Xtrinsic Intelligent Sensing Framework v1.1 Software Reference Manual

on the FXLC95000 Intelligent Motion-Sensing Platform

Document Number: ISF1P195K_SW_REFERENCE_RM
Rev 1.0, 2/2014

freescale
Xtrinsic

Contents

Section number **Title** **Page**

Chapter 1
About This Document

1.1 Purpose....7

1.2 Audience....7

1.3 Terminology and Conventions....7

1.3.1 Technical Terms....7

1.3.2 Abbreviations....9

1.3.3 Notational Conventions....10

1.4 References....11

Chapter 2
Introduction

2.1 FXLC95000 System Overview....13

2.1.1 Development System Requirements....15

2.2 FXLC95000CL Hardware Device....15

Chapter 3
Intelligent Sensing Framework

3.1 ISF Architecture....17

3.2 Application Support....19

3.2.1 Freescale MQX™ RTOS Components....19

3.2.1.1 Component Configuration....20

3.2.1.2 ISF Tasks and Initialization....20

3.2.2 Freescale MQX™ RTOS Port for FXLC95000....22

3.2.2.1 Bootup Reset....22

3.2.2.2 RTOS Timer....22

3.2.2.3 Linker File Memory Map....23

3.2.2.4 Queued SPI Module Configuration....24

3.2.2.5 Task Memory Deallocation....24

3.2.2.6 Servicing Non-Maskable Interrupts....25

3.2.3 Freescale MQX™ RTOS Task Preemption Control Considerations....25
3.3 Communications....26
3.3.1 ISF Command Interpreter....26
3.3.2 Mailbox Interface....27
3.3.3 The Command/Response Paradigm....28
3.3.3.1 Command Response Mailbox Layout....29
3.3.3.2 Command Processing....30
3.3.4 The Quick-Read Paradigm....32
3.3.4.1 Quick-Read Mailbox Layout....34
3.3.4.2 Configuring the Quick-Read Mailboxes....35
3.3.4.3 Mailbox Application Control Config Register....40
3.3.5 Device Messaging....41
3.3.5.1 Device Messaging concepts....42
3.3.5.2 Usage Example....44
3.3.6 Host Proxy....45
3.3.6.1 Host Proxy Concepts and Theory of Operation....45
3.3.6.2 Implementing a Proxy Application using the Host Proxy....47
3.3.7 I2C Master Interfaces....53
3.3.7.1 Theory of Operation....53
3.3.7.2 Bus Locking....53
3.3.7.3 Usage Example....54
3.3.8 Communications Channel Configuration....55
3.3.9 Bus Management....55
3.3.10 Built-in Commands....57
3.3.10.1 Device Info command....57
3.4 Sensor Management....59
3.4.1 Sensor Manager Signal Tap Mechanism....61
3.4.2 Sensor Manager Subscription Tokens....61

3.4.3 Using the Sensor Management API 61
3.4.3.1 Sensor Manager Subscription Quality of Service....64
3.4.3.2 Sensor Subscription Compatibility 65
3.4.4 Digital Sensor Abstraction (DSA)....66
3.4.4.1 Digital Sensor Abstraction Theory of Operation....67
3.4.4.2 Implementing a New Digital Sensor Abstraction Adapter....68
3.4.5 System Configuration....69
3.5 Power Management....70
3.5.1 Power Management Concepts and Theory of Operation....70
3.5.2 Power Management Design....71
3.5.3 Power Level implementations for the FXLC95000....72
3.5.4 Using the Power Management Interfaces....73
3.5.5 Timer Service....73
3.6 Application Integration....74
3.6.1 Interrupt Output Integration....77

Chapter 1
About This Document

1.1 Purpose

This reference manual describes the features, architecture, and programming model of the FXLC95000 system. The system includes the Xtrinsic FXLC95000CL device, the Freescale MQX™ firmware, the Xtrinsic Intelligent Sensing Framework (ISF) firmware, and embedded applications which leverage the ISF. This document focuses on the core ISF functionality and its use to build user embedded applications. Comprehensive documents that focus on installation instructions, release notes, application development and API for ISF are listed in References and are available on the Xtrinsic ISF website.

1.2 Audience

This document is primarily for system architects and software application developers currently using or considering use of the FXLC95000 platform as the basis for an intelligent sensor system in an end-user product.

1.3 Terminology and Conventions

This section defines the terminology, abbreviations, and other conventions used throughout this document.

1.3.1 Technical Terms

application callback ID	The identifier used by the Command Interpreter to determine which registered callback function is invoked by the Command Interpreter on behalf of the embedded application. Depending on the context, the terms *application callback ID* or *application ID* or *callback ID* may be used.

Table continues on the next page...

application ID	See *application callback ID*.
BusHandle	A handle identifying the bus to use for I^2C transactions.
callback	See *callback function*.
callback ID	See *application callback ID*.
callback function	A function registered by a software component, invoked on behalf of the registering component. The function usually contains instructions to communicate with or call back to the registering component. Also referred to as *callback*.
channel	A representation of a separate communications pathway to one or more external slave devices.
ChannelDescriptor	A descriptor identifying the channel for communications using Device Messaging.
DeviceHandle	A handle identifying the device used for Device Messaging transactions.
Digital Sensor Abstraction	Abstraction layer in the Sensor Manager to enable communications with multiple types of sensors.
embedded application	A program that executes on the intelligent sensing platform as an independent unit of functionality. It consists of a set of one or more tasks providing outputs consumed outside the intelligent sensing platform. Independence means that an application may be added or removed from a firmware build without interfering with the functionality of other applications. Applications typically are run on behalf of a user as opposed to a simple support task which is run as part of the Intelligent Sensing Framework.
end-user product	A third-party product that hosts a sensing sub-system.
event group	A 32-bit group of event bits used to let tasks synchronize and communicate. There are two event group types: fast event groups and named event groups.
event number	The category number, which could be either configuration or data ready.
firmware	The combination of code and data stored in a device's flash memory.
framework	The infrastructure code providing the execution environment for embedded applications.
function	A portion of code taking a pre-defined set of input parameters that performs a series of instructions and returns a pre-defined set of output values. A function may be invoked from one or more points in an executable program.
FXLC95000	FXLC95000 firmware—The combination of code and data stored in the FXLC95000CL device's flash memory. FXLC95000 intelligent sensor system—The combination of the FXLC95000 platform and external sensor hardware that interact together via hardware and software protocols. Also referred to as *FXLC95000 system*. FXLC95000 platform—The combination of the FXLC95000CL device and FXLC95000 firmware. Also referred to as *FXLC95000* and *platform*. FXLC95000 system—The FXLC95000 platform and external sensor hardware that interact together via hardware and software protocols. Also referred to as *FXLC95000 intelligent sensor system*.
FXLC95000CL	The physical packaged part. Also referred to as *FXLC95000CL hardware device*.
host application	A program that executes on the host processor.
host proxy	An Intelligent Sensing Framework (ISF) component that allows a host processor to remotely configure and subscribe to all the managed sensors.
intelligent sensing platform	The combination of the device and firmware. Also referred to as *platform*.
Intelligent Sensing Framework (ISF)	The Freescale-provided software middleware layer enabling the development of custom embedded sensor applications with increased portability and ease-of-use, and decreased time-to-market.
intelligent sensor system	The platform and external sensor hardware that interact together via hardware and software protocols. Also referred to as *system*.
period	The time between successive repetitions of a given phenomena. Period is equal to the inverse of frequency).

Table continues on the next page...

platform	The combination of the device and firmware. Also referred to as *intelligent sensing platform*.
proxy number	A unique number assigned at the time the application is registered with the host proxy.
Sensor Adapter	A Sensor Adapter implements the Sensor Manager's Digital Sensor Abstraction interface for a particular physical sensor and handles the device-specific communications and interactions with the physical sensor allowing the Sensor Manager to manage sensors at a higher level of abstraction. ISF requires a Sensor Adapter for each sensor being managed in the system.
sensor ID	The enumerated value that indexes into the global sensor configuration array.
service family	A logical grouping of software components providing related functionality.
signal tap	An access mechanism to sensor data. Also referred to as *tap*.
SlaveHandle	A handle identifying the slave device used for I^2C transactions.
system	The platform and external sensor hardware that interact together via hardware and software protocols. Also referred to as *intelligent sensor system*.
tap	An access mechanism to sensor data. Also referred to as *signal tap*.
task	An operating entity within the Intelligent Sensing Framework (ISF) scheduled to execute by the Freescale MQX™ RTOS. A task may entail the execution of one or more functions.
transport	Communications mechanism. Examples: I^2C, Bluetooth©, Ethernet, and USB
token	Result of a successful callback function registration with the bus manager used in subsequent bus management calls to refer to a registered callback function.
vector base register	A register in the ColdFire memory map that controls the location of the exception vector table.

1.3.2 Abbreviations

AFE	Analog Front End
API	Application Programming Interface
BM	Bus Manager
BSP	Board Support Package
CI	Command Interpreter
CMD	Command
COCO	Conversion Complete (hardware), Command Complete (software)
CRC	Cyclic Redundancy Check
CSR	Control and Status Register
DFC	Data Format Code
DM	Device Messaging
DSA	Digital Sensor Abstraction
FOPT	Flash OPTions register
IIR	Infinite Impulse Response
ISF	Intelligent Sensing Framework
ISP	Intelligent Sensing Platform
ISR	Interrupt Service Routine
MB	Mailbox
NMI	Non-Maskable Interrupt

Table continues on the next page...

PDB	Programmable Delay Block
PM	Power Manager
POR	Power-On-Reset
PSP	Processor Support Package
QR	Quick-Read
QSPI	Queued SPI
RCSR	Reset Control and Status Register
SM	Sensor Manager
SOF	Start Of Frame
SPI	Serial Peripheral Interface
VBR	Vector Base Register

1.3.3 Notational Conventions

cleared/set	When a bit has the value 0, it is said to be cleared; when it has a value of 1, it is said to be set.
MNEMONICS	Mnemonics which may represent command names, defined macros, constants, enumeration values are shown as, for example, `CI_DEV_INFO`.
`programming domain entity`	Entities such as functions, data structures are shown as, for example `device_info_t`.
0x	Prefix to denote a hexadecimal number
h	Suffix to denote a hexadecimal number
nibble	A 4-bit data unit
byte	An 8-bit data unit
word	A 16-bit data unit
longword	A 32-bit data unit

CAUTION, *Note*, and *Tip* statements may be used in this manual to emphasize critical, important, and useful information. The statements are defined below.

CAUTION

A CAUTION statement indicates a situation that could have unexpected or undesirable side effects or could be dangerous to the deployed application or system.

Note

A Note statement is used to point out important information.

Tip

A Tip statement is used to point out useful information.

1.4 References

1. FXLC95000CL Intelligent, Motion-Sensing Platform Data Sheet: FXLC95000CL, Freescale Semiconductor, Rev. 1.2, 08/2013 http://www.freescale.com/files/sensors/doc/data_sheet/FXLC95000CL.pdf
2. FXLC95000CL Intelligent, Motion-Sensing Platform Hardware Reference Manual: FXLC95000CLHWRM, Rev. 0.6, 05/2013 http://www.freescale.com/files/sensors/doc/ref_manual/FXLC95000CLHWRM.pdf
3. Release Notes for Intelligent Sensing Framework 1.1 on the FXLC95000 Intelligent Motion-Sensing Platform: ISF1P1_95K_CORE_LIB_RELEASE_RN, Rev 1.0, 12/2013. http://www.freescale.com/files/sensors/doc/support_info/ISF1P1_95K_CORE_LIB_RELEASE_RN.pdf
4. Installation Instructions for Intelligent Sensing Framework 1.1 on the FXLC95000 Intelligent Motion-Sensing Platform: ISF1P1_95K_CORE_LIB_INSTALL_INS, Rev 1.0, 12/2013. http://www.freescale.com/files/sensors/doc/support_info/ISF1P1_95K_CORE_LIB_INSTALL_INS.pdf
5. API Reference Manual for Intelligent Sensing Framework 1.1 on the FXLC95000 Intelligent Motion-Sensing Platform: ISF1P1_95K_API_REFERENCE_RM, Rev 1.0, 12/2013. http://www.freescale.com/files/sensors/doc/support_info/ISF1P1_95K_API_REFERENCE_RM.pdf
6. User's Guide for Applications and Templates for Intelligent Sensing Framework 1.1 on the FXLC95000 Intelligent Motion-Sensing Platform: ISF1P1_95K_APPS_TMPL_PG, Rev 1.0, 12/2013. http://www.freescale.com/files/sensors/doc/support_info/ISF1P1_95K_APPS_TMPL_PG.pdf
7. User's Guide for Reference Applications for Intelligent Sensing Framework 1.1 on the FXLC95000 Intelligent Motion-Sensing Platform: ISF1P1_95K_ELF_INSTALL_INS, Rev 1.0, 12/2013. http://www.freescale.com/files/sensors/doc/support_info/ISF1P1_95K_ELF_INSTALL_INS.pdf
8. Freescale MQX™ RTOS Reference Manual: MQXRM, Rev. 6 04/2011 http://www.freescale.com/files/32bit/doc/ref_manual/MQXRM.pdf
9. Freescale MQX™ RTOS User's Guide: MQXUG, Rev. 3 04/2011 http://www.freescale.com/files/32bit/doc/user_guide/MQXUG.pdf

Chapter 2
Introduction

2.1 FXLC95000 System Overview

The FXLC95000 system includes one FXLC95000CL device, Freescale MQX™ RTOS firmware, Intelligent Sensing Framework (ISF) firmware, user embedded applications, and an optional host application.

The FXLC95000CL device is the second in Freescale's line of intelligent sensor devices and integrates a low-g 3-axis MEMS accelerometer, a low-power 32-bit programmable CPU for digital signal processing as well as several peripheral functional blocks. For more information on hardware features, refer to the Xtrinsic FXLC95000CL Intelligent Motion-Sensing Platform data sheet, listed in References.

The presence of the 32-bit CPU transforms the FXLC95000CL device to an intelligent sensing platform and distinguishes it from an ordinary digital smart sensor. The FXLC95000CL device offers the ability to offload the signal processing and external device management functions traditionally performed by the host processor onto the intelligent sensing platform. Doing so has the potential to reduce the overall power consumption of the device and/or free some host processor cycles for other tasks.

Unlike sensor devices with fixed firmware images or embedded functions with limited configurability which severely constrain customization, the FXLC95000CL device provides users an open programming environment. This environment allows users with widely varying and growing functional requirements the ability to add user-specific functionality to the system. A software framework hosted on the FXLC95000CL provides a common, easy to use interface which empowers users to quickly develop hardware-independent, portable, real-time application code. This framework is named the **Xtrinsic Intelligent Sensing Framework (ISF)**.

Supporting this framework is the Freescale real-time operating system, Freescale MQX™ RTOS.

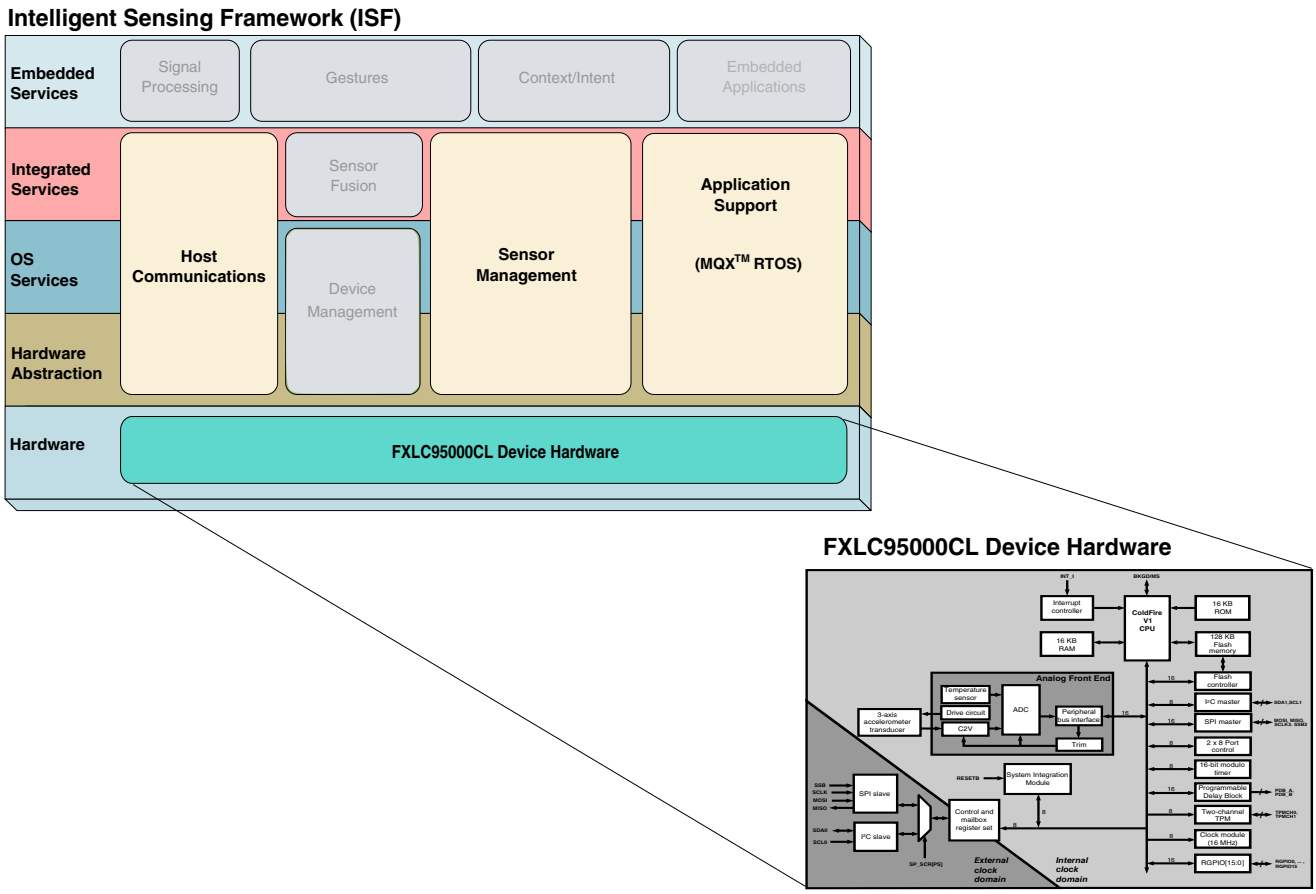

Figure 2-1. FXLC95000 system components

See Figure 2-1 for a graphical overview of the FXLC95000 system components. The FXLC95000 system components providing the functionality required for developing sensor applications are as follows:

- **Device hardware:** The FXLC95000CL device hardware incorporates dedicated accelerometer MEMS transducers, signal conditioning, data conversion, and a 32-bit microcontroller with 16K ROM, 16K RAM and 128K flash memory, timers, GPIO, SPI, and I^2C.
- **ISF:** ISF provides the capability to subscribe to external, as well as on-board sensor data and read such data at various rates. It also supports communication between the host processor and the FXLC95000 platform via the slave port mailboxes. ISF allows the FXLC95000 to act as a sensor hub for external sensors and to manage that data for the host processor.
- **MQX:** Freescale MQX™ RTOS is a run-time library of functions that provides real-time multi-tasking capabilities to user applications. It operates as a priority-

preemptive operating system. The main features are its scalable size, component-oriented architecture, and ease of use.

- **FXLC95000 support:** Freescale's CodeWarrior development tool provides customers the means to load and run their applications on the FXLC95000 platform. CodeWarrior versions 10.4 and above are currently supported.

2.1.1 Development System Requirements

For developing applications with the FXLC95000 system, CodeWarrior 10.4 or later must be installed in a Microsoft® Windows® environment.

2.2 FXLC95000CL Hardware Device

The FXLC95000CL hardware is programmable with user-specific firmware to create an intelligent high-precision, and highly flexible, motion-sensing platform able to manage multiple sensor inputs. The user-specific programming flexibility allows the device to be used for sophisticated applications such as e-compass with tilt compensation and calibration, pedometer, and gesture recognition, .

Figure 2-2 shows FXLC95000CL hardware blocks and interfaces.

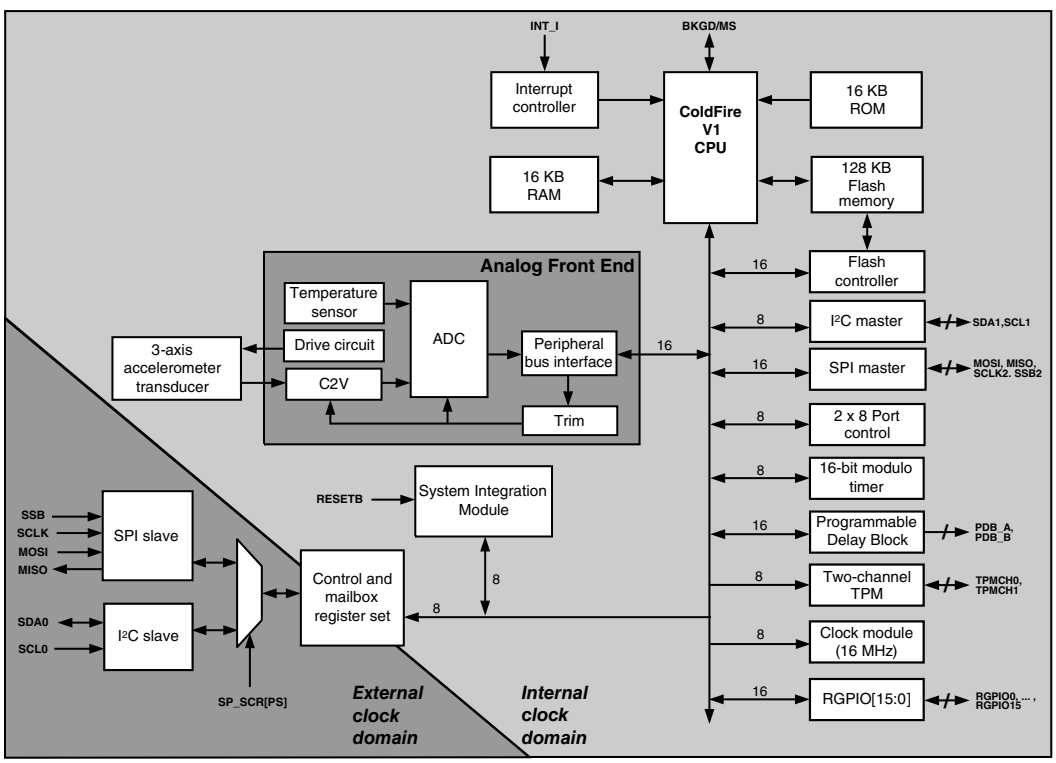

Figure 2-2. Block diagram of the FXLC95000CL

Chapter 3
Intelligent Sensing Framework

The Intelligent Sensing Framework (ISF) consists of the three service families listed below. Each service family provides a set of related capabilities exposed through service family components.

- Application Support provided by Freescale MQX™ RTOS
- Communications
- Sensor Management

3.1 ISF Architecture

The ISF architecture has been developed by taking into account a large set of requirements from various sensor related application domains (motion sensing, orientation detection, e-compass, etc.). These requirements are divided into distinct areas of domain-level functionality.

ISF partitions the domain-level functions into well-defined low-level components or specific areas within those components. By doing so, ISF separates platform-specific interfaces from the general domain-level functionality. The low-level components or specific areas within components are highly portable so as to minimize the impact of supporting multiple platforms and deployment environments. Closely related components are organized into a named group referred to as a service family. The ISF architecture can best be visualized by the block diagram in Figure 3-1.

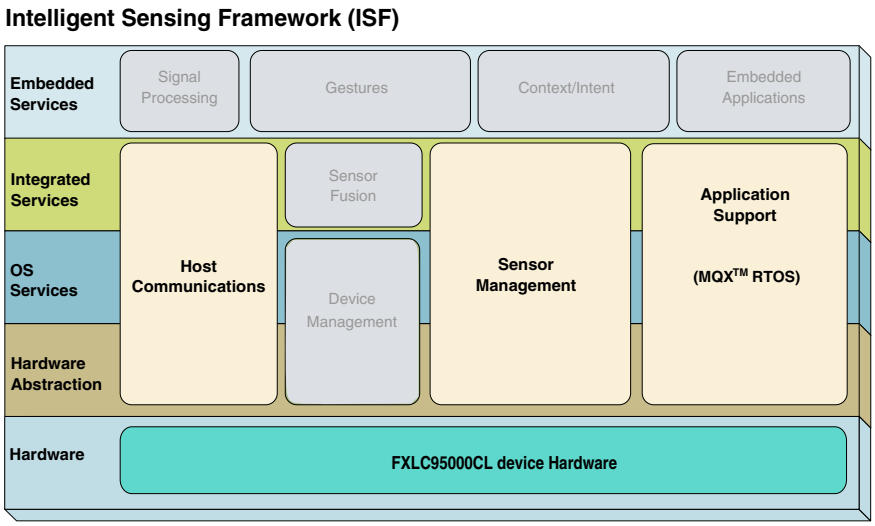

Figure 3-1. ISF Architecture

The ISF software stack is built in layers in a bottom up fashion starting with the Hardware Abstraction layer and by adding progressively advanced services in each layer above it. The top layer of the stack is the Embedded Services layer which provides powerful functions for the user to build intelligent sensor applications. The diagram also shows the service families which span the layers. Note that the service families in gray in the block diagram represent future functionality and are not present in ISF v1.1. The functionality grouped together within each ISF service family is briefly described below and more detailed documentation is available on features, configuration and use of the functionality provided.

The **Host Communications** service family provides the ability to pass data into and out of the Intelligent Sensing Platform via various interfaces such as the Slave port (I^2C and SPI and external interrupt). It provides a common abstraction for these various interfaces as well as some domain-specific application-layer protocol implementations for use when operating as a slave device on the Slave I^2C and Slave SPI buses.

The **Sensor Management** service family provides uniform interfaces to applications running on either the intelligent sensing platform or the external host for accessing physical data measured by the on-board accelerometer. The ISF Sensor Manager component (server) provides sensor data to registered applications (clients) that need sensor data. The Sensor Manager interface allows each application to subscribe to a sensor's data samples at different rates.

The **Application Support** service family provides the basic run-time operating environment and exposes the system functionality necessary to support the execution of embedded applications on the Intelligent Motion-Sensing Platform. This includes such things as scheduling of an application’s threads, synchronization mechanisms, memory

management, and access to timer and clock functionality. In addition, Application Support provides some higher-level management capabilities for collecting and publishing application run-time status information.

Software components from the above mentioned service families are packaged into libraries. The ISF core library packages several software components to form the set of base functionality. Additional functionality provided by ISF is contained in separately deployable extension units (libraries) of functionality, each enabling a different capability or set of related capabilities. These add-on units may make use of functionality provided in the core and additional functionality from other extensions.

3.2 Application Support

This section provides details of Freescale MQX™ RTOS as configured for the ISF implementation in FXLC95000, and its usage. The ISF implementation provided for the FXLC95000 is based upon Freescale's MQX™ RTOS version 3.7. For more information, refer to the Freescale MQX™ RTOS Reference Manual listed in References.

3.2.1 Freescale MQX™ RTOS Components

The Freescale MQX™ RTOS component configuration is managed by the user via header file preprocessor macros. Configuration is done in header files that are part of the FXLC95000 port. When a change to the configuration is made, the Freescale MQX™ RTOS build must be recompiled to incorporate the changes.

The MQX for FXLC95000 software package is configured to use the lightweight Freescale MQX™ RTOS components. These components provide basic functionality, require minimal memory, and run quickly. For more information on the lightweight Freescale MQX™ RTOS components, refer to the Freescale MQX™ RTOS User's Guide listed in References.

The following Freescale MQX™ RTOS components are included in the MQX for FXLC95000 software package.

- Lightweight event
- Lightweight memory management
- Lightweight messaging
- Lightweight logs
- Lightweight timer

In some cases, more functionality may be needed by the user than the lightweight components provide. Freescale MQX™ RTOS also provides heavyweight components with more features that could be included in a user embedded application. The following Freescale MQX™ RTOS heavyweight components are included in the MQX for FXLC95000 software package.

- Semaphores

The heavyweight semaphore component provides priority inheritance whereas the lightweight semaphore does not. The heavyweight components do incur more overhead in terms of memory and latency. Therefore, the following heavyweight Freescale MQX™ RTOS components are not configured in ISF but may be added to a user embedded application if required.

- Events
- Mutexes
- Memory management
- Messages
- Named objects
- Partitions
- Logs
- Timer
- Software watchdog

3.2.1.1 Component Configuration

The Freescale MQX™ RTOS allows components to be configured through header files. In the FXLC95000 port, the optional heavyweight components are configured in the *fxlc95000eval_ram_config.h* header file. This header file can be found in the following folder after a default installation:

C:\Program Files\Freescale\Freescale MQX 3.7\lib\Fxlc95000eval.cw10

Defines are contained within the header file to include or exclude the optional components by setting a value of 1 to include or 0 to exclude. For more information on Freescale MQX™ RTOS components and configuration, see the Freescale MQX™ RTOS User's Guide listed in References.

3.2.1.2 ISF Tasks and Initialization

ISF has several tasks that must be initialized and running to support its services. These include but are not limited to the Command Interpreter task, the Sensor Manager task, and the Bus Manager task. ISF makes it easy to declare these tasks under Freescale MQX™ RTOS by creating a preprocessor macro called `ISF_TASKS_ATTRIBUTES`. This macro expands to declare all of the ISF tasks in a form that is usable inside the `TASK_TEMPLATE_STRUCT MQX_template_list` required by Freescale MQX™ RTOS.

```
// Task Template Structure
TASK_TEMPLATE_STRUCT MQX_template_list[] = {
   { 5, main_task,   512, 13, "main",  MQX_AUTO_START_TASK },
     ISF_TASKS_ATTRIBUTES
   { 8, user_task1, 2000, 15, "test1", MQX_AUTO_START_TASK },
   { 0, 0, 0, 0, 0, 0 }
};
```

The `ISF_TASKS_ATTRIBUTES` macro automatically accounts for the user's configuration settings selected in the file *isf_user_config.h* where additional macros are defined allowing a user to choose which ISF components are to be enabled in their system. In addition to declaring the ISF tasks, it is necessary to call initialization functions for some of these components. ISF provides a top-level initialization function that must be called prior to using the ISF component functionality. To ensure that this task runs first, ISF declares a special `isf_init_task` that is also placed in the task list by the `ISF_TASKS_ATTRIUBUTES` macro. This init task is declared with highest priority to guarantee that it executes first. Once the necessary component initialization routines have been executed, the init task exits and any memory it consumed is released back to the operating system.

The `isf_init_task` also generates a MQX lightweight event called SYSTEM_READY_EVENT at the completion of the ISF initialization process. In order to guarantee proper initialization, user-defined tasks must wait for this event prior to execution of their own initialization. The following code can be inserted at the beginning of the task to wait for the event:

```
// Wait for ISF system initialization to complete.
isf_system_sync();
```

Care should be taken to honor the task priority assignments made by ISF for proper system operations. User tasks must not be assigned at higher priority levels than ISF system tasks. As configured, out-of-the-box ISF tasks are at priorities between 9 and 12. Therefore, user tasks are expected to run at priority numbers of 13 or greater. ISF task IDs start at 50 and increase. User-defined tasks are expected to be defined with task IDs less than 50.

In the example, two user tasks are declared in addition to the ISF tasks. The first task called `main` has task ID = 5, and entry point `main_task()`. It is configured for 512 bytes of stack space, has priority level 13, and is started automatically by Freescale MQX™ RTOS.

The `ISF_TASKS_ATTRIBUTES` macro expands into task definitions for the ISF tasks, for example, the Sensor Manager, the Command Interpreter, and the Bus Manager. The second user task is named `test1`, has task ID = 8, is configured for 2000 bytes of stack space, has priority level 15 and is also started automatically by Freescale MQX™ RTOS.

As specified by Freescale MQX™ RTOS, a task definition entry of all zeros marks the end of the task definition array.

3.2.2 Freescale MQX™ RTOS Port for FXLC95000

The Freescale MQX™ RTOS is ported to run on the FXLC95000CL hardware device. The porting changes are isolated to the Freescale MQX™ RTOS for FXLC95000.

3.2.2.1 Bootup Reset

As a consequence of the FXLC95000 port changes, the Freescale MQX™ RTOS boot process has the ability to reset to internal ROM code to avoid a lockup cycle if the previous reset occurred due to illegal conditions. Forcing a reset to internal ROM code prevents the CPU from encountering the same illegal instruction or address again and going into an endless reset cycle. In this situation, the debug port may not be able to stop the CPU, the debugger may fail, and the user may not be able to download new code.

To detect the cause of the previous reset, the boot code examines the Reset Control and Status (RCSR) register. The RCSR register contains bits to indicate if the last CPU reset was due to an illegal instruction or address. If the reset was due to any illegal condition, then the boot code directs the CPU to boot into the internal ROM code and perform a software reset. The CPU runs in ROM command interpreter mode when the software reset is complete, allowing the debugger to download new code.

3.2.2.2 RTOS Timer

Freescale MQX™ RTOS, like many other RTOSes, uses a timer to generate an interrupt at a predefined interval to run the kernel's scheduler. For the FXLC95000CL device, the Programmable Delay Block (PDB) timer is reserved for use by the RTOS. The PDB timer generates an interrupt every 1 millisecond.

CAUTION

The user should not use the PDB timer and must not modify any of the PDB registers listed here.

PDB Register	Address
Control & Status Register (CSR)	0xFFFFEC00
Delay A Register (DELAYA)	0xFFFFEC02
Delay B Register (DELAYB)	0xFFFFEC04
Modulus Register (MOD)	0xFFFFEC06

3.2.2.3 Linker File Memory Map

A default linker file is provided for the user application to link with the correct flash and RAM memory space available. The linker file, *intflash_mqx.lcf*, is included in the Freescale MQX™ RTOS for FXLC95000 and can be found in the following folder:

C:\Program Files\Freescale\Freescale MQX 3.7\lib\Fxlc95000eval.cw10\bsp

The FXLC95000 has 128KB of flash memory with address range from 0x0 to 0x20000. It has 16KB of RAM memory that ranges from 0x800000 to 0x804000. There are small pockets of memory that are reserved as shown in Table 3-2 and Table 3-3. The supplied default linker file configures the flash and RAM reserved memory areas to prevent accidental use by the user application. It targets the user application code and data to the non-reserved region noted in Table 3-2 and Table 3-3.

Table 3-2. RAM memory space allocation

RAM Memory Range	Memory Size (bytes)	Description
0x800000 to 0x800023	36	Free for use by applications
0x800024 to 0x8000FB	216	Reserved for internal ROM code
0x8000FC to 0x804000	16,132	Free for use by applications

In the flash memory space, the first part of the memory stores exception vectors.

Table 3-3. Flash memory space allocation

RAM Memory Range	Memory Size (bytes)	Description
0x00000 to 0x001CF	464	Reserved for exception vectors
0x001D0 to 0x1FFFB	130604	Free for user application
0x1FFFC to 0x1FFFF	4	Reserved for the FOPT register

Most of the FLASH and RAM memory is free for the user application. The user application consists of the following:

- Freescale MQX™ RTOS, including board support package (BSP) and processor support package (PSP)
- additional Freescale MQX™ RTOS and ISF components
- any user-written code

3.2.2.4 Queued SPI Module Configuration

The Queued SPI (QSPI) module in the FXLC95000CL contains signals that are connected to the FXLC95000CL pins at power up. Some of these signals may be in a dangling state until the QSPI module is enabled. This condition may cause a small amount of current to leak and unneccesarily consume power.

To prevent this from happening, the boot code configures the QSPI pin to FXLC95000CL Rapid General Purpose I/O (RGPIO) input pins. If the QSPI module is used by an embedded application, then it is necessary to configure these pins to QSPI functionality. The RGPIO_ENB register at address 0x00C00000 should be modified to switch between QSPI and RGPIO functionality.

The QSPI signals that have been redirected as RGPIO input functions are shown in Table 3-4.

Table 3-4. QSPI Signals Redirected as RGPIO Input Functions

Pin #	QSPI Pin Description	RGPIO Signals
18	Master Clock	RGPIO10
19	Master output/Slave input	RGPIO11
20	Master input/Slave output	RGPIO12
21	Slave select	RGPIO13

3.2.2.5 Task Memory Deallocation

At task initialization, the Freescale MQX™ RTOS for FXLC95000 kernel allocates a dedicated memory area for the task. The size of the memory area is based on the task descriptor overhead to maintain the task state and the requested stack size. A task is permitted to allocate private memory “pools” within this dedicated kernel memory block during execution. If these private memory pools are not deallocated when the task exits, they continue to be held by the Freescale MQX™ RTOS kernel by default and cannot be reused. The Freescale MQX™ RTOS specifically modified for the FXLC95000 automatically deallocates private memory pools when the task exits.

In addition, there is a need for tasks to share persistent data allocated inside a task's dedicated memory block even after the original task exits. Thus, there is an additional modification for the Freescale MQX™ RTOS specifically modified for the FXLC95000 which allows a task's dedicated memory to persist after the original task exits.

3.2.2.6 Servicing Non-Maskable Interrupts

The ColdFire core in the FXLC95000CL device has seven levels of interrupt priority, with level 7 being the highest level. The level 7 interrupt is a non-maskable interrupt (NMI). The FXLC95000CL device has three sources of NMIs:

- INT_I (external interrupt)
- Frame Error
- Software Interrupt Level 7 (SWI7)

NMIs cannot be masked. Therefore, the execution code can be interrupted at any time. However, the Freescale MQX™ RTOS kernel contains critical sections of code that should not be interrupted. Such interruptions may cause corruption of the stack pointer. Because of this limitation, the user cannot use the Freescale MQX™ RTOS API `_int_install_isr()` to install an interrupt service routine (ISR) to service NMIs.

In order to properly handle NMIs, the Freescale MQX™ RTOS for FXLC95000 redirects them to a lower level SWI6 interrupt. The user can use `_int_install_isr()` to install an ISR for SWI6 and service the NMI. Three API function calls allow the user to determine the source(s) of the NMI(s) and clear them.

- `_get_nmi_source()` returns the source(s) of the NMI interrupt. These predefined mask values are used to mask out the desired interrupt: `INT_NMI_BIT_MASK`, `FRAMEERR_NMI_BIT_MASK`, and `SW7_NMI_BIT_MASK`.
- `_clear_nmi_source()` clears a specific NMI source. Only one interrupt source can be cleared at a time. This function accepts these parameters to clear the NMI interrupt source: `INT_NMI_BIT`, `FRAMEERR_NMI_BIT`, and `SW7_NMI_BIT`.
- `_clear_all_nmi_source()` clears all NMI sources. This API is typically called during initialization.

A small amount of latency is introduced with the redirection of the NMI to the SWI6 interrupt. This measured latency is 14.5 µs.[1] Based on this measurement, the system is theoretically unable to service NMIs faster than the rate of 1/14.5 µs or 68.97 kHz.

1. Measurement conducted using Freescale MQX™ RTOS v3.7 and CodeWarrior 10.2 with FXLC95000 bus clock running at 16 MHz.

3.2.3 Freescale MQX™ RTOS Task Preemption Control Considerations

Freescale MQX™ RTOS provides two functions for controlling preemption. These functions allow a task to control when it may be preempted by the scheduler.

- `_task_start_preemption()`
- `_task_stop_preemption()`

The behavior of these two functions is worth noting.

According to the Freescale MQX™ RTOS scheduler, preemption only occurs when a running task is removed involuntarily from the run queue due to a higher priority task. A task may also give up the CPU voluntarily, for example, by blocking or using task synchronization mechanisms such as events, semaphores, and mutexes.

This means that even when preemption has been disabled by a task (by calling `_task_stop_preemption()`), the task may give up the CPU by setting events, posting semaphores, or otherwise *explicitly* causing a higher priority task to become ready-to-run.

Put another way, calling `task_stop_preemption()` will only guarantee that actions taken *in an interrupt service routine* will not cause the currently executing task to be swapped out until preemption is re-enabled, but calls to `_lwevent_set()`, `lwsem_post()`, etc. may still cause the task to be swapped out in favor of a higher-priority task waiting for such an event.

3.3 Communications

The ISF Communications service family enables data passing between the host and the intelligent sensing platform. This data can include commands, status, results, and sensor data. Abstractions for communicating between the host and user embedded applications via the Slave-Port Mailboxes are provided in the Command Interpreter component of the Communications service family. In addition, ISF provides abstractions for communicating with external I^2C devices via the Device Messaging and Master I^2C components.

3.3.1 ISF Command Interpreter

The ISF Command Interpreter (CI) handles the dispatching of commands received via the Slave-Port Mailboxes. The CI provides a general mechanism to take commands and data read from the mailboxes and trigger the execution of callback functions registered by the application that handles that command.

The CI processes commands in the order they are received. Commands can be built-in or user-registered. Built-in commands are part of the ISF core functionality and cannot be removed or modified by the user. User-registered commands can be tied to an application and can register with the CI at run time by adding a command to the `ci_callback` array.

3.3.2 Mailbox Interface

The FXLC95000CL hardware provides a mailbox abstraction on top of the slave-port serial interface. The term slave-port is used to refer to the FXLC95000CL's slave interface to the host processor, which may be configured to use either the I^2C or SPI protocol. The mailbox abstraction consists of 32 mailboxes, each an addressable register that is 8 bits wide. For more information about the mailbox abstraction, refer to the FXLC95000CL Hardware Reference Manual listed in References. The mailboxes are used in turn to hold both incoming and outgoing data. That is, data sent by the bus master is placed in these mailboxes by the slave-port peripheral hardware for use by the firmware. Data written to the mailboxes by the firmware is used by the slave-port peripheral hardware to satisfy read requests from the bus master.

The ISF Command Interpreter component provides an interface to the mailboxes that embedded applications may use when communicating with the host processor. The Command Interpreter interface allows an embedded application to interact with the host processor using two different paradigms.

The **Command/Response** paradigm is a synchronous interface paradigm where the bus master initiates an exchange by writing a command to the Slave-Port Mailboxes and the firmware responds by placing its response data in the mailboxes. The bus master then sends a second command to read the data from the mailboxes. A Command Complete (COCO) bit in MB1 is used as a semaphore for when a command is sent to the Command Interpreter and when a complete response put into the mailboxes by the Command Interpreter may be read coherently.

The **Quick-Read** paradigm is an asynchronous interface paradigm where the firmware keeps specific mailboxes filled with current data by updating the mailbox contents whenever new data becomes available. This allows the host to send a single read command at any time and have the data returned immediately. This Quick-Read paradigm is most useful for making periodic output data from the embedded application available to the host.

These two paradigms may be used separately or in combination depending on the configuration of the mailboxes. By default, the mailboxes are all configured to support Command/Response interactions, but the Command Interpreter provides the capability to reconfigure some of the mailboxes to hold Quick-Read data. Characteristics of each

paradigm are presented next and example applications are supplied with FXLC95000 (documented in the Application Developer's User Guide listed in References) that may be referred to for additional guidance.

3.3.3 The Command/Response Paradigm

The Command Interpreter (CI) implements the Command/Response mode using a callback design pattern. The CI is notified whenever the bus master writes to the mailboxes. It then reads the mailboxes, and uses the contents to identify the correct recipient for the command written by the bus master and invokes that recipient's registered callback function. The CI writes the callback return status, response data if generated, and a Command Complete (COCO) indication to the mailboxes. When the CI sets the COCO bit to 1, the mailboxes contain the complete response, which may now be read coherently. The bus master may choose to poll for this bit or may elect to configure the device behavior to have an external interrupt sent to the master when the complete response has been written, provided the interrupt pin has been appropriately wired. For more information about proper wiring of the interrupt pin, refer to the FXLC95000CL Hardware Reference Manual listed in References.

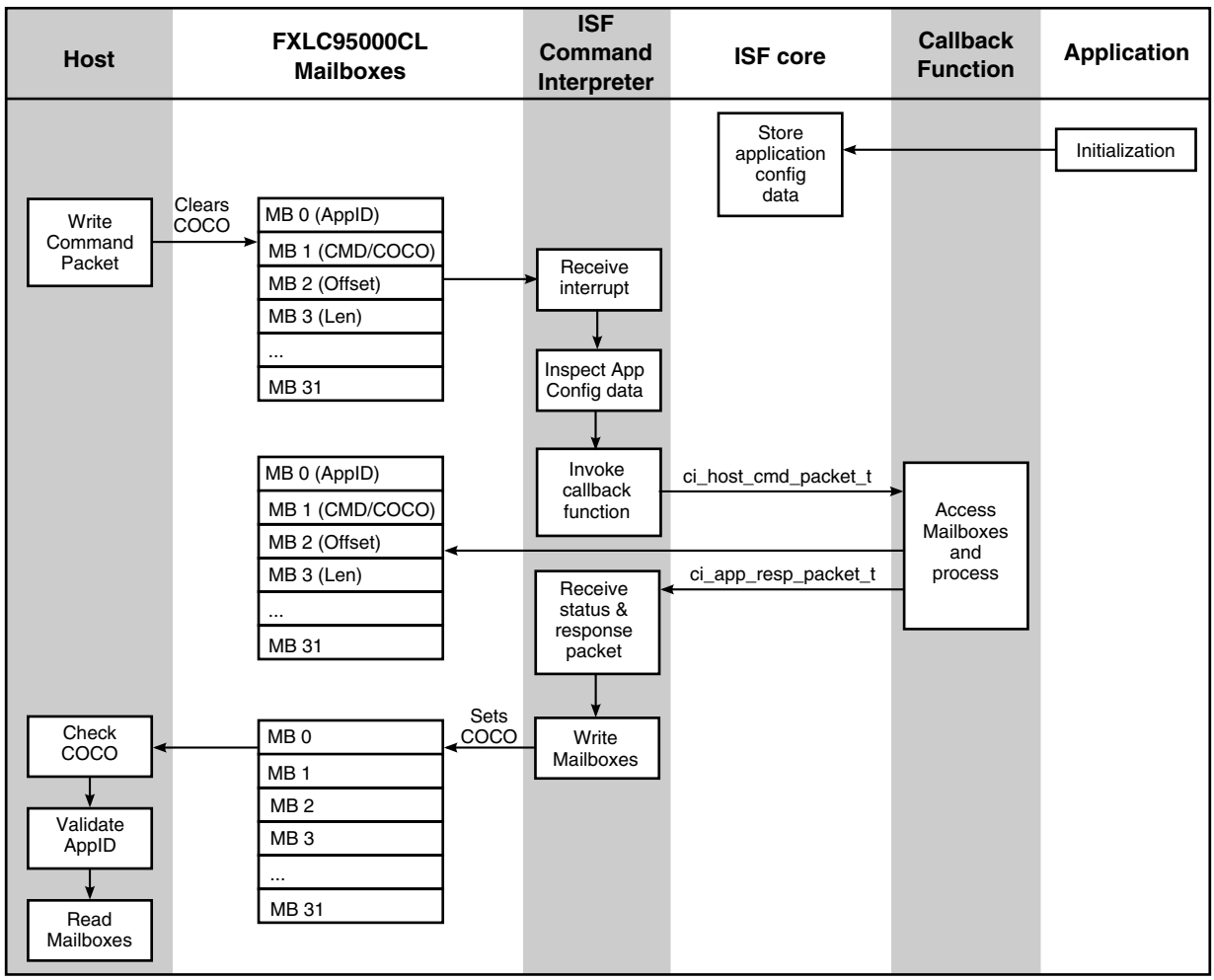

Figure 3-2. The Command/Response paradigm

3.3.3.1 Command Response Mailbox Layout

To support Command/Response interactions, the Command Interpreter (CI) makes use of the 32 8-bit mailbox registers as shown in Table 3-5.

Table 3-5. Mailbox Usage for Command/Response

Mailbox	Mailbox Use
MB0	Application callback ID used by the CI to determine which registered callback to invoke.
MB1	Bit[6:0] contains the command to be executed. Commands are defined by applications. Bit[7] is reserved for the command complete (COCO) status. The host writes a zero to this bit when sending the command. After writing the command packet, the host may then poll this bit. The bit is set by the CI when the callback completes and the response data has been written to the mailboxes.
MB2	Byte offset of the data requested by the host.

Table continues on the next page...

Table 3-5. Mailbox Usage for Command/Response (continued)

Mailbox	Mailbox Use
	To access only the data of interest, the host can specify an offset into the data block. The offset range is from 0 to 255.
MB3	Requested number of data bytes to be read or written depending on the command.
MB4 - MB31	General data transfer. **NOTE:** Any mailbox from MB4 to MB31 may be configured by the host for Quick-Read data. If a mailbox is configured for Quick-Read data, it is not available for use by the CI for Command/Response data transfer.

3.3.3.2 Command Processing

To understand the Command/Response mailboxes and their use, it is helpful to think of each embedded application running on the device as having two logical buffers, one for input and one for output data. Structurally, the buffers can be thought of as having a fixed layout such that the value at a specific location within a buffer always contains the same type of data.

Each application can allocate its own input buffer. The host can send data to a particular target application by writing data into specific locations within that application's input buffer. The locations are specified as an offset into the buffer along with the number of bytes to write followed by the actual data values. To do this, the host writes the data to the FXLC95000's mailboxes in the following format:

The application ID of the target application is written in MB0,
the command type is written in MB1,
the offset into the target application's buffer is written to MB2,
the number of bytes to write is written in MB3, and
the actual data values to write are then written starting with MB4.

As an example, to send a series of four bytes, containing values of 5, 6, 7, and 8 respectively, to the application with application callback ID 5 and place them at, for instance, offset 2 in the application's input buffer, the host would send an 8-byte command sequence as follows:

MB0	MB1	MB2	MB3	MB4	MB5	MB6	MB7
Application Callback ID	Command Number	Offset	#Bytes	data	data	data	data
5	CI_CMD_WRITE_CONFIG	2	4	5	6	7	8

The following command types are defined for use in isf_ci.h:

```
typedef enum
{
  CI_CMD_READ_VERSION     = 0,  // Requset to read version information
  CI_CMD_READ_CONFIG      = 1,  // Request to read an app's configuration data
  CI_CMD_WRITE_CONFIG     = 2,  // Request to write an app's configuration data
  CI_CMD_READ_APP_DATA    = 3,  // Request to read from an app's output buffer
  CI_CMD_UPDATE_QUICKREAD = 4,  // Request an app to update its Quick-Read data
  CI_CMD_READ_APP_STATUS  = 5,  // Request to read an application's status
  CI_CMD_MAX              = 128 // Maximum number of possible CI commands
} ci_commands_enum;
```

Each user-defined application is allowed to define its own commands in the range between `(CI_CMD_READ_APP_STATUS + 1)` and `CI_CMD_MAX`. For example, an application may define the following new command:

```
#define CI_CMD_APP_LOCAL_1 (CI_CMD_READ_APP_STATUS + 1)
```

Then, within the application's CI callback, the following code example could be used to handle the new command:

```
// Embedded Application defined command
case CI_CMD_APP_LOCAL_1:
    // Local command processing
    if (ERROR == local_cmd_processing(pHostPacket))
    {
        callbackRet = CI_ERROR_COMMAND;
    }
    break;
```

Similar to the way that applications may use the mailboxes as input buffers, each embedded application can utilize mailboxes as a logical buffer of outputs. Output data within this buffer can be thought of as having a fixed layout such that the value at a specific location always contains the same type of data.

CAUTION

While it is possible to send data from the host to the FXLC95000's mailboxes with a starting mailbox offset other than zero, e.g., writing data to MB4 through MB9 without writing to MB0 through MB3, the Command Interpreter expects MB0 through MB3 to be written for each Command/ Response message. Sending a command without writing MB0-MB3 will result in undefined behavior.

For example, an application may produce an X, Y, Z output vector where each vector component is a 16-bit value. The application may place these values in its output buffer as: X at bytes 12 and 13, Y at bytes 14 and 15, and Z at bytes 16 and 17. To read these values, the host asks the application for the 6 bytes of data located at an offset of 12 in its buffer. An application callback ID is used to identify the application whose data is to be retrieved. The Command Interpreter (CI) uses the application callback ID as an index into

its table of registered callbacks to determine the callback to invoke. Each application that exposes data for the host to read has a callback registered with the CI. Each callback needs to be able to read and write its associated application's data buffer. Because a callback is closely associated with an application, the application callback ID can also be thought of as an application ID, although not all applications will have an associated application callback ID.

Note

The terms application callback ID, application ID, and callback ID are used interchangeably throughout this reference manual.

When the callback function completes, it returns the status of the command execution along with a response packet to the CI. The CI then writes the response status and packet to the mailboxes and sets the command complete (COCO) bit to notify the host that the command has been processed. Once the host sees the COCO bit set, it can safely read the mailboxes to obtain the status and the data from the application.

Continuing the previous example, in order to request 6 bytes at offset 12 from an application with application callback ID 5, the host would send a 4-byte command sequence as follows:

MB0	MB1	MB2	MB3
Application Callback ID	**Command Number**	**Offset**	**#Bytes**
5	CI_CMD_READ_APP_DATA	12	6

Note

The `CI_CMD_READ_APP_DATA` command enum should be used when requesting to read from an application's output data buffer.

Several command numbers have been defined for general use and are shown in ci_commands_enum.

3.3.4 The Quick-Read Paradigm

An application may have data that the host wants to read on a regular basis without sending a command packet and waiting for a response. To reduce the latency of data access, the application can make its output data available through Quick-Read mailboxes.

Figure 3-3 shows how an application could make data available to the host on a regular basis via Quick-Read in a streaming mode. The application can set up Quick-Read such that a callback is triggered every time a mailbox is read. The callback function can then put new data into the Quick-Read registers and the whole process repeats.

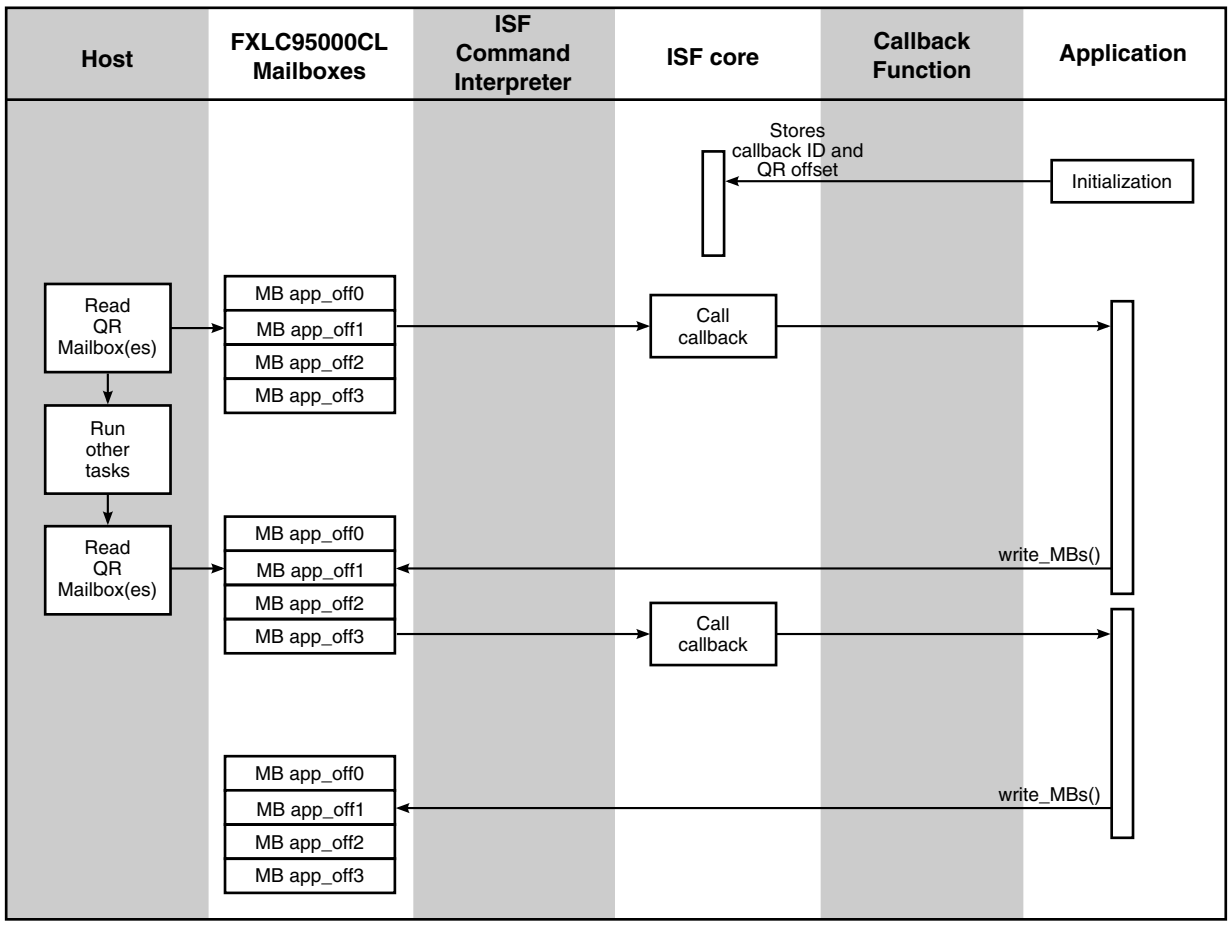

Figure 3-3. Quick-Read paradigm—streaming mode

Figure 3-4 shows how an application could read Quick-Read mailboxes in an asynchronous mode without having to set up a callback.

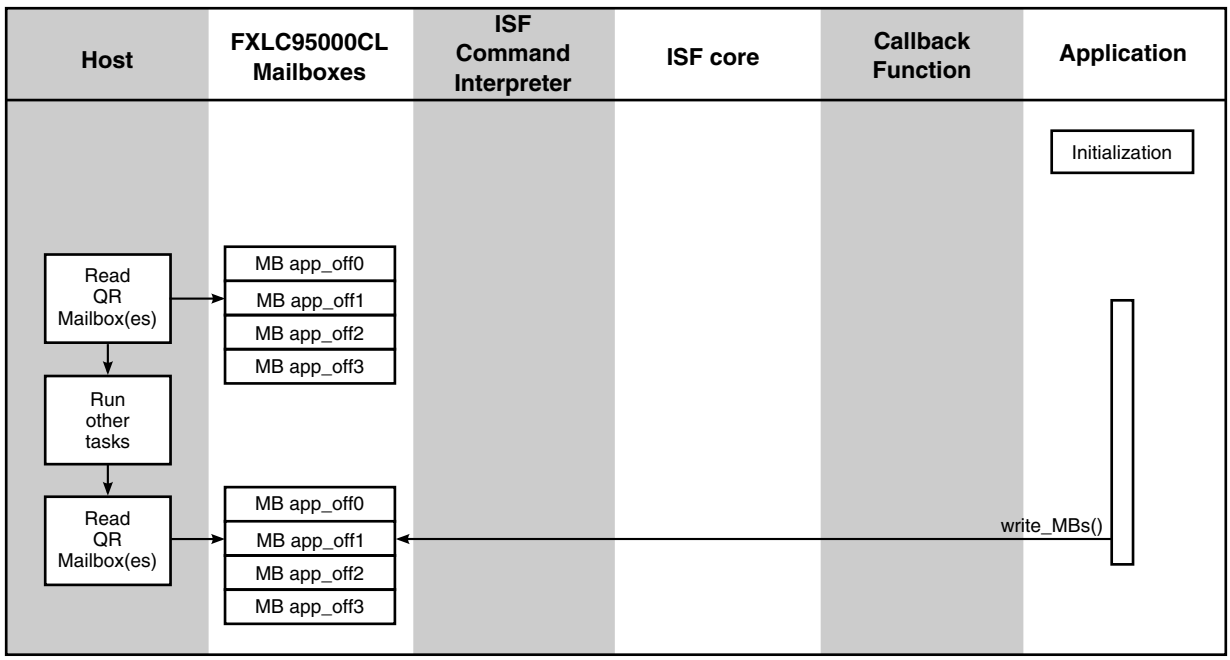

Figure 3-4. Quick-Read paradigm—asynchronous mode

3.3.4.1 Quick-Read Mailbox Layout

To support Quick-Read interactions, the Command Interpreter (CI) allows 28 mailboxes (MB4 - MB31) to be configured to hold Quick-Read data as shown in Table 3-6.

Table 3-6. Mailbox Usage for Quick-Read

Mailbox	Mailbox Use
MB0 - MB3	Dedicated for Command/Response transfers.
MB4 - MB31	Mailboxes in this range can be configured by the host for Quick-Read data, individually or in groups. If a mailbox is configured for Quick-Read data, the Command Interpreter will not use it for Command/Response data transfer, and the host should not write to these mailboxes. Quick- Read mailboxes are written by the application and read by the host. **Note:** If not configured for Quick-Read, a mailbox will remain allocated as a Command/Response mailbox. **Note:** Once configured, a Quick-Read mailbox should be used exclusively by the application which maps an output to that mailbox until the application no longer needs to send the data. The QR mailboxes can be reconfigured by another QR configuration command from the host.

When a mailbox is configured for Quick-Read by the host application, a callback id and offset are assigned to each Quick-Read mailbox. Each time a Quick-Read mailbox is read, the CI invokes the associated callback to allow it to place new data in the Quick-Read mailbox. An embedded application may also update its Quick-Read mailboxes anytime it has new data to be written to the Quick-Read registers by calling `isf_ci_qr_update()`.

The mapping of data from an application's output buffer to Quick-Read mailbox registers is specified by the host and managed by the CI. This abstracts the specific mailbox configuration from the application.

Figure 3-5 illustrates three different applications and how they might be configured to map different outputs to different mailboxes for Quick-Read.

- The application with AppId 3 maps its 3rd output to MB19.
- The application with AppId 4 maps its 1st output to MB13, its 3rd and 4th outputs to two contiguous mailboxes, MB20 and MB21 respectively.
- The application with AppId 8 maps its (non-contiguous) 11th and 13th outputs to two contiguous mailboxes, MB30 and MB31 respectively.

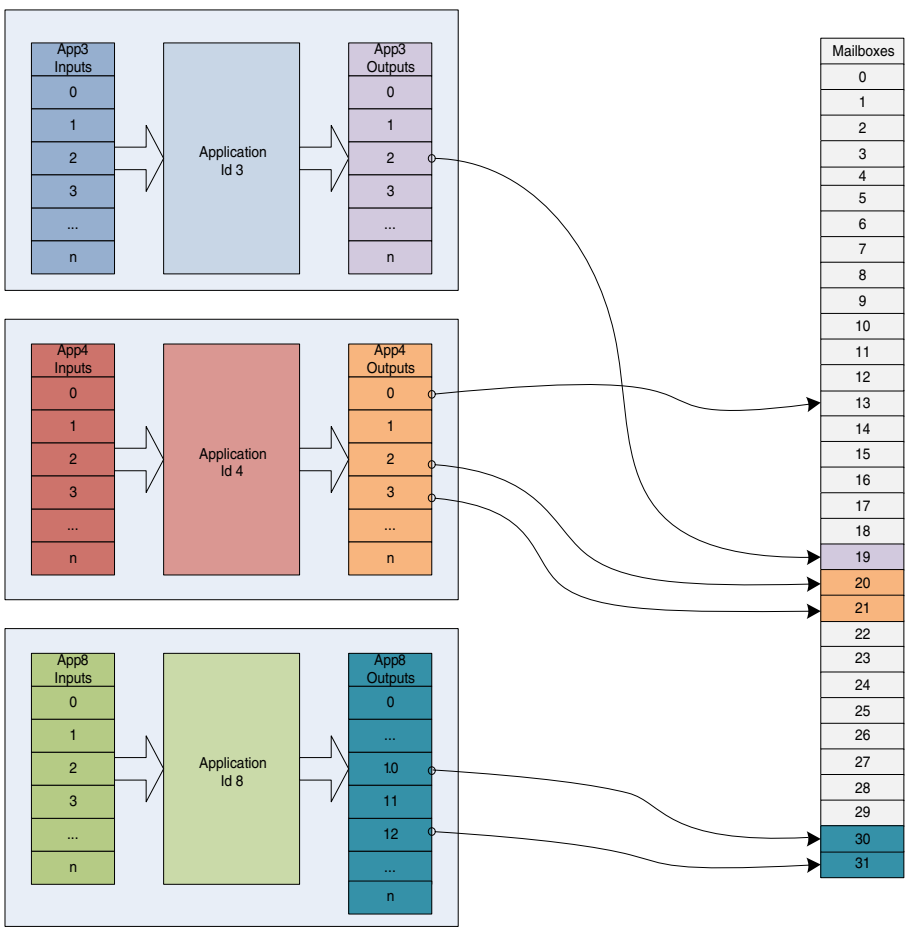

Figure 3-5. Mapping Application Output to the Quick-Read Mailboxes

3.3.4.2 Configuring the Quick-Read Mailboxes

To configure Quick-Read (QR) mailboxes, the configuration data is placed in the Command Interpreter(CI)'s own input buffer.The CI makes its input buffer available to the host in the same manner as described in Command Processing. To write to the CI's

input buffer, a CI_CMD_WRITE_CONFIG command is sent to the CI's application ID (ISF_APP_ID_MBOX). For each mailbox configured for QR, the CI tracks which application is associated with that mailbox and which byte from the respective application's output buffer is to be copied into it. The CI's input buffer is laid out as shown in Table 3-7.

Table 3-7. Command Interpreter Input Buffer Layout

Command Interpreter Input Buffer Byte Offset	Parameter	Command Interpreter Input Buffer Byte Offset	Parameter
0	QR MB 4: APP_ID	28	QR MB 18: APP_ID
1	QR MB 4: OFFSET	29	QR MB 18: OFFSET
2	QR MB 5: APP_ID	30	QR MB 19: APP_ID
3	QR MB 5: OFFSET	31	QR MB 19: OFFSET
4	QR MB 6: APP_ID	32	QR MB 20: APP_ID
5	QR MB 6: OFFSET	33	QR MB 20: OFFSET
6	QR MB 7: APP_ID	34	QR MB 21: APP_ID
7	QR MB 7: OFFSET	35	QR MB 21: OFFSET
8	QR MB 8: APP_ID	36	QR MB 22: APP_ID
9	QR MB 8: OFFSET	37	QR MB 22: OFFSET
10	QR MB 9: APP_ID	38	QR MB 23: APP_ID
11	QR MB 9: OFFSET	39	QR MB 23: OFFSET
12	QR MB 10: APP_ID	40	QR MB 24: APP_ID
13	QR MB 10: OFFSET	41	QR MB 24: OFFSET
14	QR MB 11: APP_ID	42	QR MB 25: APP_ID
15	QR MB 11: OFFSET	43	QR MB 25: OFFSET
16	QR MB 12: APP_IDQR	44	QR MB 26: APP_ID
17	QR MB 12: OFFSET	45	QR MB 26: OFFSET
18	QR MB 13: APP_ID	46	QR MB 27: APP_ID
19	QR MB 13: OFFSET	47	QR MB 27: OFFSET
20	QR MB 14: APP_ID	48	QR MB 28: APP_ID
21	QR MB 14: OFFSET	49	QR MB 28: OFFSET
22	QR MB 15: APP_ID	50	QR MB 29: APP_ID
23	QR MB 15: OFFSET	51	QR MB 29: OFFSET
24	QR MB 16: APP_ID	52	QR MB 30: APP_ID
25	QR MB 16: OFFSET	53	QR MB 30: OFFSET
26	QR MB 17: APP_ID	54	QR MB 31: APP_ID
27	QR MB 17: OFFSET	55	QR MB 31: OFFSET

The CI contains a Mailbox Application (ISF_APP_ID_MBOX) which configures the Mailboxes as its input buffer as illustrated in Figure 3-6.

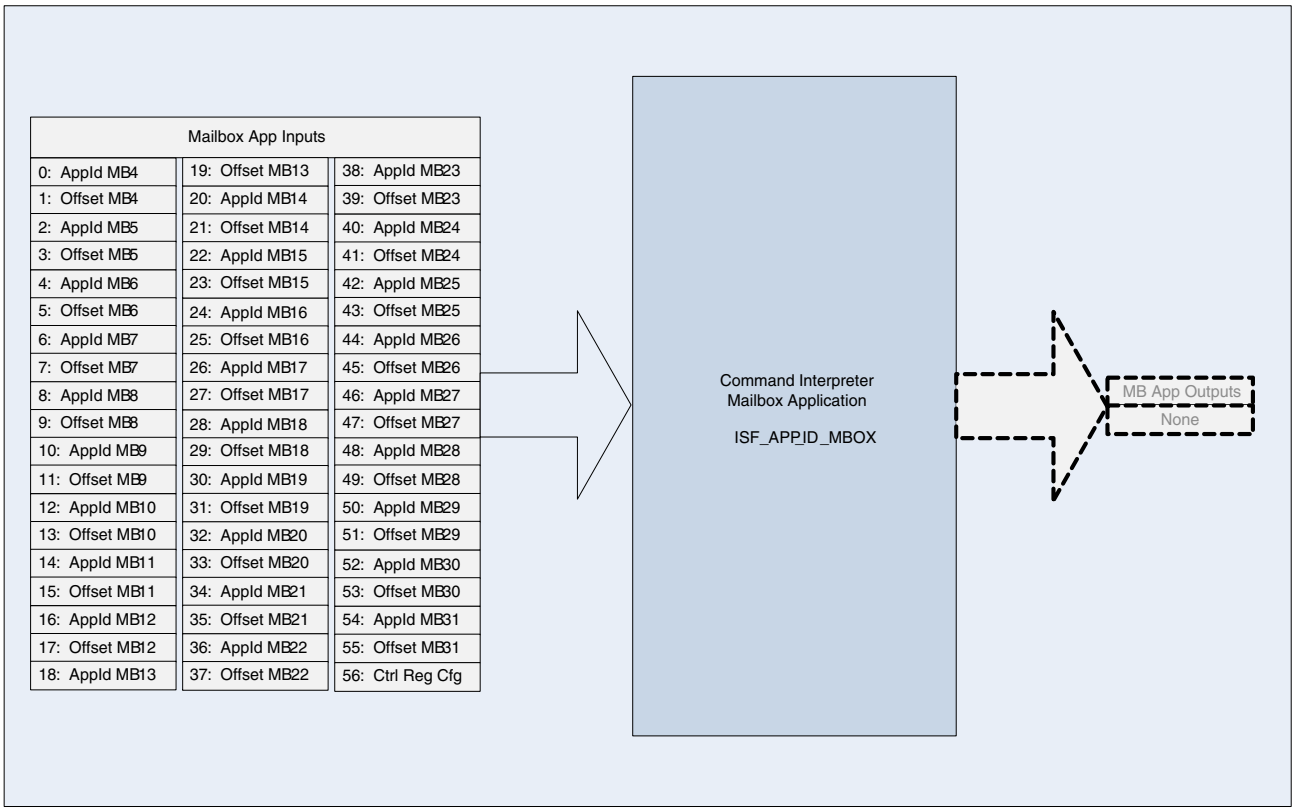

Figure 3-6. Command Interpreter Mailbox Application

As found in Table 3-7, the APP_ID specifies the source of the Quick-Read data and the OFFSET specifies the offset into that application's output data buffer when retrieving Quick-Read data. Because there are 28 data mailboxes available for Quick-Read (MB4 - MB31), the CI's Quick-Read configuration data totals 56 bytes with 2 bytes of configuration data per mailbox.

The host can assemble a command to configure a Quick Read register by referring to the data in Table 3-7. The configuration command itself is a Command/Response type command sent to the CI shown in Table 3-8. The CI application ID is preconfigured through the `ISF_APP_ID_MBOX` enumeration value defined in *isf.h*.

Table 3-8. Quick-Read Configuration Command Packet

Mailbox	Quick-Read Configuration Command Packet	Description
MB0	`ISF_APP_ID_MBOX`	Application ID (predefined). ISF has predefined applications, one being `ISF_APP_ID_MBOX`.
MB1	`CI_CMD_WRITE_CONFIG`	Command to write to the configuration data.
MB2	offset	Offset into the 56 byte configuration data. Table 3-7 shows the offsets for the application ID and Quick-Read offset for each mailbox of the Quick-Read configuration data.
MB3	length (# bytes)	Number of configuration data bytes.

Table continues on the next page...

Table 3-8. Quick-Read Configuration Command Packet (continued)

Mailbox	Quick-Read Configuration Command Packet	Description
MB4-MBn	Configuration data bytes	For each Quick Read Mailbox assignment, two consecutive bytes where the first byte is the application ID and the second byte is the offset of the application's Quick-Read dataset.

Figure 3-7 shows an example contents of the CI's Mailbox Application's (`ISF_APP_ID_MBOX`) input buffer. AppId 0x03 is configured with MB19 as offset 0x02 , AppId 0x04 is configured with MB20 as offset 0x02 and MB21 as offset 0x03, AppId 0x08 is configured with MB30 as offset 0x0A and MB31 as offset 0x0B.

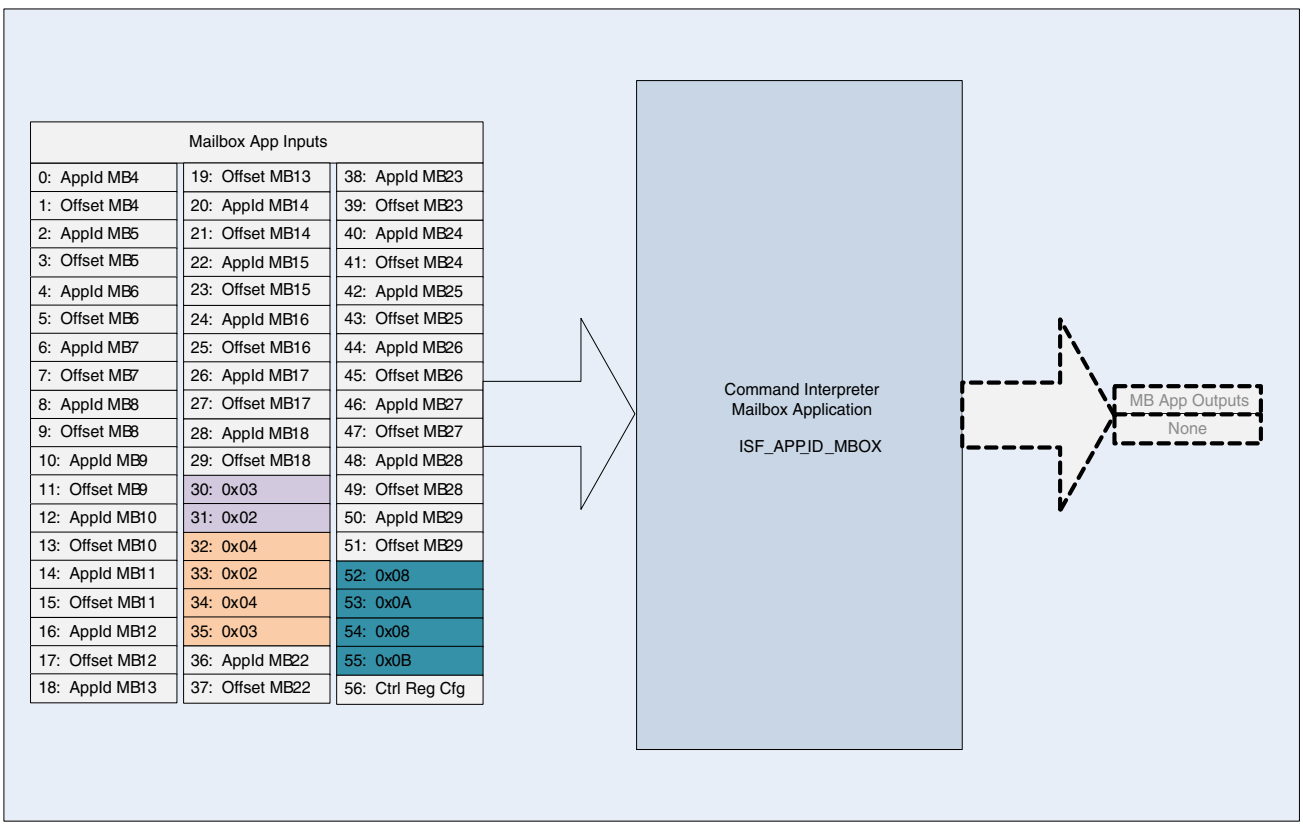

Figure 3-7. Command Interpreter Mailbox Application

The following is an example host command to configure MB30 - MB31 for Quick-Read taking data from application ID = 8 offsets 10 and 11 with the data from offset 10 being placed in MB30 and offset 11 being placed in MB31.

Table 3-9. Quick-Read Configuration Command Packet Example

Mailbox	Value	Description
MB0	`ISF_APP_ID_MBOX`	Application ID (predefined)

Table continues on the next page...

Table 3-9. Quick-Read Configuration Command Packet Example (continued)

Mailbox	Value	Description
MB1	CI_CMD_WRITE_CONFIG	Command to write to the configuration data
MB2	52	Offset into Quick-Read config data to access the application ID of MB30
MB3	4	4 bytes total—2 bytes for MB30 and 2 bytes for MB31
MB4	8	MB30: application ID
MB5	10	MB30: offset
MB6	8	MB31: application ID
MB7	11	MB31: offset

When any byte in the 32 byte mailbox registers is read, the FXLC95000CL hardware caches the entire 4-byte region in which that byte resides in a 4-byte, line buffer. Reads of subsequent bytes are done from the buffer, ensuring that consistent data is present in multiple-byte variables. Whenever feasible, users should consider configuring Quick-Read mailboxes as groups of four aligned to a mod 4 address boundary.

The host should track which mailboxes it has configured for Quick-Read and how many data mailboxes remain for general data transfer. For example, if the host configures MB26 - MB31 (6 mailboxes) for Quick-Read, then 22 mailboxes remain for general Command/Response transfer (28 – 6 = 22). In this case, if the host sends a command to read or write data, it must set the requested byte count to less than or equal to 22. Otherwise, unpredictable results may occur.

The host should also keep the Quick-Read mailboxes contiguous for efficient use of mailbox resources. The CI Command/Response interactions can only make use of the data mailboxes that are contiguous starting at MB4. For example, if the host configures MB20 - MB23 (4 mailboxes) for Quick-Read, two noncontiguous sets of data mailboxes remain, MB4 - MB19 and MB24 - MB31. In this case, when the CI receives a host command to read or write data, it will only be able to use MB4 - MB19 for Command/Response data. It will not use MB24 - MB31.

Note

The recommendation is for the host to configure Quick-Read mailboxes at the end of the mailbox range with no gaps between Quick-Read mailboxes. Configuring the Quick- Read data in this manner leaves the largest contiguous set of data mailboxes starting at MB4 available for Command/Response data transfer.

The application is provided an opportunity to update the Quick-Read mailboxes after they are read by the host. When the host completes reading Quick-Read data, the CI invokes the associated application's callback function with the command `CI_CMD_UPDATE_QUICKREAD`. The application can then call `isf_ci_qr_update()` to update the Quick-Read mailboxes.

When an application uses `isf_ci_qr_update()`, it is not required to specify the data mailboxes to be updated. The `isf_ci_qr_update()` call only requires an application ID and a pointer to the data buffer from which the new data is obtained. The CI tracks which mailboxes are designated as Quick-Read. With the Quick-Read configuration data, the CI updates the appropriate mailboxes from the supplied buffer.

Note

The CI requests the application to update its Quick-Read data when at least one Quick-Read data for that application has been read. The host does not need to read all of the Quick-Read data in order for the CI to request that the application update its Quick-Read data. For example, consider the case where the host has configured MB24 - MB31 (8 mailboxes) for Quick-Read data from an application and then sends a command to read only MB30 - MB31. This read will still cause the CI to invoke the associated callback function with a `CI_CMD_UPDATE_QUICKREAD` command.

Note

The host is able to subscribe to just a portion of the Quick-Read data that an application offers. For a particular application, the host may subscribe to a subset of the data by configuring only the number of Quick-Read mailboxes needed with the desired offset into the Quick-Read data.

3.3.4.3 Mailbox Application Control Config Register

The Mailbox Application (Mailbox App) contains the Ctrl Cfg register that allows the user to configure the FXLC9500 INT_O pin functionality and the Quick-Read streaming mode. The INT_O pin can be configured as an interrupt output allowing it to be used by the Mailbox App to interrupt the host when new data is available in the mailboxes. Using an interrupt is often more efficient than having the host poll the mailboxes for new data. By default, the host interrupt functionality is disabled. An application can enable the host interrupt function by setting bit 0 of the Mailbox App Ctrl Cfg register to a value of 1. When enabled, the FXLC95000 device generates an interrupt to the host using the INT_O pin when any of the mailboxes are written by the application. Note that the INT_O pin is configured by default as a GPIO pin after reset, but the Mailbox App reconfigures the pin

muxing automatically based on its Ctrl Cfg register settings. If the host enables the INT_O feature, the Mailbox App reconfigures the pin muxing from GPIO to the INT_O function. If the host disables the INT_O feature, the Mailbox App reconfigures the pin muxing back to its GPIO function.

The Quick-Read streaming mode allows applications to request Quick-Read data to be updated in the mailboxes. By default, streaming mode is disabled and any application update to the Quick-Read mailboxes is ignored and the mailbox contents remains the same. The application can enable the Quick-Read streaming mode by setting bit 2 of the Mailbox App Ctrl Cfg register to a value of 1. When enabled, the application calls `isf_ci_qr_update()` to update the Quick-Read mailboxes with new data.The Mailbox App Ctrl Cfg register can be accessed via the Mailbox App's Input Buffer byte offset 56 (see Figure 3-6). The definition of the register is as follows:

Bit	Name and function
0	Pin INT_O enable
	0 - Disable pin INT_O from generating interrupt to the host 1 - Enable pin INT_O to generate interrupt to the host
1	Pin INT_O polarity
	0 - INT_O is active low 1 - INT_O is active high
2	Streaming mode
	0 - Streaming mode is disabled 1 - Streaming mode is enabled

3.3.5 Device Messaging

The Device Messaging component exposes the user-level APIs for communicating with external slave devices. The goal of Device Messaging is to abstract the communications protocol so it provides a unified interface for communications regardless of the underlying transport method being used.

The Device Messaging component is optional in an ISF deployment and is provided as a linkable library. Because the ISF functionality is linked with the embedded application code in a single executable image, this library is shared between all embedded application tasks and ISF tasks that use it.

Figure 3-8 depicts the architecture of the Device Messaging component. The Device Messaging abstraction depends on the individual protocol libraries used. The protocol libraries each install their own peripheral interrupt handlers with the RTOS kernel when they are initialized. ISF v1.1 supports the I^{2}C protocol. SPI and GPIO will be added in a future release.

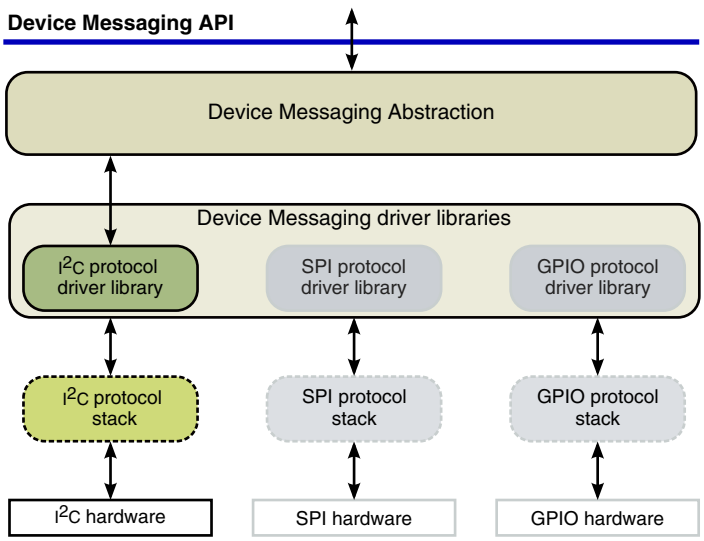

Figure 3-8. Architecture of the Device Messaging component

3.3.5.1 Device Messaging concepts

The Device Messaging component provides a high-level, abstraction layer on top of the communications protocols supported by the ISF. This allows applications as well as other ISF component instantiations included in the firmware to communicate with external devices in the same way regardless of how that device is physically connected.

The Device Messaging interface is loosely modeled after the POSIX file I/O interfaces. A Device Messaging `deviceHandle` behaves similarly to a file descriptor. In order to communicate with an external device, the device must be opened with a `dm_device_open()` call which returns a deviceHandle. The deviceHandle is then passed to the `dm_read()` or `dm_write()` functions to designate the desired communications endpoint.

The Device Messaging component depends on the individual protocol implementations such as I^{2}C to perform the actual communications. The late-binding mechanisms enabled by the Device Messaging implementation and employed by the Sensor Manager allow the Sensor Manager to be protocol independent. The Sensor Manager can manage sensors using any communications transport protocol. Users can define transport protocols other than those provided by FXLC95000 without needing to recompile the ISF core library.

Figure 3-9 shows the relationship between Device Messaging and an underlying transport protocol, in this case I²C, and two embedded applications each talking to an external I²C device, one through the Device Messaging and one using the I²C component directly. The example shows that a magnetometer is present at I²C device address 0x0E.

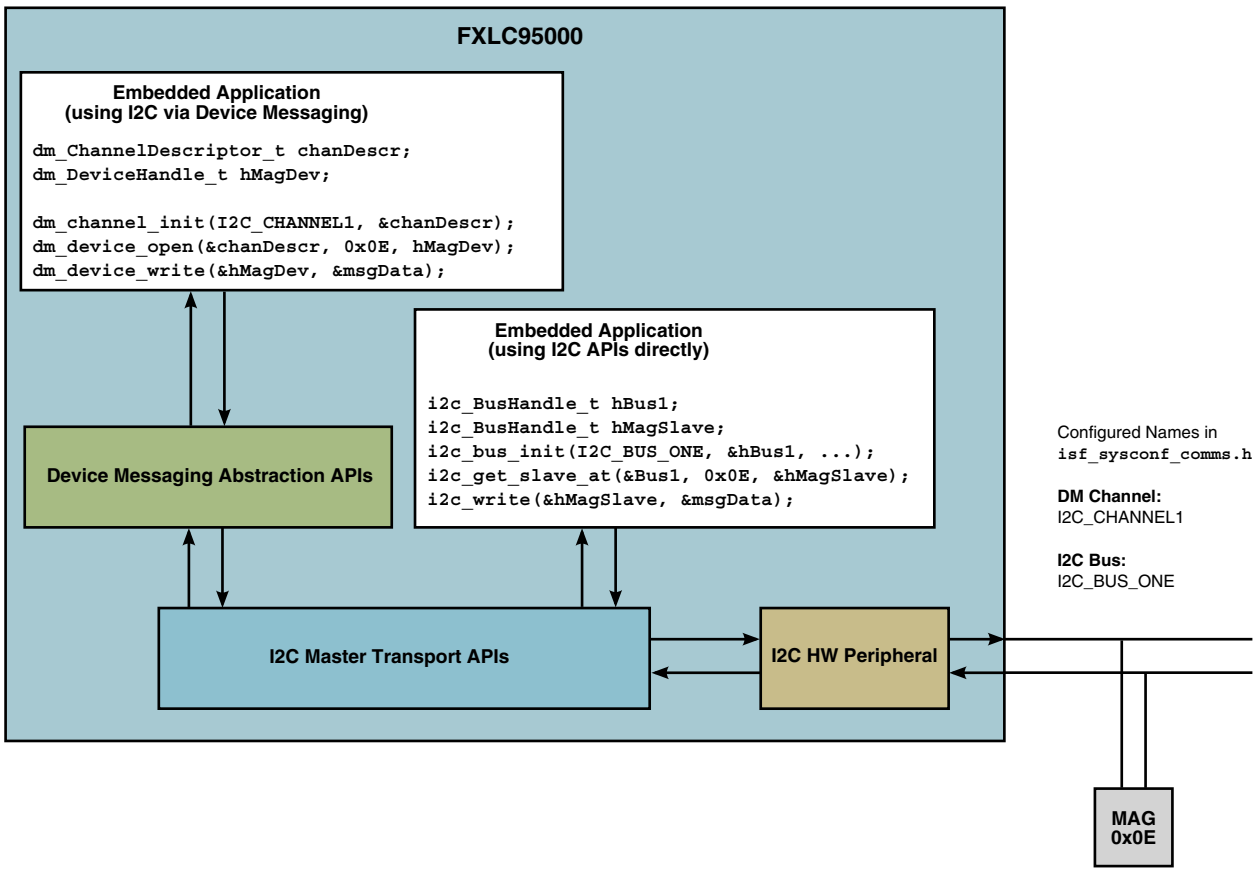

Figure 3-9. Device Messaging abstracts underlying transport protocols

3.3.5.1.1 Channels and Devices

The object types used by Device Messaging are channels and devices. These objects encapsulate the object types used by the underlying transport protocol in order to provide a unified Device Messaging interface. For example, when using the ISF I²C transport protocol, a bus object identifies which one of potentially several different I²C peripherals are used when talking to a particular external I²C slave.

Using the Device Messaging interfaces, a Device Messaging channel object wraps the I²C bus object and a Device Messaging device object wraps the I²C slave object. A global array of the available device messaging channels is declared as part of the ISF system configuration. System configuration files are provided in source format to allow developers to modify the files for their own slave devices.

To use a particular channel for communications, the channel must first be initialized. When a channel is initialized, a channel descriptor is returned to the calling function. The calling function uses this channel descriptor in subsequent Device Messaging calls to invoke operations on the channel. In a similar manner, to communicate with a particular external device on a channel, the device must first be opened for communications. When a device is opened, a DeviceHandle is returned and used in subsequent calls to communicate with that device.

3.3.5.1.2 Channel Locking

An explicit channel locking capability allows extended exclusive access to a channel. When a channel lock is held, no other task may communicate to any devices on the channel until the lock is released. Calls to device operations without first acquiring an explicit channel lock cause an implicit channel lock to be acquired but only for the duration of that call. Channel locks are implemented with priority inversion protection using a priority inheritance scheme that automatically raises the current lock holder's priority to the priority of the highest waiting task until the lock is released.

3.3.5.1.3 Device Handle

The Device Messaging component uses a logical function abstraction table to interact with multiple transport protocols transparently. The Device Messaging APIs operate on device handles. Each device handle contains a reference to an internal channel structure used to communicate with the device. The Device Messaging component, through the channel reference, determines the protocol used to communicate with the device.

The DeviceMessaging APIs cover channel operations including initialization, locking, reconfiguration, status query and control, as well as device operations including open/close, read/write. For details of these APIs and examples of their use, refer to the Xtrinsic ISF API Reference Manual for the FXLC95000 Intelligent Motion-Sensing Platform listed in References.

3.3.5.2 Usage Example

In the following example, communication with an I^2C slave device using Device Messaging is demonstrated.

```
#include <isf_devmsg.h>
#include <isf_sysconf_comms.h>
```

```
typedef COMM_SIZED_BUFFER(18) ReadBuffer18_t;

const comm_Address_t  MAG3110_ADDR  = 0x0E;     // I2C Bus address of MAG3110

// A command to initialize the magnetometer
static const comm_Command_t mag3110WrReg = {2, 0, {0x10, 0x00}};

// A command to read magnetometer data
static const comm_Command_t mag3110RdReg = { 1, 1, {0x01}};

void get_mag_data()
{
  // Setup a buffer to hold data from the magnetometer
    ReadBuffer18_t         readBuffer  = { .size=18 };
    comm_MessageBuffer_t  *pReadBuf    = (comm_MessageBuffer_t *)&readBuffer;

  isf_status_t          status;
  isf_duration_t        timeout = 0; // zero will mean 'wait forever'
  dm_ChannelDescriptor_t chanDesc;        // holds the returned channel descriptor
  dm_DeviceHandle_t      magDeviceHandle; // holds the returned device handle

  // Init creates and initializes data structures required to manage the channel.
  // The channel Id enumeration comes from sysconf_comms.h::sys_channelId_t and
  // identifies the I2C channel to use
  status = dm_channel_init(I2C_MASTER1, &chanDesc);

  // Start Bus (Enables Hardware Peripheral associated with channel)
  status = dm_channel_start(&chanDesc);

  // Open device (Gets device handle)
  status = dm_device_open(&chanDesc, MAG3110_ADDR, &magDeviceHandle);

  // Explicitly Acquire Lock (not strictly necessary)
  status = dm_channel_acquire_lock(&chanDesc, timeout);

  // Write a command to the device
  status = dm_device_write(&magDeviceHandle, &mag3110WrReg);

  // Send command to read data from the device
  status = dm_device_read(&magDeviceHandle, &mag3110RdReg, pReadBuf);

  // Release Lock
  status = dm_channel_release_lock(&chanDesc);

  // <Do something with data in pReadBuf here>

  // When done talking to the mag
  // Stop the channel (Disables hardware peripheral associated with channel)
  status = dm_channel_stop(&chanDesc, timeout);
}
```

3.3.6 Host Proxy

ISF provides a Host Proxy component that allows a host processor to remotely configure and subscribe to all the managed sensors, whether internal to the platform or external.

3.3.6.1 Host Proxy Concepts and Theory of Operation

The Host Proxy's name derives from the fact that it acts as a local proxy for the remote host processor to facilitate communications with the Sensor Manager and other user applications.

The Host Proxy defines an interface that must be implemented and registered for each sensor needing proxy support. Figure 3-10 shows salient classes, data types and tasks that form the Host Proxy interface for a sensor. For details of the host proxy APIs, refer to the Xtrinsic ISF API Reference Manual for the FXLC95000 Intelligent Motion-Sensing Platform listed in References.

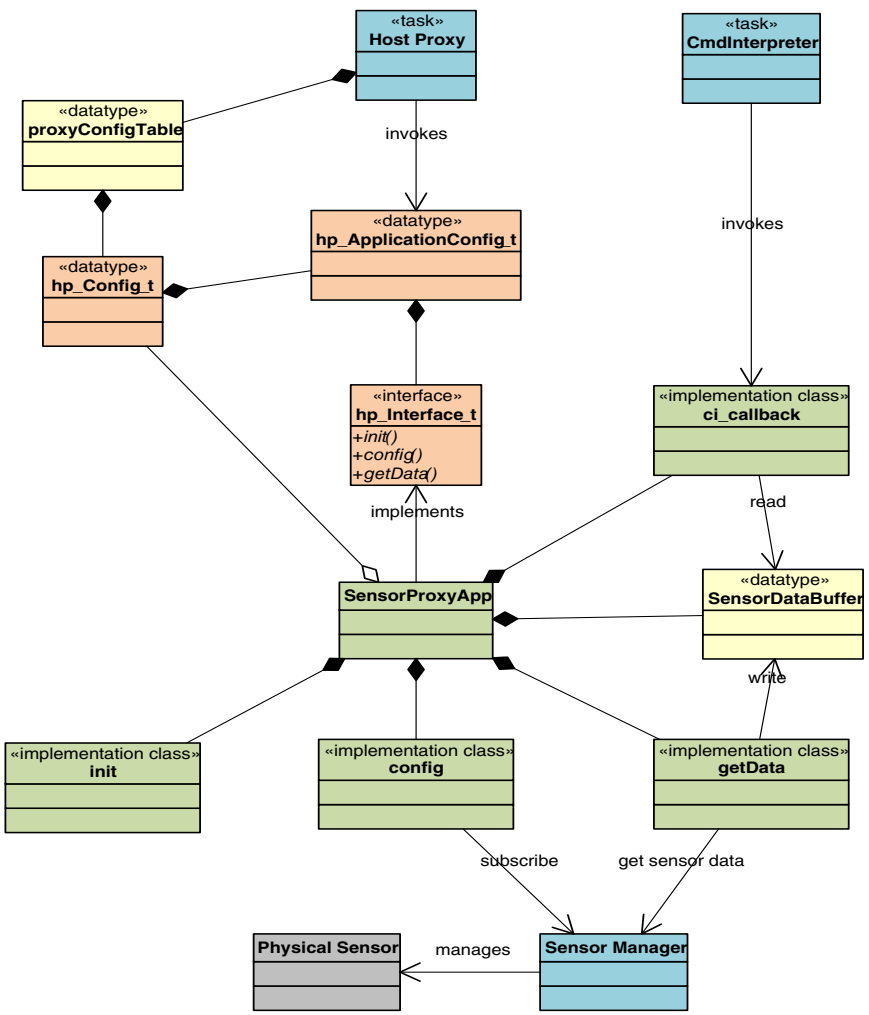

Figure 3-10. Host Proxy class diagram

The Host Proxy service executes as a single Freescale MQX™ RTOS task containing an event handling loop that blocks waiting for events from components such as the Sensor Manager (SM) and the Command Interpreter (CI). This single task design was used to keep the number of tasks in the system as low as possible.

At system startup, the Host Proxy service performs its initialization. During its initialization, the initialization functions for all registered sensor proxy applications are invoked. Once initialization is complete, the Host Proxy enters an event loop waiting for events. The Host Proxy handles two types of events: Configuration events and Data-Ready events. When the Host Proxy receives an event, it looks up the sensor proxy application registered for that event and invokes its corresponding registered callback. Several functions are provided that may be used within the sensor proxy application callback to interact with the Host Proxy.

3.3.6.2 Implementing a Proxy Application using the Host Proxy

If it is desirable to provide a custom interface for configuring and using a specific sensor or a specific sensor type, such as a gyroscope, a sensor proxy application can be developed using the Host Proxy service. The proxy application needs to satisfy certain minimum requirements as explained herein.

The sensor proxy application must implement the following three functions:

- Configuration interface,
- Host Proxy interface,
- Command Interpreter callback.

In addition, the sensor proxy application must be registered with the Host Proxy so that it can be initialized and executed properly.

3.3.6.2.1 Implementing the Proxy Application Configuration interface

The proxy application interface consists of some configuration data and a set of function pointers and is defined in *isf_hostproxyconfig.h* as:

```
typedef struct
   {
      /* The Application ID for this sensor proxy */
      uint8 appId;

      /* The sensor ID of the sensor being proxied */
      uint8 sensorId;

      /* The functions implementing the proxy interface */
      hp_Interface_t interface;

   } hp_ApplicationConfig_t;
```

The `appId` field contains the application ID used by the host to communicate with the sensor proxy application. It must be the same as the proxy's index in the `ci_callback` array as discussed in section Application Integration.

The `sensorId` is the ID of the sensor being proxied. It must be the same as the sensor's index in the `gSensorList` array in `isf_sensor_configuration.c`.

3.3.6.2.2 Implementing the Host Proxy interface

The interface field of the `hp_ApplicationConfig_t` structure is itself a structure defined as:

```
typedef struct
{
/* The proxy initialization function */
isf_status_t (*init)(void);

/* The proxy configuration function. */
isf_status_t (*config)(void);

/* The callback used to retrieve new sample data from the sensor*/
isf_status_t (*getData)(hp_hostUpdate_t *pHostData);

} hp_Interface_t;
```

The function pointers in this structure contain the specific proxy functionality in the sensor proxy application being implemented.

The `init` function is automatically invoked one time by the Host Proxy task during its initialization cycle at system startup. This is the appropriate place to initialize proxy-specific variables and data structures. For example, sensor configuration parameters can be initialized to their desired default settings and other one-time actions, such as semaphore creation, can be handled in the proxy's `init` function.

The sensor proxy application's `init` config function is invoked by the Host Proxy task each time it receives a Configuration command for that sensor from the host. Because new configuration data from the host is written directly into the proxy's configuration data buffer, the `config` function should be designed to re-read the proxy's configuration data and update accordingly. The Host Proxy itself does not impose any constraints on the format and contents of a sensor proxy application's configuration data buffer contents or layout. However, it is recommended that a uniform buffer layout be established and followed for consistency. In a typical sensor proxy application implementation, the configuration buffer should provide a control data structure that allows the host to subscribe and unsubscribe to the sensor, specify configuration parameters for the sensor like sample rate, data resolution, data ranges, etc. In general, the proxy application should enable the host to configure and use the sensor through the proxy with the same amount of control as it would have if it were accessing the sensor directly via the Sensor Manager interface.

As an example, the Freescale provided sensor application, `ISF1P095K_ACCELMAG_PROJ`, uses the structure shown below for the FXLC95000 internal accelerometer.

```
/* FXLC95000 Accel sensor proxy configuration structure */
typedef struct
{
    ctrl_reg_t         ctrl;           // ctrl information
    SM_SensorSetting_t settings;       // accel settings
```

```
} hp_fxlc95000Config_t;
```

The `ctrl_reg_t` structure is defined, for ease of reuse, in *isf_hostproxy.h* as:

```
typedef union
{
  /* The control byte. */
  uint8 reg;

  struct {
      /* The data streaming bit.
       * When this bit is set, data streaming is enabled.
       * Data streaming is disabled by clearing this bit.
       */
    uint8 startStop      :1;

      /* The subscription state bit.
       * Setting this bit enables the subscription.
       * Clearing this bit disables the subscription.
       */
     uint8 subState        :1;

     /* The host interrupt bit.
      * Setting this bit enables the host interrupt.
      * Clearing this bit disables the host interrupt.
      */
    uint8 hostIntEn        :1;

     /* The quick read bit for the application.
      * Setting this bit enables the quick read.
      * Clearing this bit disables the quick read.
      */
    uint8 quickReadEn      :1;

     /* These bits are for future use. */
    uint8 reserved         :4;
  } Bits;

} ctrl_reg_t;
```

The sensor subscription settings are directly exposed to the host using the `SM_SensorSetting_t` as defined by the Sensor Manager in *isf_sm_api.h*. Because the Host Proxy contains a single event loop for all the sensor proxy applications in the system, and because it uses the Freescale MQX™ RTOS `lwevent` facilities for event notifications, all the events handled by the Host Proxy must belong to the same event group. Therefore, the Host Proxy is designed to allocate the event group it uses and provides functions for each sensor proxy application to call to return the event group and event flag it should use to signal a Data-Ready event for its sensor. In the case where the sensor proxy application is subscribing to the Sensor Manager for its sensor data, a sensor proxy application can pass this event group and flag pair to the Sensor Manager in its subscription whereby the Host Proxy is automatically notified when new data is available for that sensor. Upon receipt of the event, the Host Proxy invokes the sensor proxy application's `getData()` interface function which can then call the `isf_sm_get_sensor_data()` function to obtain the new sample.

When it invokes the `getData()` callback, the Host Proxy passes a pointer of type `hp_hostUpdate_t` as a parameter. `hp_hostUpdate_t` is defined as:

```
typedef struct
{
    /* Status indicating whether or not Quick Read is enabled. */
  uint8 isQuickReadUpdate;

    /* Status indicating whether or not host interrupt is enabled.*/
  uint8 isHostInt;

    /* The size of the buffer holding user application data. */
  uint8 size;

    /* The current application ID for the user application. */
  uint8 AppId;

    /* The buffer containing user application data. */
  uint8 *pDataBuffer;

} hp_hostUpdate_t;
```

The sensor proxy application's `getData()` callback is responsible for filling this structure with valid data which gives the Host Proxy the information necessary for it to handle the new data appropriately. For example if the sensor proxy application is supporting mailbox quick read data it should set the `isQuickReadUpdate` field to 1. This causes the Host Proxy to invoke `isf_ci_qr_update()` on behalf of the sensor proxy application.

Likewise, if the host has requested an interrupt from the sensor proxy application when new data is available, the proxy application should set the `isHostInt` field to 1. This causes the Host Proxy to invoke `isf_ci_assert_int_o()` on behalf of the sensor proxy application.

The sensor proxy application needs to know the event group and event flag to use to notify the Host Proxy task of Data-Ready events. The `isf_hp_get_proxy_event()` may be called to obtain the event flag the sensor proxy application should use. The event group to use is declared by the Host Proxy as a global variable and may be obtained by referencing `gProxyEvent` declared in *isf_hostproxy.h*. These methods require the proxy number and the event number as an input. Proxy number is a unique number assigned at the time the application is registered with the host proxy and the event number is the category number, which could be either configuration or data ready.

The `isf_hp_set_proxy_event()` function allows the sensor proxy application to set a Configuration event which will cause the Host Proxy to invoke the sensor proxy application's registered `config` function. This is typically done in the proxy application's registered Command Interpreter callback function after a `CI_CMD_WRITE_CONFIG` command has been processed and it is necessary to allow the sensor proxy application to update to the newly written configuration.

Once all interface functions have been implemented, a configuration structure can be defined and initialized with function pointers to these implementations. The configuration structure is of type `hp_ApplicationConfig_t`. The configuration structure has references to the applicable sensor id and application ID. For example, a proxy configuration for the FXLC95000's internal accelerometer might be defined as:

```
hp_ApplicationConfig_t fxlc95000Config = {
  .appId             = APP_PROXY_INTERNAL_ACCEL,
  .sensorId          = DSA_FXLC95000_3D_ACCEL,
  .interface.init    = &internal_accel_app_init,
  .interface.config  = &internal_accel_app_config,
  .interface.getData = &internal_accel_app_get_data,
};
```

3.3.6.2.3 Implementing the Command Interpreter callback

To interact with the host, the sensor proxy application must have a callback registered with the Command Interpreter as described in section Application Integration. An example callback for the FXLC95000's accelerometer might look as follows:

```
ci_response_t fxlc95000_accel_app_callback(
    ci_host_cmd_packet_t *pHostPacket,
    ci_app_resp_packet_t *pAppPacket)
{

     ci_response_enum callbackRet = CI_ERROR_NONE;

  // Each Sensor Proxy App would assign the following data from
  // sources appropriate to its implementation.
     uint8 *pDataBuffer = <a pointer to the app's data buffer>;
     int8   size        = <the size of the app's data buffer>;

     switch(pHostPacket->cmd)
     {
         case CI_CMD_READ_APP_DATA:
            if(pHostPacket->byte_cnt > size)
            {
                callbackRet = CI_INVALID_COUNT;
                break;
            }
            if(pHostPacket->offset +  pHostPacket->byte_cnt > size ){
                callbackRet = CI_ERROR_COMMAND;
                break;
            }
            pAppPacket->bytes_xfer = (uint8)isf_ci_app_write
                (
                    pHostPacket->appId,
                    (uint32)pHostPacket->byte_cnt,
                    (uint8*)(pDataBuffer + pHostPacket->offset)
                );
            break;
        case CI_CMD_UPDATE_QUICKREAD:
                isf_ci_qr_update
                    (
                APP_ID_FXLC95000_ACCEL,
                 size,
                 pDataBuffer
              );
                break;
        case CI_CMD_READ_CONFIG:
        case CI_CMD_WRITE_CONFIG:
            if(pHostPacket->byte_cnt > FXLC95000_ACCEL_APP_NUM_REGS ){
                callbackRet = CI_INVALID_COUNT;
                break;
            }

            if(pHostPacket->offset +  pHostPacket->byte_cnt >
               FXLC95000_ACCEL_APP_NUM_REGS)
            {
```

```
                callbackRet = CI_ERROR_COMMAND;
                break;
            }

            if (CI_CMD_READ_CONFIG == pHostPacket->cmd)
            {
                pAppPacket->rw = CI_RW_WRITE; // Write the data to host.

                pAppPacket->bytes_xfer = (uint8)isf_ci_app_write
                    (
                        pHostPacket->appId,
                        (uint32)pHostPacket->byte_cnt,
                        pDataBuffer
                    );
            }
            else
            {
                // Read data from the host, which essentially sets
                // the application's configuration register.
                pAppPacket->bytes_xfer = (uint8)isf_ci_app_read
                    (
                        pHostPacket->appId,
                        (uint32)pHostPacket->byte_cnt,
                        pDataBuffer
                    );
                isf_hp_set_proxy_event
                    (
                        APP_PROXY_NO_FXCL95000_ACCEL,
                        EVENT_HOST_CONFIG
                    );
            }
            break;
        default:
            callbackRet = CI_ERROR_COMMAND;
            break;
    }

  return callbackRet;
}
```

3.3.6.2.4 Implementing the Proxy Application Registration

Once the sensor proxy application's Host Proxy interface functions have been implemented and a proxy configuration struct has been defined, it is necessary to register them with the Host Proxy. The Host Proxy configuration table array, `proxyConfigTable[]` array in file `isf_hostproxyconfig.c` contains the list of registered applications. Each array element is a configuration structure containing a unique proxy number to be used for each application and a pointer to an application configuration structure. To register a new sensor proxy application, a new entry in the `proxyConfigTable` array must be added with the new application's data. For example, an application that provides a simple interface to the FXLC95000's internal accelerometer might be added as follows:

```
  hp_Config_t proxyConfigTable[] = {
          {
        .proxyNo = A_PREVIOUSLY_EXISTING_PROXY,
        .appConfig = &aPreviouslyExistingConfig
          },
          {
        .proxyNo = APP_PROXY_INTERNAL_ACCEL,
        .appConfig = &fxlc95000Config
```

```
        }
    };
```

For convenience, the proxy numbers are defined in *isf_hostproxyconfig.h* file as an enum and corresponding additions to the enum list can also be made:

```
enum hp_AppProxyNO
    {
        A_PREVIOUSLY_EXISTING_PROXY = 0,
        APP_PROXY_INTERNAL_ACCEL    = 1
    };
```

The Host Proxy will initialize and execute all sensor proxy applications it finds in the `proxyConfigTable` as discussed above.

3.3.7 I²C Master Interfaces

The Master I²C component is delivered as part of the ISF core library but is optional in an ISF deployment - that is, the I²C component is only linked into the generated image if referenced by the application code.

3.3.7.1 Theory of Operation

The main Master I²C objects are buses and slaves. A bus is used to represent each separate I²C communications pathway to one or more external I²C slave devices. When a bus is initialized, a BusHandle is also initialized and returned to the calling function. The calling function uses this BusHandle to invoke operations on the bus. In a similar manner, a SlaveHandle is initialized and provided for each slave device opened on a bus. The SlaveHandle is passed to the I²C APIs when operations with that slave device are desired.

The Master I²C component is delivered as part of the ISF core library but is optional in an ISF deployment - that is, the I²C component is only linked in to the generated image if referenced by the application code. The Master I²C library installs its own peripheral interrupt handler with the RTOS kernel when initialized.

The I²C APIs cover bus operations including initialization, locking, reconfiguration, status query and control, as well as slave operations including open/close, and read/write. For details of these APIs, refer to the Xtrinsic ISF API Reference Manual for the FXLC95000 Intelligent Motion-Sensing Platform listed in References.

3.3.7.2 Bus Locking

An explicit bus locking capability allows extended exclusive access to a bus. When a bus lock is held, no other task may communicate with any slave devices on that bus until the lock is released. Calls to device operations without first acquiring an explicit bus lock will cause an implicit bus lock to be acquired but only for the duration of that call. Bus locks are implemented with priority inversion protection using a priority inheritance scheme that automatically raises the current lock holder's priority to the priority of the highest waiting task until the lock is released.

It should also be noted that holding an explicit bus lock automatically selects the use of the I^2C repeated start capability between sequential bus messages. Care should be taken to avoid locking the bus because some slave devices do not handle a repeated start condition gracefully.

3.3.7.3 Usage Example

In the following example, communication with an I^2C slave device using the I^2C APIs is demonstrated.

```
#include <isf_i2c.h>
#include <isf_sysconf_comms.h>

typedef I2C_SIZED_BUFFER(18) ReadBuffer18_t;

const comm_Address_t  MAG3110_ADDR  = 0x0E;     // I2C Bus address of MAG3110

// A command to initialize the magnetometer
static const i2c_Command_t mag3110WrReg = {2, 0, {0x10, 0x00}};

// A command to read magnetometer data
static const i2c_Command_t mag3110RdReg = { 1, 1, {0x01}};

void get_mag_data()
{
// Setup a buffer to hold data from the magnetometer
ReadBuffer18_t          readBuffer  = { .size=18 };
i2c_MessageBuffer_t     *preadBuf    = (i2c_MessageBuffer_t *)&readBuffer;
isf_status_t            status;
isf_duration_t          timeout = 0;       // zero will mean 'wait forever'
i2c_BusHandle_t         hBus1;             // holds the returned bus handle
i2c_SlaveHandle_t       magSlaveHandle;    // holds the returned slave handle

// Init creates and initializes data structures required to manage the bus.
// The bus Id enumeration comes from sysconf_comms.h::sys_channelId_t and
//     identifies the I2C channel to use
status = i2c_bus_init(I2C_MASTER1, &hBus1);

// Configure the I2C bus
status = i2c_bus_configure(&hBus1, &gSys_I2cBusConfig[I2C_MASTER1]);

// Start Bus (Enables Hardware Peripheral associated with channel)
status = i2c_bus_start(&hBus1);

// Open device (Gets device handle)
status = i2c_get_slave_at(&hBus1, MAG3110_ADDR, &magSlaveHandle);
```

```
// Explicitly Acquire Lock (not strictly necessary)
status = i2c_bus_acquire_lock(&hBus1, timeout);

// Write a command to the slave
status = i2c_write(&magSlaveHandle, &mag3110WrReg);

// Send command to read data from the slave
status = i2c_read
(
  &magSlaveHandle,
  &mag3110RdReg,
  pReadBuf,
  I2C_READ_FLAGS_DEFAULT
);

// Release Lock
status = i2c_bus_release_lock(&hBus1);

// <Do something with data in pReadBuf here>

// When done talking to the mag
// Stop the bus (Disables hardware peripheral associated with channel)
status = i2c_bus_stop(&hBus1, timeout);
}
```

For details of these APIs and examples of their use, refer to the Xtrinsic Intelligent Sensing Framework API Reference Manual for the FXLC95000 Intelligent Sensor listed in References.

3.3.8 Communications Channel Configuration

The communications channel configuration information is organized in an easy-to-use component called the Communications System Configuration component.

The available channels for use with Device Messaging are declared in the *isf_sysconf_comms.h* file and the corresponding data structure is defined in *isf_sysconf_comms.c*. These files are provided as source enabling the user to modify and compile them according to their particular system configuration.

The typical user only needs to include the *isf_sysconf_comms.h* file and select the enumeration values defined in `sys_channelId_t` to pass to `dm_channel_init()`. In addition, the user must ensure that the *isf_sysconf_comms.c* file is compiled and linked in their executable image.

3.3.9 Bus Management

The ISF Communications service family provides the capability to schedule periodic bus communications via the Bus Manager component. This allows a user to design an application that initializes, registers periodic communications functions with the Bus Manager and enters an event loop to handle incoming events without explicitly managing the timing of communications with selected slave devices.

To schedule periodic bus communications using the bus management functionality, users create and register callback functions with the Bus Manager. A period is the desired time between successive invocations of the callback and is supplied at the time of registration as a parameter. A token is returned to the user upon successful registration of the callback function and is used in subsequent bus management calls when a particular callback function must be referenced. For example, a token is required when enabling or disabling callback activation or requesting to unregister the callback.

Once a callback function has been registered with the Bus Manager, it can be activated by passing its token to `bm_start()`. The Bus Manager invokes active callbacks periodically at their registered rates. To temporarily stop a callback function from being invoked, the callback's token is passed to `bm_stop()`. Multiple callbacks may be controlled together by performing a logical OR of several token values and passing the result to `bm_start()` or `bm_stop()`.

In the following example, two callback functions are registered, one at 1000 Hz and one at 100 Hz. Each callback function simply increments a counter and returns. The counter values reflect the number of times the callback function was invoked.

```
/* This is example code intended to demonstrate basic usage of the
** APIs only and does not include all of the error/return-code
** checking typically found in production code. */

#include <bus_management.h>

uint32 func_cnt[2]; /* holds counts of callback invocations */
uint8  idx1 = 0;    /* index into func_cnt[] for first callback */
uint8  idx2 = 1;    /* index into func_cnt[] for second callback */
isf_status_t st;   /* holds return status from isf calls */

/*
** This is the callback that will be registered with Bus Management
*/
void cb_func(void *p)
{

    /* For the example just increment a counter */
    uint8 *idx = (uint8 *)p;
      ++func_cnt[*idx];
}

/* Returns number of microseconds per period for the
** specified frequency
*/
isf_duration_t freq_to_usec_period( uint16 aFreq_hz)
{
    return (1000000 / aFreq);
}


void test_callbacks()
{
    bm_callback_token_t token1; /* returned from bm_register_callback() */
    bm_callback_token_t token2; /* returned from bm_register_callback() */

    token1 = bm_register_callback
             (
             freq_to_usec_period(1000),
             cb_func,
```

```
            &idx1       /* pointer that will get passed to cb_func() */
            );

    token2 = bm_register_callback
            (
            freq_to_usec_period(100),
            cb_func,
            &idx2       /* pointer that will get passed to cb_func() */
            );

    /* Callbacks are registered so start the timers */
    bm_start(TRUE, BM_ALL_TOKENS); /* TRUE means synchronize the callback periods */

    /* Wait for 10 seconds and let the callbacks run */
    _time_delay(10 * 1000);  /* _time_delay() takes milliseconds */

    /* Stop the callbacks */
    bm_stop(BM_ALL_TOKENS);

    /* And unregister them */
    st = bm_unregister_callback(token1);
    st = bm_unregister_callback(token2);

    // Add call to user-provided function that verifies the counts of callback invocations.
        // For example:
        //
        // verify_counts(func_cnt);
        //
}
```

3.3.10 Built-in Commands

In addition to the built-in Mailbox Application, ISF provides a Device Information command that returns hardware, firmware, and software information pertaining to the device.

3.3.10.1 Device Info command

The Device Info command (DevInfo) is a special Command/Response mode command. It does not conform to the complete Command/Response protocol described previously. Instead, it operates the same way as the FXLC95000's ROM Command Interpreter's `CI_DEV_INFO` Command. The DevInfo command is invoked at runtime by writing two zeros in MB0 and MB1. This commonality allows a "0x0 0x0" to be sent to the part regardless of its operating mode , either the ROM or the ISF Command Interpreter, and device information is returned. It is also possible to determine whether the FXLC95000 is operating out of ROM or flash by examining the Firmware Version information in MB12 and MB13. This is reported as 0xFF 0xFF only if it is in ROM CI mode.

The DevInfo command is also automatically invoked when initializing out of flash memory, provided the ISF Command Interpreter component is enabled in the firmware image loaded on the part. Internally the command executes by invoking the ROM Command Interpreter's `CI_DEV_INFO` command and overlaying its results with the ISF

specific information. The table below describes the device info data returned in the mailboxes and whether the information came from the ROM CI or from the ISF firmware. See the FXLC95000CL Hardware Reference Manual listed in References for more information about the ROM CI, the commands available and meanings of the RCSR and FOPT register values.

The DevInfo command places the following data in the mailboxes:

Table 3-11. Mailboxes and fields of the DevInfo command

Mailbox number	Field Name	Source	Description
MB00	Command	ISF	Echoes the Application ID providing the Response (for example, 0)
MB01	Command Status	ISF	If the DevInfo data is in response to a Command, e.g. “0x0 0x0” is written to the mailboxes, then a command response is written in MB1, 0x80 indicates successful completion. If the DevInfo in the mailboxes is due to a reset/ initialization, then no explicit command is sent, and the Command Status field is set to 0x0.
MB02-05	device_id	ROM	32 bit pseudo random part identification value
MB06-07	rom_version	ROM	16 bit ROM version code: major.minor (for example, 01 00 = 1.0)
MB08-09	fw_version	ISF	16 bit firmware version code: major.minor (for example, 01 2C = 1.44)
MB10-11	hw_version	ROM	16 bit hardware version code: major.minor
MB12-13	build_code	ISF	16 bit firmware build number and date code. The value is encoded in the following bit fields: [15:13] daily build number, 0 to 7 [12: 9] build month, 1 to 12 [8: 4] build day, 1 to 31 [3: 0] build year, 2012 to 2027 E.g., 0x23 0x61 in mailboxes 12-13 would decode as:

Table 3-12. Build code encoding example

	Build #	Month	Day	Year
Bit value	001	0001	10110	0001
Decoded	1st build	January	22	2013

The device information can also be retrieved programmatically using the `_fw_device_info_get(device_info_t *info_ptr)` command. The `_fw_device_info_get()` command fills the memory at the passed in pointer location with data according to the following structure:

```
typedef struct {
    uint_32   device_id;
    uint_16   rom_version;
    uint_16   fw_version
    uint_16   hw_version;
    uint_16   build_code
    uint_16   part_number
    uint_8    reset_cause;
```

```
    uint_8    secure_mode;
} device_info_t;
```

3.4 Sensor Management

The Sensor Management service family provides the capability to manage the operating modes and configurations of multiple sensors. It provides uniform interfaces to obtain sensor data from multiple sensors.

The ISF Sensor Manager (SM) component provides uniform interfaces to applications running on either the FXLC95000 or the external host for accessing physical data measured by the on-board accelerometer in FXLC95000CL and other sensors external to FXLC95000CL. The SM implements the well known client-server model wherein a server component services one or more clients.

The ISF SM component (acting as server) is architected to provide sensor data to registered applications (clients) that need sensor data. It is also helpful to think of the sensor manager architecture as an implementation of the well known publish-subscribe design pattern. Sensors managed by the SM will have their data published at some sample rate. The SM interface allows each application to subscribe to a particular sensor's data at different output data rates. This is accomplished using a signal tap mechanism. The SM interface allows up to three signal taps for any one sensor. The sample rate for each signal tap is configurable at the time of subscription.

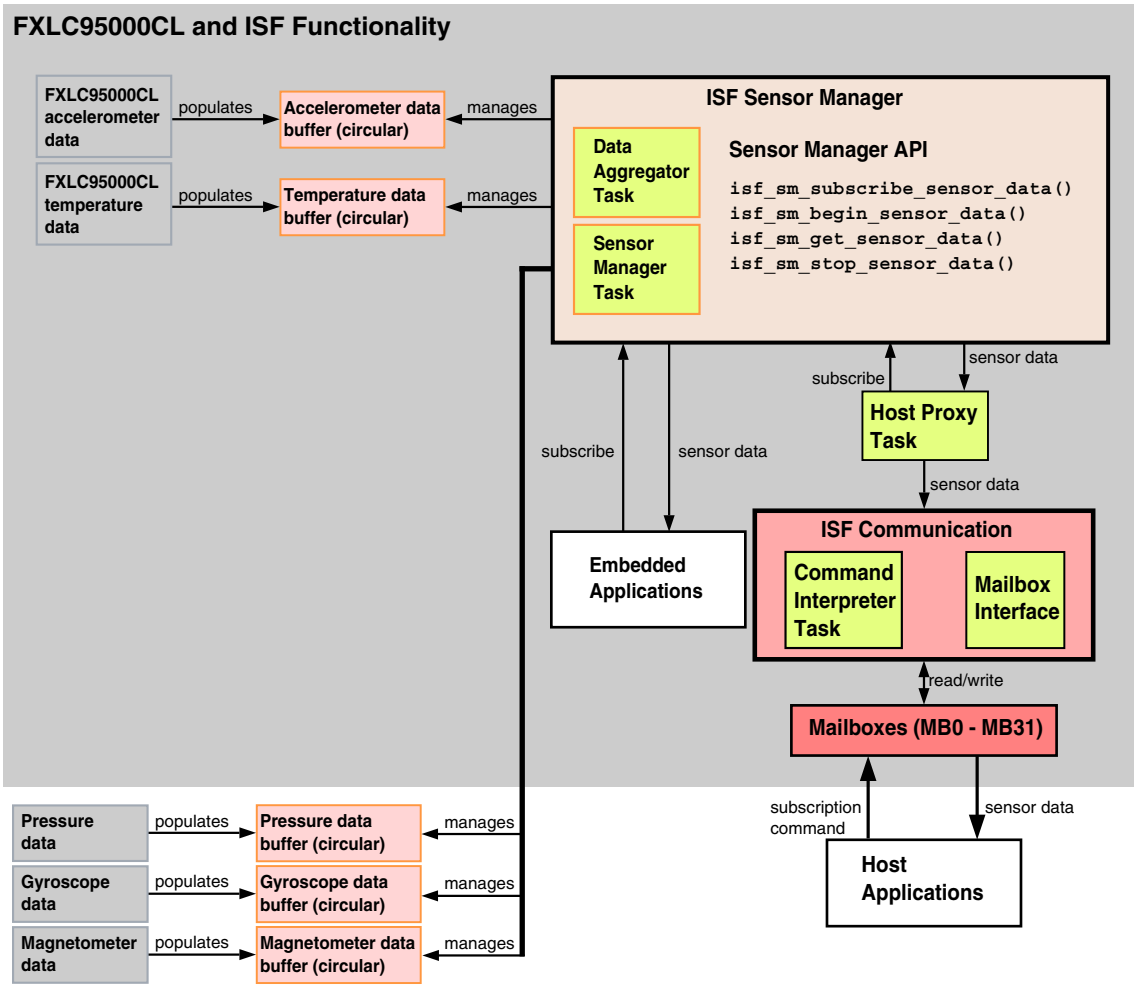

Figure 3-11. Sensor Manager Architecture

Figure 3-11 is a simplified view of the architecture of the ISF SM, ISF related components, and some example data sources.

The SM exposes sensor sample data via circular data buffers. Applications that run on the FXLC95000 platform can subscribe to sensor data via calls to the SM API. See Using the Sensor Management API for details. Host applications, however, need to use the slave interface to send a subscription command to the ISF Command Interpreter to subscribe to sensor data of interest. The user writes a host proxy task that fetches the requested data via the ISF SM API and provides it to the ISF communication component which then outputs the data via the Mailbox Interface. Alternatively, the ISF Host Proxy component could be used to do this.

Both embedded and host applications can choose to subscribe to the sensor data of interest. Each application can specify the number of bits required per sample. The SM arbitrates between multiple requests from multiple applications for the sensor data and determines the appropriate configurations to use—it controls the optimal population of the circular buffers for the sensor data to support all application needs.

By decoupling the production and consumption of sensor data, the SM increases scalability by removing all explicit dependencies between the interacting applications. This makes the ISF communication infrastructure well adapted to distributed environments that are asynchronous by nature, such as mobile environments.

3.4.1 Sensor Manager Signal Tap Mechanism

The Sensor Manager component provides every embedded application an interface to a particular sensor's data through a signal tap mechanism. A tap provides any registered consumer access to sensor data. A tap is configured to accumulate sensor data at a particular frequency known as the sample rate. A consumer may subscribe to a sensor data tap to receive data at a particular frequency known as the report rate. The report rate may be configured to be lower than the sample rate, resulting in more than one sample available for consumption each time the subscriber is notified that new sensor data is available. For example, if a sensor tap is configured to accumulate sensor data sampled at 488 Hz with a report rate of 122 Hz, then there are four new 488 Hz samples available each time a consumer is notified that new data is available.

3.4.2 Sensor Manager Subscription Tokens

When a subscriber successfully registers to receive sensor data, a unique token ID is returned. The Sensor Manager associates each subscription with its corresponding token ID. Subsequent Sensor Manager API calls require this token ID to be passed as a parameter.

3.4.3 Using the Sensor Management API

Sensor Manager (SM) API functions must be called in the correct sequence to successfully access sensor sample data.

An application must first register for sensor data by calling `isf_sm_subscribe_sensor_data()` with the appropriate parameters.

Note

The report rate for the subscription is actually specified in terms of the report period in microseconds. This allows the specification of frequencies less than 1 Hz in an integer representation. The report period in microseconds may be calculated from frequency according to the formula:

T = 1000000 / F

where:

T = period in microseconds (µs)

F = desired frequency in hertz (Hz)

It must also provide a FIFO depth and an event group for the Sensor Manager to notify the subscriber whenever new sensor data is available. An event group is a 32-bit group of event bits used to let tasks synchronize and communicate. There are two event group types: fast event groups and named event groups. The application is free to define one or more event groups where each event group may be for an event type, for example, a single tap event. If the subscription is successful, the Sensor Manager returns a unique token ID assigned to that subscription.

After successful subscription, the subscriber may request the Sensor Manager to send sensor data associated with the subscription by calling, `isf_sm_begin_sensor_data()`, passing the assigned token ID as a parameter. Event notifications are sent by the Sensor Manager when new data is available.

The SM places the sensor data into a unique data buffer associated with each subscriber. When notification is received, the subscriber task should call `isf_sm_get_sensor_data()` with the appropriate token ID and a pointer to a memory buffer to receive the data.

In the following example, the Sensor Manager APIs retrieve accelerometer data. This is example code intended to demonstrate basic usage of the APIs only and does not include all of the error/return-code checking typically found in production code.

```
#include <lwevent.h>
#include "isf.h"
#include "isf_types.h"
#include "isf_comm.h"
#include "isf_sm_api.h"

/*
 * Struct to hold the sensor data - filled by Sensor Manager in isf_sm_get_sensor_data()
 typedef struct {
     uint16      sampleNum;
     uint16      data[3];   //[0]=X, [1]=Y, [2]=Z
 } threeAxisData_t;
*/

LWEVENT_STRUCT    gSystemEvent;
threeAxisData_t   gAccelData;
// data ready event of accelerometer
#define EVENT_FXLC95000_ACCEL_DATA_RDY  (1<<1)

void sensor_mgr_example_app(uint_32 param)
```

```
{
#pragma unused (param)
    static _mqx_uint         signalledEvents;
    _mqx_uint                mqxStatus;
    sm_result_t              smStatus;
    uint16                   accelTokenID;
    SM_SensorSetting_t       myAccelSettings;
    uint32                   events = 0;
    uint32                   currentEventFlag = 0;
    uint32                   currentEventID = 0;

    // Set Default Sensor Settings
    myAccelSettings.nSettingsToUse     = SM_GIVEN_SETTINGS;
    myAccelSettings.nFifoDepth         = 1;
    myAccelSettings.nCurrentReportRate = 10000;  // 100 Hz data rate in usecs;
    myAccelSettings.nCurrentResolution = AFE_ACCEL_RESOLUTION_16_BIT;
    myAccelSettings.nRange             = AFE_ACCEL_G_RANGE_8G;

    smStatus = isf_sm_subscribe_sensor_data(
         SM_AFE_ACCEL_SENSOR_3D, &myAccelSettings, &gSystemEvent,
EVENT_FXLC95000_ACCEL_DATA_RDY, &accelTokenID);

    smStatus = isf_sm_begin_sensor_data(accelTokenID);

    // Read 1000 samples
    for ( i=0; i<1000; i++)
    {

      // _lwevent_wait_for() blocks until the Sensor Manager sets the
      // event indicating new data is available.
       mqxStatus = _lwevent_wait_for(&gSystemEvent, EVENT_FXLC95000_ACCEL_DATA_RDY, FALSE,
0);

      //  Get Sensor Data.
       smStatus = isf_sm_get_sensor_data(accelTokenID, (void  *)&gAccelData);

       gAccelDataSampleNum++;

       // do something with the data
       my_process_data_function(gAccelData);

       _lwevent_clear(gSystemEvent, EVENT_FXLC95000_ACCEL_DATA_RDY);
    }
    smStatus = isf_sm_end_sensor_data(accelTokenID);
    smStatus = isf_sm_unsubscribe_sensor_data(accelTokenID);
}
```

NOTE

If a FIFO depth of larger than one is specified when subscribing to the sensor data, the buffer must be allocated large enough to hold all data sets returned.

The Sensor Manager may be requested to pause delivery of sensor data by calling `isf_sm_end_sensor_data()` with the appropriate token ID. This suspends notifications of new data availability without ending the subscription. Delivery of sensor data can be resumed by calling `isf_sm_begin_sensor_data()`—this will provide current data in the memory buffer at the next notification.

NOTE

In ISF v1.1, the following conditions must be met.

- There must be three or fewer subscriptions per sensor.
- If there is more than one subscription:
  - Sampling frequencies of additional subscriptions must divide evenly into the highest subscription frequency.

3.4.3.1 Sensor Manager Subscription Quality of Service

Due to power consumption concerns, the Sensor Manager attempts to configure the physical sensor to run at the lowest possible frequency (longest period) that can fulfill its current subscriber's needs. Of course, this corresponds to the lowest CPU power consumption value. When a new subscription is received, or when an existing subscriber unsubscribes, the Sensor Manager evaluates its remaining subscriptions to see if the physical sensor's configured rate can be reduced while still meeting current subscriber needs.

Subscribers communicate their subscription needs to the Sensor Manager through the quality of service (QoS) parameter (`nSettingsToUse`) in the subscription request structure. The Sensor Manager uses this information to determine how it must configure the physical sensor in order to support all its current subscribers along with the new one. A subscription's QoS setting can be set to one of the following values:

- `SM_GIVEN_SETTINGS`
- `SM_BEST_POSSIBLE_SETTINGS`
- `SM_CURRENT_SETTINGS`

`SM_GIVEN_SETTINGS` instructs the Sensor Manager that the settings provided in the subscription must be used exactly as specified in the request. If they are not compatible, the subscription request fails and an error is returned. By using `SM_GIVEN_SETTINGS`, the developer tells the Sensor Manager the expected physical sensor configuration settings to change to support the new subscription request. For example, if the physical sensor is currently configured to sample at 100 Hz (period of 10,000 µs) and a new subscription is received with a QoS specification of `SM_GIVEN_SETTINGS` and a desired sample rate of 200 Hz (period of 5000 µs), the Sensor Manager reconfigures the physical sensor to sample at 200 Hz and provides every other sample to the existing subscriber in order to maintain its requested 100 Hz rate.

A QoS setting of `SM_BEST_POSSIBLE_SETTINGS` tells the Sensor Manager that the settings provided in the request are the subscriber's first choice but if these settings are not possible, then the closest compatible settings are acceptable. The discussion below describes how the Sensor Manager determines compatible settings. When `SM_BEST_POSSIBLE_SETTINGS` is used, it is possible that the physical sensor configuration settings are changed to support the new subscription QoS. For example, if the physical

sensor is currently configured to sample at 100 Hz (10,000 µs) and a new subscription is received with SM_BEST_POSSIBLE_SETTINGS for 90 Hz (11,111 µs), the subscription is adjusted to receive data at 100 Hz instead with no changes to the physical sensor configuration settings necessary. But if a new subscription request for 300 Hz (3333 µs) is received, the physical sensor configuration settings are changed to sample at 300 Hz and the existing subscriber starts receiving every third sample from the sensor.

A subscription specifying SM_CURRENT_SETTINGS means that the new subscriber should not affect the current physical sensor configuration settings in any way.The subscription parameters are adjusted to be compatible with the current physical sensor configuration settings. Whenever the Sensor Manager adjusts subscription parameters, they are adjusted in the subscription request structure passed by reference by the subscriber and a return code indicating that the subscription parameters were modified is returned. As a result, the subscriber is alerted to the change and can see the updated values.

3.4.3.2 Sensor Subscription Compatibility

It is useful to understand how the Sensor Manager handles subscribers and signal taps and how that translates to the configuration settings of the physical sensor. This enables the developer to choose the appropriate subscription settings when designing the application.

When subscribers request sensor data at different rates, the Sensor Manager configures the physical sensor at a base rate that can support all subscribers by providing every sample from the sensor to the highest rate subscriber and satisfying the lower rate subscriber's requested rate by skipping some samples. For example, if the physical sensor is configured to sample at 1000 Hz (1000 µs) to support subscriber A, but subscriber B wants data at 250 Hz (4000 µs), then subscriber B2 will receive every fourth sample from the sensor. All subscriptions must have rates that have an integer sample ratio with the base physical sensor sample rate. The fastest rate subscriber has an integer sample ratio of 1 and the lower frequency subscribers have integer sample ratios of 2 or higher.

When the Sensor Manager has no subscribers for a given sensor, the sensor is not sampled. As subscriptions are received, the Sensor Manager configures the physical sensor sample rate to accommodate the requested rate. For example, suppose the Sensor Manager receives its first subscription with a request for samples at 50 Hz (20,000 µs). The Sensor Manager configures the sensor to sample at 50 Hz and sets the subscriber's integer sample ratio to be 1. Subsequently, a second subscription is received for 100 Hz (10,000 µs) data. The Sensor Manager reconfigures the physical sensor to sample at 100

Hz and modifies the first subscriber's integer sample ratio to be 2. If a third subscriber then requests data at 25 Hz (40,000 µs), no change to the physical sensor configuration is required. The Sensor Manager simply sets the third subscriber's integer sample ratio to 4.

Suppose now, that the second subscriber unsubscribes, leaving two active subscriptions for 50 Hz (20,000 µs) and 25 Hz (40,000 µs) data. The Sensor Manager then evaluates the remaining subscriptions and determines that it can reduce the physical sample rate to 50 Hz (20,000 µs) and adjusts the subscriber's integer sample ratios to 1 and 2 respectively.

It is important to understand that when evaluating the compatibility of requests, the Sensor Manager is designed in the following way. It never raises the physical sensor sample rate above the highest requested rate from any new or current subscriber in order to satisfy a subscription request for lower rate data.

For example, even though a physical sensor rate of 100 Hz (10,000 µs) might satisfy a new request for 20 Hz (50,000 µs) data when the physical sensor is currently configured at a 50 Hz (20,000 µs) rate, the Sensor Manager does not configure the sensor for faster than the current rate to support the lower rate request. It still increases the base sensor rate in order to meet higher frequency requests. If the sensor was configured to sample at 50 Hz (20,000 µs) and a 100 Hz (10,000 µs) request is received, it reconfigures to support the 100 Hz (10,000 µs) request. At that point, if a request for 20 Hz (50,000 µs) is received, the request is supported with an integer sample ratio of 5. It should also be understood that even though the Sensor Manager does not reconfigure the physical sensor to sample at a rate faster than that of its highest rate subscriber, it is possible for the sensor to be configured faster than any current subscriber.

Consider the previous example with three subscribers at 50 Hz , 100 Hz, and 20 Hz (20,000 µs, 10,000 µs, and 50,000 µs) with the physical sensor configured for 100 Hz (10,000 µs). Now suppose that the 100 Hz subscriber unsubscribes. At this point, the Sensor Manager evaluates the remaining subscribers to determine if the physical sensor rate can be reduced, but it determines that there is no lower rate possible than can support both remaining subscriptions. In this particular case, the physical configuration remains at 100 Hz. When either the 20 Hz or 50 Hz subscriber unsubscribes, the physical sensor rate is reduced at that time. Once that reduction occurs, it is no longer possible for either one to resubscribe at their previous rate because a 50 Hz request is not compatible with an existing 20 Hz rate nor is a 20 Hz rate compatible with an existing 50 Hz rate. A 100 Hz subscriber also needs to resubscribe before the 20 Hz or 50 Hz subscribers are compatible.

3.4.4 Digital Sensor Abstraction (DSA)

The Sensor Manager (SM) communicates with its sensors via an abstraction layer called the Digital Sensor Abstraction (DSA). By design, SM functions are not sensor-specific. The DSA provides the means to implement sensor-specific functions. The DSA exposes the ability to initialize, configure, validate settings of, get state of, start, stop, and shut down a sensor.

These capabilities are exposed as a set of functions conforming to an interface defined by the SM and the set of functions implementing these functions for a given sensor is known as a DSA adapter. The architecture of the SM enables the user to write new DSA adapters and to associate these adapters with existing or new sensors connected to the platform.

For any system, the list of the sensors available is maintained in the System Configuration component (see System Configuration). This list associates each instance of a sensor on the system with a system-unique Sensor ID, a DSA adapter, and other specific instance data needed to uniquely address the sensor. The SM API enables its users to refer to sensors via their assigned Sensor ID when subscribing. Internally, the SM uses the provided Sensor ID either to lookup the sensor configuration information contained in the Sensor Configuration list or to invoke the appropriate DSA adapter functions.

3.4.4.1 Digital Sensor Abstraction Theory of Operation

The Sensor Manager uses Digital Sensor Abstraction (DSA) adapter functions to interact and manage its sensors. The DSA adapter functions are designed to allow multiple sensor instances of a particular type to all reference the same adapter. This means that instance-data specific to a particular sensor must be kept separate from the adapter code and passed into each adapter function through a reference pointer. Thus, the adapter may be thought of as a set of class methods, each taking an explicit *this* pointer in addition to any other arguments pertinent to the specific function.

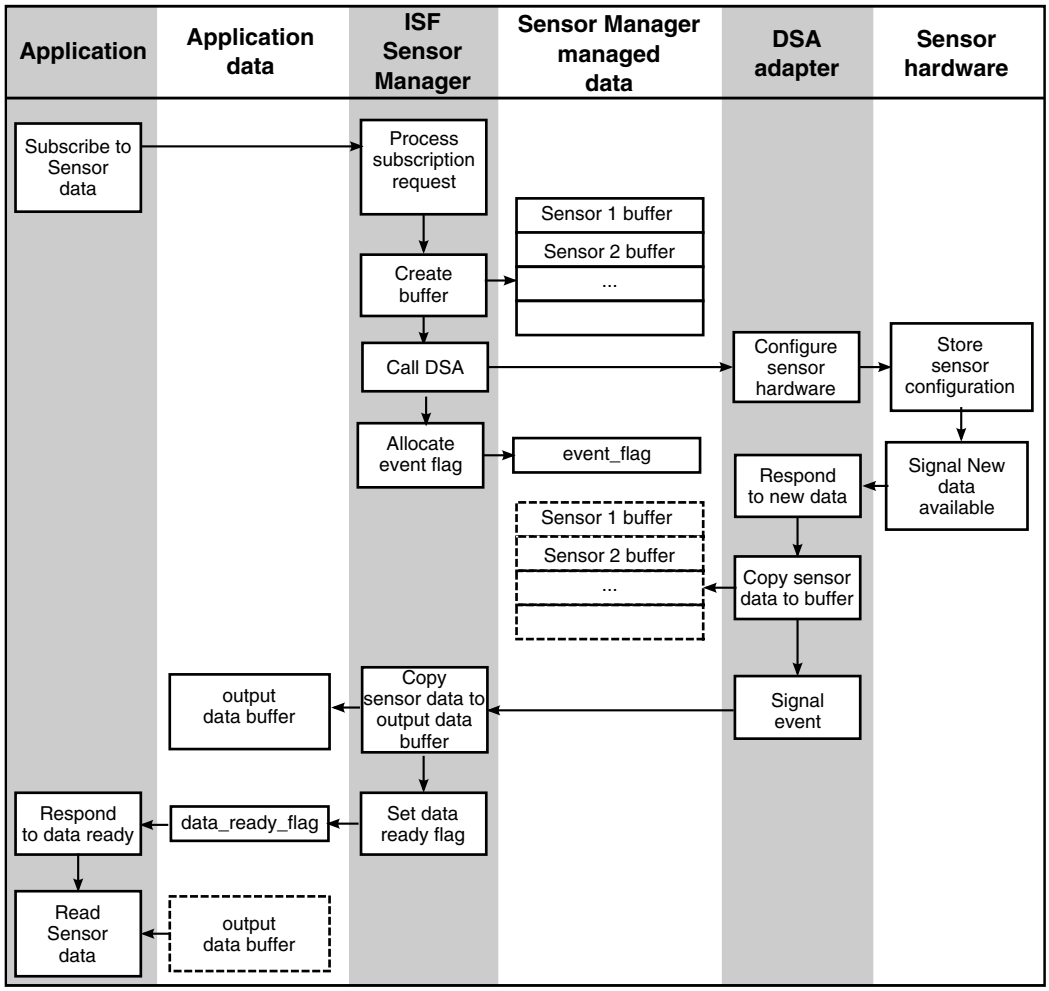

Figure 3-12. DSA operation

Figure 3-12 helps to understand the interface between the ISF Sensor Manager, the DSA adapter and the application. To set up receipt of sensor data, the Sensor Manager (SM) creates a buffer to hold a sample set from each sensor and allocates an event flag that the DSA adapter uses to signal the SM when new sensor data is available. After creating the buffer, the Sensor Manager initializes and configures the sensor, passing the instance data pointer, the event flag, a reference to the sample buffer and any other required sensor configuration data including the requested sample rate. The DSA configures the sensor hardware to provide samples at the specified rate, places new samples in the provided buffer and signals the Sensor Manager via the provided event flag each time a new sample is available. The application gets informed about the availability of data that it requested by the SM and may read the appropriate output buffer set up earlier for that data.

3.4.4.2 Implementing a New Digital Sensor Abstraction Adapter

New DSA adapters may be written by users to support new sensor types. The required type definitions are found in the file *isf_sm_dsa_adapter.h*. The source code for the Freescale MAG3110 I^2C DSA adapter implementation is also available as a full working example.

3.4.5 System Configuration

The ISF Sensor Configuration component maintains a list of all the sensors available on the platform along with the data structures necessary for the Sensor Manager to initialize, configure and use the sensors. This list is indexed using Sensor ID enumerations, also defined in the Sensor Configuration component. The Sensor Configuration component is provided as source-code to enable the user to add or remove sensors and the corresponding DSA adapters from the system.

Sensor Configuration component type definitions, structures and enumeration values are declared in *isf_sensor_configuration_extern.h* and are defined and initialized statically in *isf_sensor_configuration.c*.

To make a new sensor available to the Sensor Manager, the following steps must be followed:

1. Write or obtain a DSA adapter for the new sensor type. Examples of sensor types are gyroscope and altimeter.
2. Modify *isf_sensor_configuration.h* to add a new enumeration value for the new sensor. This is most easily done by adding the new sensor to the end of the existing sensor configuration list.
3. Modify *isf_sensor_configuration.c* to:
   - add the new entry in the Sensor Configuration array with values appropriate for the new sensor.
   - update the global `gNumSupportedSensors` variable with the total number of available sensors.

The following example sensor configuration file demonstrates the declaration of two sensors.

```
// File: isf_sensor_configuration.c
#include <isf_sensor_configuration.h>
#include <fsl_fxlc95000_mmap_3D_accel_config_types.h>
#include <fsl_mag3110_i2c_3D_mag_config_types.h>
static mag3110_Specific_Config_t     mag3110SpecificSettings;

SensorConfig_t   gSensorList[] =
{
   // The onboard accelerometer
   {
```

```
    &fsl_fxlc95000_mmap_3D_accel_adapter, // adapter code to use
    NULL,                                 // (allocated by SM)
    NULL,                                 // pointer sensor-specific config data
    INTERNAL_CHAN,                        // channel to use
    0xFFFEC4                              // device address
  },

  // An I2C FSL MAG3110 magnetometer
  {
    &fsl_mag3110_i2c_3D_mag_adapter,      // adapter code to use
    NULL,                                 // (allocated by SM)
    &mag3110SpecificSettings,             // pointer sensor-specific config data
    I2C_MASTER1,                          // channel to use
    0x0E                                  // device address
  }
};

uint8 gNumSupportedSensors = sizeof(gSensorList)/sizeof(SensorConfig_t);
// (in this case 2)
```

3.5 Power Management

The Power Manager (PM) exposes simplified functions to set the device behavior with respect to operating power modes of the device. In addition, it is responsible for maintaining operational consistency for ISF components as the device power states are changed.

3.5.1 Power Management Concepts and Theory of Operation

The Power Manager (PM) provides APIs that allow an embedded application to request changes to the operating power mode of the device.

In general, power savings is expected to be achieved by controlling CPU cycling and the speed and enable status of the clock(s) driving the device and its peripherals when the device is idle.

Each port of ISF to a new hardware device maps the ISF-defined power modes to its available device-specific power modes and must ensure that the total power consumed in each mode is less than or equal to each higher-powered mode subject to CPU loading constraints.

There are four high-level abstract power modes defined by the ISF PM component in `isf_pm.h` - one normal operating power mode and three reduced power standby modes:

```
typedef enum
{
  /* Normal operating power level */
  ISF_POWER_NORMAL    = 1,

  /* Low power level */
  ISF_POWER_LOW       = 2,
```

```
  /* Lowest power */
  ISF_POWER_LOWEST    = 3,

  /* Sleep level */
  ISF_POWER_SLEEP     = 4

} pm_power_level_enum;
```

These four power modes are generally expected to conform to the following high-level behavior:

- The `ISF_POWER_NORMAL` mode corresponds to the normal full-power operating mode- all clocks operate at full speed.
- The `ISF_POWER_LOW` mode is the first lower power mode- The CPU may enter STOP when finished with computations but all clocks continue to operate at full speed. Recovery from `ISF_POWER_LOW` mode back to `ISF_POWER_NORMAL` occurs when any interrupt is received.
- The `ISF_POWER_LOWEST` mode is the lowest power mode supported by a device that is internally recoverable. The CPU enters STOP when finished with computations and the clock operates at reduced speed until recovery via any interrupt.
- The `ISF_POWER_SLEEP` mode is a device sleep mode in which the CPU enters STOP when finished with computations and the clock is shut off. Recovery from `ISF_POWER_SLEEP` mode is only via external activity such as an external interrupt, reset, host sending messages to CI etc.

The PM is designed to run as the lowest priority task in the system. This ensures it is safe for the PM to command the device to enter the configured low power mode when the PM task executes.

3.5.2 Power Management Design

The PM is designed to run as a task with an initialization function, two interface API functions, and one interrupt service routine (ISR). The initialization function is called at system startup and performs one-time static initialization. The two interface API functions are exposed to the application developer to set the desired power level and to get the current power level setting. The ISR is used internally to keep track of time when the device is in low power. The PM task implements the low power functionality and is created along with other Freescale MQX™ RTOS tasks during startup.

At startup, the PM performs a one-time initialization, including creation of Freescale MQX™ RTOS objects and registration of the Start-Of-Frame (SOF) ISR. It also initializes the power to the `ISF_POWER_NORMAL` setting. After initialization has been completed, the PM idle task is created by Freescale MQX™ RTOS along with all the other tasks in the system.

The idle task runs at the lowest priority level in the system. This means it will be executed by the OS only when all other higher priority tasks are inactive, for example by being blocked on Freescale MQX™ RTOS objects waiting for an event occur. When the idle task does run, it sets the device to the currently active power level. Thus, for all power level settings except for `ISF_POWER_SLEEP`, power management is automatically implemented whenever all higher priority tasks are inactive.

If the power level is set to `ISF_POWER_NORMAL`, the `isf_power_set()` function sets the level and exits. When the idle task does run, it performs a task block operation on itself, to prevent it from running to completion. When an application subsequently sets the power level to something other than `ISF_POWER_NORMAL`, the idle task is unblocked, allowing it to complete and initiate the low power modes.

If the power level is set to `ISF_POWER_LOW`, when the idle task does run, it puts the device into $STOP_{FC}$ mode. In this mode, any interrupt wakes the CPU. At a minimum, the Freescale MQX™ RTOS timer is running and wakes up the CPU when its interrupt occurs on a periodic basis. The application may also have programmed other peripherals on the device to generate interrupts as well, causing a wake up. After interrupt ISRs are serviced and when all higher priority tasks are inactive, the idle task runs and returns the device to $STOP_{FC}$ mode assuming that the power level remains set at `ISF_POWER_LOW`.

If the power level is set to `ISF_POWER_LOWEST`, when the idle task runs, it puts the device into $STOP_{SC}$ mode unless the BM has active callbacks. In this case, the PM behaves in the same way as the `ISF_POWER_LOW` power level. For the case when the BM has active callbacks when the idle task runs, see Timer Service for further details.

If the power level is set to `ISF_POWER_SLEEP`, when the idle task runs, it immediately puts the device into $STOP_{NC}$ mode. In this mode, all clocks in the device are stopped. An external interrupt or slave port access from the host is required to wake the device.

3.5.3 Power Level implementations for the FXLC95000

The PM operating modes are implemented using the FXLC95000 STOP modes. The STOP modes are briefly described below. For more details see the FXLC95000 Hardware Reference Manual listed in References.

By configuring the hardware for the desired STOP mode behavior and executing the STOP machine instruction, the FXLC95000 can be taken from normal operational mode to one of the following modes:

Stop Fast Clock ($Stop_{FC}$)	In this mode, the CPU clock is stopped but peripheral clocks continue to run at normal speed. Any interrupts cause the CPU clock to resume at normal speed.

Stop Fast Clock ($Stop_{SC}$)	In this mode, the CPU clock is stopped and peripheral clocks continue to run but at a reduced speed (CLK/256). Any interrupts cause the CPU and peripheral clocks to resume at normal speed.
Stop Fast Clock ($Stop_{NC}$)	In this mode, the CPU and all peripheral clocks are stopped. An external interrupt or host slave port access causes the CPU and peripheral clocks to come out of sleep and resume processing at normal speed.

The power mode mapping for the FXLC95000 is as follows:

Power Mode	FXLC95000 Specific Mode Mapping				
	FXLC95000 Operational Mode	CPU Stopped When Idle	Bus Clock Speed During Idle	CPU Awakened By Interrupt	Rate-Monotonic Time-Base
ISF_POWER_NORMAL	RUN	No	Full	n/a	Yes
ISF_POWER_LOW	$STOP_{FC}$	Yes	Full	All	Yes
ISF_POWER_LOWEST	$STOP_{SC}$	Yes	Slow	All	Yes
ISF_POWER_SLEEP	$STOP_{NC}$	Yes	Stop	Ext only	No

3.5.4 Using the Power Management Interfaces

The PM interface provides a simplified interface for device power control. There are only two main control API functions in the interface:

`power_level_t isf_power_set( power_level_t aLevel)`	This API is used to set the desired power level. After setting the power level via this API, the power level is used the next time the idle function is invoked. If the requested power level cannot be set, the current power level is returned.
`power_level_t isf_power_get(void)`	This API obtains the current power level setting.

3.5.5 Timer Service

One of the functions of the ISF Power Management component is to provide time-keeping services for the BM because the BM timer resource is affected by the power settings. In normal operation, the BM uses the FXLC95000 MTIM timer to keep track of time and invoke active callbacks. The timer runs at normal speed in `ISF_POWER_NORMAL` and `ISF_POWER_LOW` power settings, and for these two power settings, the BM runs normally.

For ISF_POWER_LOWEST, however, using the $STOP_{SC}$ power mode causes the device oscillator to run at a reduced frequency. This means that the elapsed time per MTIM timer tick is no longer constant between ISF_POWER_NORMAL and ISF_POWER_LOWEST. In this case, the PM relies on the FXLC95000 frame timer to track time during $STOP_{SC}$. The frame timer is designed to operate correctly during both fast clock and slow clock modes. As a result, the Start-of-Frame (SOF) interrupt it produces can be reliably used to track time in increments of the configured frame rate even during periods of slow clock operation.

During slow clock mode, the BM's MTIM timer must be turned off and the PM's idle task is awakened at each SOF interrupt at which point the PM can perform some time-keeping activities, manipulate the frame timer interval to schedule its next wake-up and return to $STOP_{SC}$ mode. When the time remaining before the next BM event becomes small, the BM is re-enabled, the MTIM is reset and $STOP_{FC}$ is used for the remaining period allowing the BM to service its next scheduled MTIM interrupt.

As noted, the PM uses the frame timer and SOF interrupt for time-keeping in slow clock mode. In this mode, the PM programs the Frame Control & Status Register (FCSR) and the AFE Control & Status Register (AFE_CSR). If the AFE is setup for manual conversion and the FCSR frame control is set for ΦA followed by ΦD, then this combination inhibits SOF interrupt and causes the PM to enter into an infinite loop waiting for SOF to occur. More specifically, the register settings that can cause this problem are: AFE_CSR[0] = 1 (manual AFE conversion) with FCSR[5] = 1 (ΦA followed by ΦD). This problem can be avoided by not modifying the AFE_CSR and FCSR registers if the PM is used for power management. For additional details, refer to the SIM and AFE chapters of the FXLC95000CL HWRM listed in References.

3.6 Application Integration

An embedded application developer must perform these steps to integrate an application with the Command Interpreter (CI):

1. Create a callback function for each application that needs to communicate with the host. This function is responsible for handling commands sent from the host. The callback function prototype is defined in *isf_ci.h*. The commands that can be sent by the host are predefined as `ci_commands_enum` in *isf_ci.h*. The callback receives a host command packet (`ci_host_cmd_packet_t`) and returns a response status and a response response packet (`ci_app_resp_packet_t`).
2. Register the callback function with the Command Interpreter. Callback registration is done statically via the callback array, `ci_callback[]`. Because the application developer must modify this array, ISF requires this array to be defined in the user's code. The callback array contains both registered callbacks for internal ISF

applications as well as user applications. The ISF internal callbacks are placed in this array via the preprocessor macro `ISF_APP_CALLBACKS`. The user callbacks should be inserted following this macro. The Command Interpreter can manage a maximum of 31 callbacks in total. Because the ISF registers some callbacks, the first available callback ID for user applications will not be zero. ISF provides a preprocessor macro called `USER_APP_ID_START` which is the first available value for user callbacks. For more information, refer to *source/isf_ci.h*.

Note

Because ISF has internally registered callbacks, there will be fewer than 31 user application callback IDs available. In ISF 1.1, there are 29 callback IDs available for user applications starting at `USER_APP_ID_START = 2`.

The following is an example of user code used to define the callback array, registering two callback functions with the Command Interpreter.

```
// Application ID must start at USER_APP_ID_START.
enum sys_app_id_enum
{
  APP_ID_XYZ_MOTION           = (USER_APP_ID_START + 0),
  APP_ID_PORTRAIT_LANDSCAPE   = (USER_APP_ID_START + 1)
}
```

The application can make use of `isf_ci_app_read()` to read data from the host and `isf_ci_app_write()` to write data to the host.

The following example demonstrates a callback function for an application. This application allows the host to subscribe to the Quick-Read data and, as required, implements the `CI_CMD_UPDATE_QUICKREAD` command.

```
ci_response_t
isf_xyz_motion_callback(
ci_host_cmd_packet_t *pHost_packet,
ci_app_resp_packet_t *pApp_packet)
{

  // In this code example, xyz_array is a hypothetical data array
  // containing XYZ motion data.

  ci_response_t callback_ret = (ci_response_t)ISF_SUCCESS;

  switch(pHost_packet->cmd)
  {

    case CI_CMD_READ_APP_DATA:

    // Host requesting to read data.
    pAppPacket->bytes_xfer = (uint8)isf_ci_app_write(pHostPacket->appId,
        pHostPacket->byte_cnt, xyz_array);

    break;

    case CI_CMD_UPDATE_QUICKREAD:

      // Host has read this application's QR data. Call API to update.
```

```
        // Note that the host has alread configured the QR data for this
        // application. The CI will fill data to mailboxes that have been
        // designated as QR for this application with the specified offset.
        isf_ci_qr_update(pHostPacket->appId, MAX_QR_MOTION_DATA_BYTES, xyz_array);

        break;

    case CI_CMD_READ_CONFIG:

        // Host requesting to read data.
        pAppPacket->bytes_xfer = (uint8)isf_ci_app_write(pHostPacket->appId,
            pHostPacket->byte_cnt, config_array);

        break;

    case CI_CMD_WRITE_CONFIG:

        // Host writing data to application.
        pAppPacket->bytes_xfer = (uint8)isf_ci_app_read(pHostPacket->appId,
            pHostPacket->byte_cnt, config_array);
        break;

    default:

        // Unknown or invalid host command.
        callback_ret = CI_ERROR_COMMAND;
        break;

    }

    return callback_ret;
}
```

The following example demonstrates the implementation. The enumeration value for the application is added in the enumerator, `sys_app_id_enum`. The application's enumeration value has to be unique. Since ISF reserves values of application ID from 0 to some number, `USER_APP_ID_START - 1`, the first available application callback ID is `USER_APP_ID_START`. In the example, the application ID is `APP_ID_XYZ_MOTION` and it's callback function, `isf_xyz_motion_callback()`, is placed in the `ci_callback[]` after the `ISF_APP_CALLBACKS` macro.

```
//! Application ID must start at USER_APP_ID_START.
enum sys_app_id_enum
{

  APP_ID_XYZ_MOTION = (USER_APP_ID_START + 0),
  APP_ID_PORTRAIT_LANDSCAPE = (USER_APP_ID_START + 1),
  NUM_ISF_APP = (APP_ID_PORTRAIT_LANDSCAPE)
}

// Command interpreter call backs.
// The array index is the application id.

const ci_funcp_t ci_callback[NUM_ISF_APP] =
{

  // ISF callbacks come first.
  ISF_APP_CALLBACKS,

  // User callbacks come after the ISF callbacks.
  isf_xyz_motion_callback,
  isf_portrait_landscape_callback
};
```

3.6.1 Interrupt Output Integration

The FXLC95000 provides the ability to configure one of its pins as an interrupt output (INT_O). This may be used to provide the host processor with asynchronous event notifications. The Intelligent Sensing Framework provides the ability to send interrupt notifications to the host in conjunction with Command Interpreter events as well as the APIs to directly send interrupt notifications.

The following APIs are declared in `isf_ci.h`:

```
void isf_ci_enable_int_o(uint8 enable);

void isf_ci_set_int_o_polarity(uint8 pol);

void isf_ci_assert_int_o(void);
```

The ISF Command Interpreter can be configured to automatically send an INT_O event notification whenever response data from a Command/Response command is placed in the mailboxes. This is done on a system-wide basis such that the configuration call need only be called once to enable the functionality for all embedded applications.

`isf_ci_enable_int_o(1)` enables the automatic INT_O functionality.

To enable the automatic INT_O functionality, an application may call:

```
isf_ci_enable_int_o(1)
```

To disable the automatic INT_O functionality, an application may call:

```
isf_ci_enable_int_o(0)
```

The polarity of the interrupt signal sent is controlled via:

```
void isf_ci_set_int_o_polarity(uint8 pol)
```

where the interrupt is configured to be active high when `pol` is set to 1, and active low when `pol` is 0.

An application can trigger an interrupt immediately by directly calling:

```
void isf_ci_assert_int_o(void)
```

Appendix A
FXLC95000 Non-Maskable Interrupt (NMI) Implementation

A.1 Implementation Steps for Handling Non-Maskable Interrupts (NMIs)

For applications that need to handle NMIs, the general steps are described here. For a detailed implementation example, see Sample Non-Maskable Interrupt (NMI) Implementation.

1. Include the bsp.h header file in the source code file.

```
#include <bsp.h>
```

2. In the initalization code of the project, clear all the NMI sources.

```
_clear_all_nmi_source();
```

3. In the SWI6 interrupt service routine (ISR), use the provided APIs to determine the source of the NMI and clear it.

```
// Checking for INT NMI.
  if (_get_nmi_source() & INT_NMI_BIT_MASK)
  {
    // TODO: Handle the INT interrupt.

    _clear_nmi_source(INT_NMI_BIT);
  }

  // Checking for Frame Error NMI.
  if (_get_nmi_source() & FRAMEERR_NMI_BIT_MASK)
  {
    // TODO: Handle the Frame Error interrupt.

    _clear_nmi_source(FRAMEERR_NMI_BIT);
  }

  // Checking for SWI7 NMI.
  if (_get_nmi_source() & SW7_NMI_BIT_MASK)
  {
    // TODO: Handle the SWI7 interrupt.
```

```
    _clear_nmi_source(SW7_NMI_BIT);
  }
```

A.2 Sample Non-Maskable Interrupt (NMI) Implementation

The following example code demonstrates the handling of an NMI.

1. `Main_task()` is created by the Freescale MQX™ RTOS kernel and runs. The initialization process installs a SWI6 ISR.

```
// Install SWI6 ISR callback "swi6_isr".
  if (_int_install_isr(VectorNumber_VL6swi, (void (_CODE_PTR_)(pointer))swi6_isr, NULL)
== NULL)
    return;
```

   Following intialization, it sets the event flag `EVENT_SYS_INIT_DONE`.

2. `NMI_task()` is created and blocks on the `EVENT_SYS_INIT_DONE` event. When `Main_task()` sets the `EVENT_SYS_INIT_DONE` event flag, it unblocks and then enters a test loop. This loop generates a SWI7 interrupt and then blocks on any of the following NMI events:
   - `EVENT_NMI_INT`
   - `EVENT_NMI_FRAMEERR`
   - `EVENT_NMI_SW7`

```
INTC_SFRC = 0x38; // Generate SWI7 interrupt.

// Block until NMI event occurs.
_lwevent_wait_for(&gEvent_nmi, (EVENT_NMI_INT | EVENT_NMI_FRAMEERR | EVENT_NMI_SW7),
FALSE, 0);
```

3. As a result of the generated SWI7 interrupt, `swi6_isr()` is called. Each possible NMI source is checked using `_get_nmi_source()`. For each NMI source set, the code sets an NMI event flag and clears that NMI source with `_clear_nmi_source()`. An example code segment for the case of SWI7 NMI is given here.

```
nmi_src = _get_nmi_source();  // Get the source(s) of the NMI interrupt.

if (nmi_src & SW7_NMI_BIT_MASK)
{
  // TODO: Service NMI here and/or block on event in task to do further processing.

  _lwevent_set(&gEvent_nmi, EVENT_NMI_SW7);
  _clear_nmi_source(SW7_NMI_BIT); // Only one source can be cleared at a time.
}
```

4. When `swi6_isr()` completes, `NMI_task()` unblocks, because the NMI event flag is set. `NMI_task()` calls `_lwevent_get_signalled()` to determine which event flag is set, and processes that event. In this template, for each NMI event, a counter is incremented.

```
// Get the event(s) flags that unblocked us.
event_signalled = _lwevent_get_signalled();
```

```
if (event_signalled & EVENT_NMI_INT)
  ++gIntNmiCnt;

if (event_signalled & EVENT_NMI_FRAMEERR)
  ++gFrameErrCnt;

if (event_signalled & EVENT_NMI_SW7)
  // As a result of the SWI7 generated above, this counter gets incremented
  // as a result.
  ++gSw7Cnt;
```

A.3 Revision History

Revision number	Revision date	Description
1.0	2/2014	Initial release of document for Xtrinsic Intelligent Sensing Framework (ISF) Software version 1.1

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number
ISF1P1_95K_SW_REFERENCE_RM
Revision 1.0, 2/2014