADDRESS>
    netmask <NETMASK>
    wireless_mode managed

    wireless_essid <SSID_NAME>
    wpa-driver wext
    wpa-conf /etc/wpa_supplicant.conf
```

7.6.4 Generating a wpa_supplicant file

This file is used to configure a protected WiFi network.

Generate the WPA Passphrase:

```
wpa_passphrase essid <PassPhrase>
```

Generate the `wpa_supplicant.conf` file:

```
network={
    ssid="essid"
    #psk=<PassPhrase>
    psk=<Result from last command>
}
```

7.6.5 Connecting to a WiFi network

```
ifup wlan0
```

7.6.6 Disconnecting from a WiFi network

```
ifdown wlan0
```

§ §

8.0 Secure Boot Implementation

8.1 Overview

A key feature of the Intel® Quark SoC X1000 is the concept of secure boot. Secure boot means that only authenticated software, that has been cryptographically verified will be run on a Intel® Quark SoC X1000 system.

The concept is predicated on a root-of-trust (RoT) from the reset vector, through to the run-time kernel. Each phase of the boot verifies the next phase of the boot, before handing off to that phase.

In this way, Intel® Quark SoC X1000 reference software stack provides a mechanism to ensure only authenticated software can be booted on a Intel® Quark SoC X1000 system.

There are two variants of Intel® Quark SoC X1000:

- Secure boot enabled
- Non-secure boot

Both variants enable Isolated Memory Regions (IMRs) during boot, through bootloader and kernel. However, only the secure boot version of Intel® Quark SoC X1000 requires cryptographic authentication of images in order to boot.

8.2 Isolated Memory Regions

IMRs are used extensively by grub and Linux to provide extra security during boot. IMRS can be used to define fine-grained access masks to defined memory regions. These access masks prevent bus masters, from accessing particular memory regions based on the definitions of access rights for a given memory region associated with an IMR.

The following table shows the usage of IMRs throughout the boot.

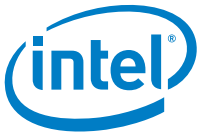


Table 4. IMR Usage During Boot

IMR	ROM	Stage 1	Stage 2	Grub	Linux Boot	Linux Run-time
0		Compressed EDKII stage 2 Uncompressed EDKII stage2 Boot time services Grub Image Stack/Data area	Compressed EDKII stage 2 Uncompressed EDKII stage2 Boot time services Grub Image Stack/Data area	Compressed EDKII stage 2 Uncompressed EDKII stage2 Boot time services Grub Image Stack/Data area	Compressed EDKII stage 2 Uncompressed EDKII stage2 Boot time services Grub Image Stack/Data area	Uncompressed Kernel Read only & initialized data section
1		AP Startup vector	AP Startup vector	Boot Params	Boot Params	
2	UNUSED					
3		Low SMRAM	Low SMRAM		Entire Memory (4G)	
4	UNUSED					
5			Legacy S3 memory	Legacy S3 memory	Legacy S3 memory	Legacy S3 memory
6			ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory	ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory	ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory	ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory
7	eSRAM protection during ROM phase (EDKII Stage 1)	eSRAM protection during ROM phase (EDKII Stage 1)		Compressed Kernel OS Image	Compressed Kernel OS Image	

8.3 Bootloader Security

The reference second stage bootloader solution carries out two important functions in terms of secure boot:

- Asset verification
  - Kernel
  - Bootloader config file - grub.conf
  - InitRD
- IMR setup/teardown
  - IMR setup for kernel boot params
  - IMR setup for compressed kernel image

This reference solution maintains a chain of trust through bootloader into kernel by ensuring that all assets executed have been validated and encapsulated within an IMR.

8.3.1 Asset Verification Flow

Grub verifies any kernel, init-ramdisk or grub configuration file, it relies upon in secure boot mode.

Grub executes the boot logic given to it in `grub.conf`. The `grub.conf` file specifies which boot assets are signed and which are not. The `grub.conf` file also specifies where to find boot assets. Supported locations are:

- SPI Flash
- SD/USB mass storage device

In secure boot mode, grub will:

- Parse the master flash header to identify the location of `grub.conf`
- Read in the contents of `grub.conf`
- Verify `grub.conf` against a cryptographic signature
- For each item marked as secure in the grub config file
  - — Search for an asset signature
  - — Verify the asset agains the asset signature

For any of the previous steps, a failure to find an asset or an asset signature, or a failure to verify an asset against an asset signature will result in grub executing an EDK callback to initiate the EDKII recovery mechanism.

8.3.2 Isolated Memory Region Flow

Grub is booted by EDK with IMRs already configured around a number of assets as indicated by Figure 5.

As part of the reference secure boot solution, grub will read a Linux kernel image from SPI flash or from USB/MMC mass storage.

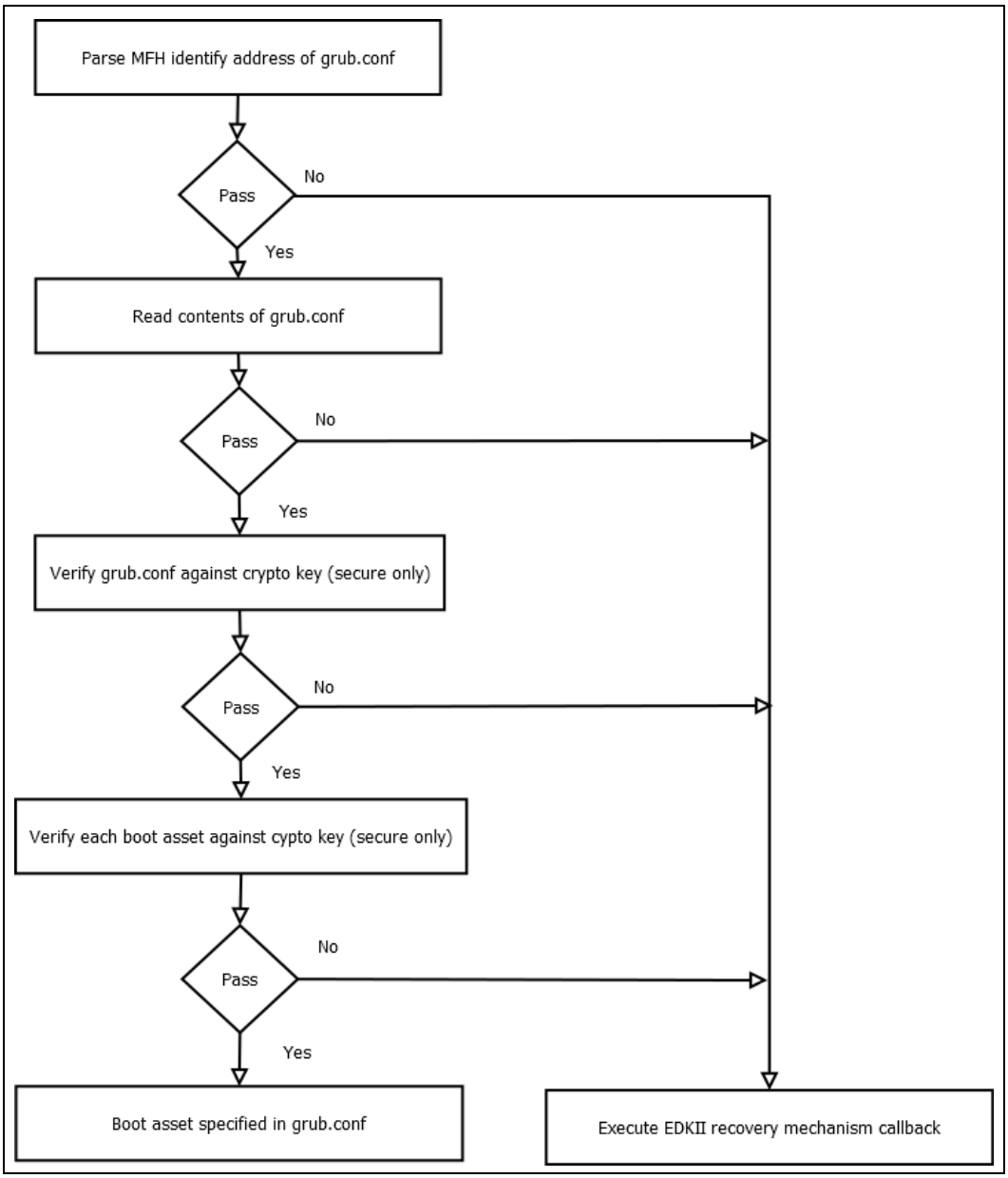
IMRs are used in the following fashion:

- IMR1 used to protect kernel boot params structure
- IMR7 used to protect the compressed bzImage in memory

Grub subsequently verifies bzImage against the cryptographic key for bzImage once the compressed image is placed within the IMR protection region.

Finally, assuming verification succeeds, control is handed from grub to the compressed kernel image with IMRs active for IMR1 and IMR7, restricting access to CPU read/write only.

Figure 5. Grub Secure Boot Flow



8.4 OS Security

The reference OS solution for Intel® Quark SoC X1000 adds IMR protection to the uncompressed kernel as well as bringing the system to a final state in terms of IMR protection.

Specifically, the reference OS solution:

- Places an IMR around executable sections of the kernel image.
- Tears down any IMRs that are not required for the run-time system.

- Locks any unlocked IMRs.
- Provides a convenient debug interface to view the size, extent and state of each IMR.

8.4.1 Early Boot IMR Support

Early in the kernel boot process, before decompression takes place, an IMR is placed around the base physical address of the kernel image to the maximum memory address range, that is, 4 gigabytes.

This is necessary to ensure that the decompressed kernel runs inside of an IMR protected region.

Later phases of the kernel boot set up a smaller IMR around the run-time kernel when the necessary data to derive the correct address range becomes available.

After setting up the initial run-time kernel IMR, the early kernel boot code removes the following IMRs:

- `grub` - IMR0
- `boot params` - IMR1
- `bzImage` - IMR7

8.4.2 Run Time IMR Support

The IMR run-time code, is distinct from the IMR “early” code. Early code on Linux is typically defined as code that emulates a more advanced run-time functionality with diminished features.

The reference IMR run-time solution on Intel® Quark SoC X1000 locks all IMRs by default.

An option is provided by the IMR run-time driver not to lock all IMRs by default. The module parameter to unlock the pre-locked IMRs may only be passed in grub.

```
intel_cln_imr.imr_lock=0
```

With IMRs unlocked, it is possible for a user of the IMR driver to allocate and free IMRs using the following in-kernel API.

8.4.2.1 intel_cln_imr_alloc

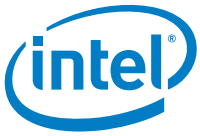
```
int intel_cln_imr_alloc(u32 high, u32 low, u32 read, u32 write,
                        unsigned char *info, bool lock);
```

- high: the end of physical memory address
- low: the start of physical memory address
- read: IMR read mask value
- write: IMR write maks value
- imr: information a descriptive name for the IMR
- lock: Boolean to indicate whether to lock the IMR

This routine allows allocation of an IMR with any of the access rights given below for reading and writing individually. It is possible to lock this IMR upon allocation. If locked an IMR cannot be torn down without a reset of the system.

Access mask bits associated with read and write are:

```
#define IMR_ESRAM_FLUSH_INIT    0x80000000  /* esram flush */
```



```
#define IMR_SNOOP_ENABLE        0x40000000  /* core snoops */
#define IMR_PUNIT_ENABLE        0x20000000 /* PMU snoops */
#define IMR_SMM_ENABLE          0x02        /* core SMM access */
#define IMR_NON_SMM_ENABLE      0x01        /* core non-SMM access */
```

For convienience, a default access mask is defined:

```
/* snoop + Non SMM write mask */
#define IMR_DEFAULT_MASK (IMR_SNOOP_ENABLE \
                                + IMR_ESRAM_FLUSH_INIT \
                                + IMR_NON_SMM_ENABLE)
```

8.4.2.2 intel_cln_imr_free

```
int intel_cln_imr_free(u32 high, u32 low);
```

- high: high boundary of memory address
- low: low boundary of memory address

This function removes an IMR based on input memory region specified at high and low.

8.4.3 Debug Interface

For the purposes of system debug, an interface is provided in `/sys` to view the setup of the IMRs on a booted reference Intel® Quark SoC X1000 system.

Read data from `/sys/class/imr` to view the address range of each IMR[0-7] and its state, in the run time system.

§ §