NXP Semiconductors
REFERENCE MANUAL

Document Number: ISF2P2_KINETIS_SWRM
Rev. 1.0, 2/2016

Intelligent Sensing Framework v2.2 Software Reference Manual

For the Kinetis Family of Microcontrollers

Contents

1. About this document

1.1 Purpose

This reference manual describes the features, architecture, and programming model of the Intelligent Sensing Framework (ISF) embedded middleware, release v2.2. This software is designed to execute on the vast majority of the Kinetis family of microcontrollers supplied by NXP to easily obtain sensor data. The framework is supported by a set of cooperative Processor Expert (PEx) components that automatically generate the framework code, embedded application code, as well as dependent drivers for a variety of internal hardware components available on Kinetis. Processor Expert technology also generates the real-time operating system (RTOS) required by ISF. This document focuses on the core ISF functionality and its use of PEx technology to build custom, embedded sensor applications. Additional information is available in the ISF v2.2 API Reference Manual, the ISF v2.2 User Guide and the ISF v2.2 Release Notes available at *nxp.com/ISF-2.2-KINETIS*.

1.2 Audience

This document is primarily for system architects and software application developers currently using or considering using the ISF v2.2 middleware on the Kinetis family of microcontrollers as the basis for an intelligent sensor system.

1.3 Terminology and conventions

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

Table 1. List of technical terms

Term	Definition
application 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 AppID or *callback ID* may be used.
callback	See *callback function*.
callback ID	See *application 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.
component	A collection of files implementing the Processor Expert Macro-processor Command Language and designed to automatically generate code based on high level configuration properties assigned by a developer.
DeviceHandle	A handle identifying the device used for Device Messaging transactions.
Digital Sensor Abstraction	Abstraction layer to enable communications with multiple types of sensors.

Term	Definition
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 that is run as part of the Intelligent Sensing Framework.
end-user product	A third-party product that hosts a sensing subsystem.
event group	A 32-bit group of event bits used to let tasks synchronize and communicate.
FIFO	First-In, First-Out; a method of processing and retrieving data
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 predefined set of input parameters that performs a series of instructions and returns a predefined set of output values. A function may be invoked from one or more points in an executable program.
host application	A program that executes on the host processor.
Intelligent Sensing Framework (ISF)	The NXP-provided software middleware layer enabling the development of custom embedded sensor applications with increased portability, 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*.
Kinetis	A family of ARM®-based microcontrollers offered by NXP.
period	The time between successive repetitions of a given phenomenon. Period is equal to the inverse of frequency.
PEx component	Processor Expert Component; see component.
platform	The combination of the device and firmware. Also referred to as *intelligent sensing platform*.
protocol adapter	A uniform interface to all communications channels in conjunction with Device Messaging. There is a Protocol Adapter for each type of communication channel, for example: I^2C, SPI, and UART Protocol Adapters.
sensor adapter	A Sensor Adapter implements the Digital Sensor Abstraction interface for a particular physical sensor and handles the device-specific communications and interactions with the physical sensor 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.
stream ID	Identifier for the Stream protocol data
system	The platform and external sensor hardware that interact together via hardware and software protocols. Also referred to as *intelligent sensor system*.
task	An operating entity within the Intelligent Sensing Framework (ISF) scheduled to execute by the OS Abstraction layer and underlying RTOS. A task may entail the execution of one or more functions.
transport	Communications mechanism. Examples: I^2C, SPI, Bluetooth©, Ethernet, and USB

Table 2. List of abbreviations

Term	Definition
6LoWPAN	Low power Wireless Personal Area Network
ADC	Analog-to-Digital Converter
API	Application Programming Interface
ARM	Any of several 32-bit RISC microprocessors that use ARM® instruction set architectures
BM	Bus Manager
CCITT	Consultative Committee for International Telephony and Telegraphy
CI	Command Interpreter
CMD	Command
COCO	Command Complete (software)
CRC	Cyclic Redundancy Check
DM	Device Messaging
DSA	Digital Sensor Abstraction
EA	Embedded Application
FreeRTOS	An open-source, real-time operating system with multitasking kernel for resource-limited MCUs.
HDLC	High-Level Data Link Control Protocol
I^2C	bi-directional, two-wire, serial communication bus
ISF	Intelligent Sensing Framework
ISR	Interrupt Service Routine
KDS	Kinetis Design Studio
KSDK	Kinetis Software Development Kit
MQX	A real-time operating system with multitasking kernel for resource-limited MCUs.
NOP	No Operation Instruction
OSA	Operating System Abstraction
PEx	Processor Expert
PIT	Programmable Interval Timer
POSIX	Portable Operating System Interface; IEEE standard for maintaining compatibility between operating systems
RTOS	Real-time Operating System
SDK	Software Developer Kit
SP	Stream Protocol
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
ZigBee	Network specification using IEEE 802.15.4 wireless standard for low-power, wireless, local area networks

1.3.1 Notational conventions

Notation	Description
0b	Prefix to denote a binary number
0x	Prefix to denote a hexadecimal number
byte	An 8-bit data unit
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 that may represent command names, defined macros, constants, enumeration values are shown as, for example, `CI_ERROR_NONE`
word	A 16-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.

2. Introduction

2.1 System overview

The Intelligent Sensing Framework (ISF) is designed to be incorporated into integrated sensing applications executing on the vast majority of the Kinetis family of ARM-based microcontrollers[1]. The Kinetis family includes several hundred variations of ARM® cores, memory configurations, and integrated peripherals, the major categories of the Kinetis family are illustrated in Figure 1. For more information on the Kinetis family of microcontrollers, see *nxp.com/kinetis*.

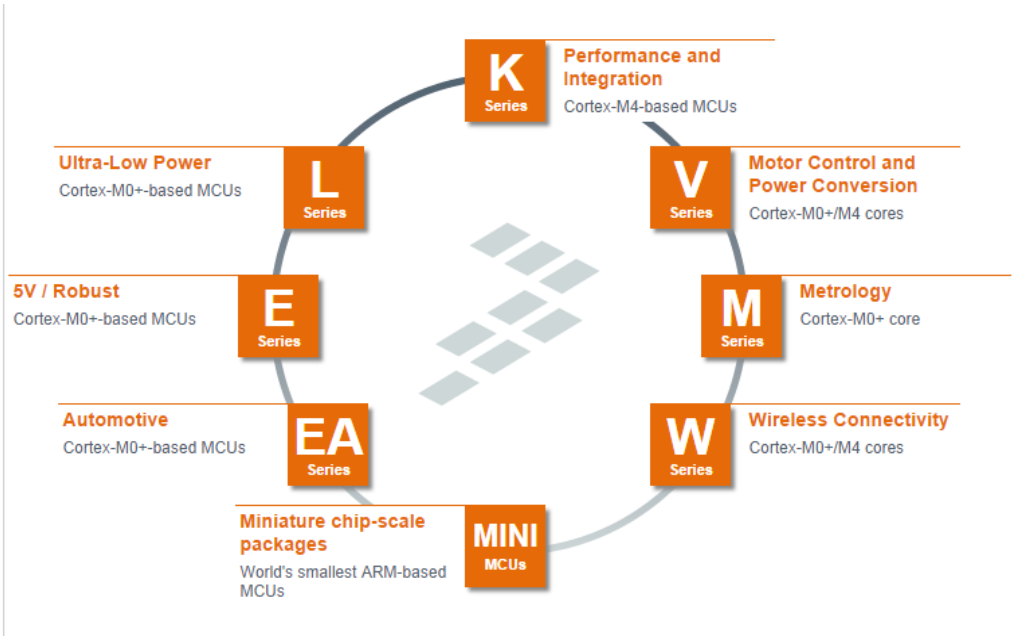


Figure 1. Kinetis MCU product portfolio

Combined with the configurability of Processor Expert (PEx) technology, ISF is a portable, easy-to-use, embedded-development and runtime framework that supports typical embedded sensor use cases as well as custom designs. ISF uses an abstract interface and adapter patterns to provide extensibility in supporting multiple sensors and sensor types as well as multiple communications protocols such as Master I^2C, Master SPI and UART.

The ISF comprises a set of source-level files generated by PEx technology that supports host communications, sensor management, and periodic interval scheduling. Wherever possible, ISF uses components supplied with PEx or included in the Kinetis Software Development Kit (KSDK) to abstract hardware-specific peripherals and operating system services such as timers, as well as I^2C, SPI and Serial (UART) device drivers.

Figure 2 is a static stack diagram that shows the high-level relationship between a customer's embedded application and the underlying framework services, generated drivers, and the hardware target.

1. ISF v2.1 supports sensors requiring the Kinetis E Series platforms supported by Logical Device Drivers in Processor Expert. ISF v1.1 supports the FXLC95000 Motion Sensing Platform with built-in 32-bit ColdFire microcontroller and 3-axis digital accelerometer.

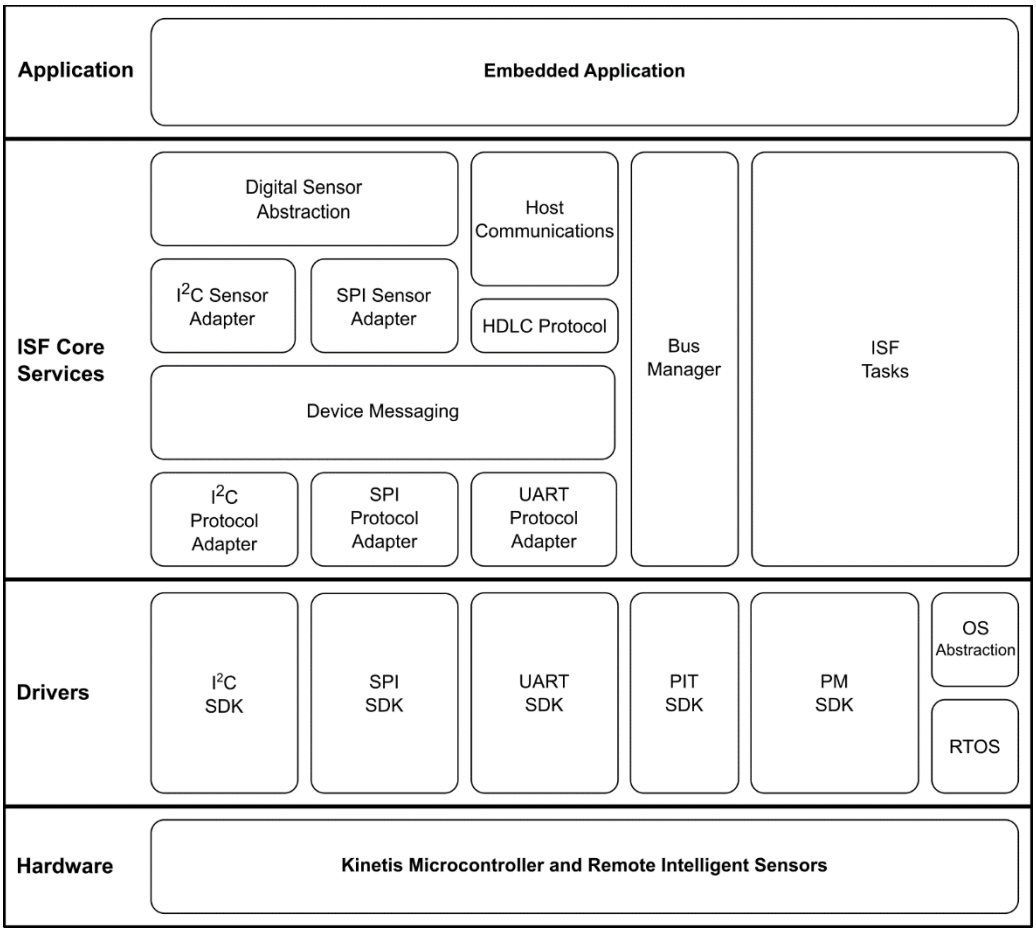


Figure 2. ISF architecture

The ISF system components that provide the functionality required for developing sensor applications, are as follows:

- **Device hardware**: The Kinetis family of microcontrollers provide several hundred different individual configurations of ARM cores, memory, and integrated peripherals. ISF is designed to run on the vast majority of the Kinetis family. ISF depends on a very small subset of the integrated hardware peripherals including a System Tick counter, a Programmable Interval Timer (PIT), any of the I2C, SPI, or UART/Serial interfaces (optional), and various interrupt/GPIO pins. For analog sensors, an Analog-to-Digital Converter (ADC) can be used to acquire sensor outputs.
- **ISF:** The Intelligent Sensor Framework (ISF) provides embedded applications the capability to subscribe to external sensor data and read such data at various rates. It also supports communication between the host processor and the application via a UART/Serial interface. ISF allows the Kinetis microcontroller to act as a sensor hub for external sensors and to manage that data for the host processor.
- **RTOS:** The NXP KSDK Operating System Abstraction (OSA) feature supports multiple Real Time Operating Systems (RTOSs). ISF v2.2 has been tested with MQX and FreeRTOS. Each RTOS is generated by Processor Expert as runtime functions that provide real-time, multitasking capabilities to embedded applications.

2.2 Development environment

ISF v2.2 was developed for Kinetis Design Studio v3.0 with integrated Processor Expert technology. ISF can target a variety of standard, prototype, and production hardware environments and is released on the NXP Freedom Development Platform family of boards.

One example of a target system for the ISF middleware framework is shown in Figure 3. This figure shows the Kinetis Freedom Board along with the FRDM-FXS-MULTI-2B. When connected to a development host platform, this configuration provides a complete prototyping environment for sensor applications.

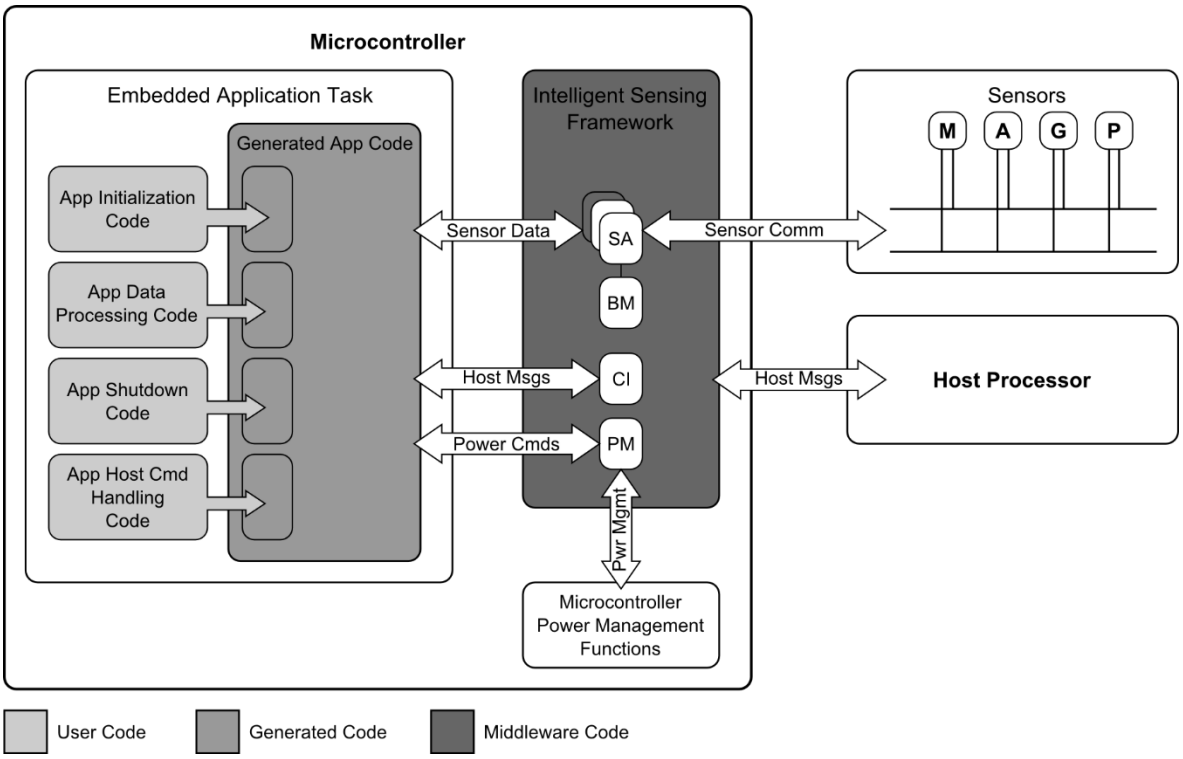


Figure 3. Block diagram of Kinetis and sensor development environment with ISF

3. Intelligent Sensing Framework

This section describes the Intelligent Sensing Framework architecture, the Processor Expert Technology components that support it, and a high level description of the services. This section also describes the details of the default Embedded Application generated by its own PEx component. Finally, it describes the usage of the OS Abstraction also generated within the PEx development environments.

The ISF uses a set of PEx components that automatically generate the ISF Core code, Communication Channels, and Sensor Interfaces based on high-level configuration properties. Wherever possible, the ISF conforms to existing PEx components and KSDK driver interfaces create driver level software for the Kinetis microcontrollers.

3.1 ISF theory of operation

The ISF v2.2 is designed to be used during the early stages of the product development cycle. This includes sensor-based products requiring access to real-time sensor data and/or processing by an intermediate microcontroller acting as both a sensor hub and an application layer. The microcontroller executes the ISF embedded middleware along with a real-time operating system (currently MQX and FreeRTOS) and application-level code supplied by the embedded application developer. ISF can support a variety of products with a wide range of resources and configurations. The framework is tested and released to run on NXP's line of development boards called Freedom Development Platforms. As the developer's project progresses, the framework can be seamlessly ported to many selected Kinetis microcontrollers, with minimal effort.

In the earliest stage of product development, exploration of the capabilities of a variety of microcontrollers and sensors is of the most importance to developers. ISF with PEx allows the developer to quickly explore the capabilities of all supported sensors, using the PEx components to modify the operation of individual sensors at a register level. ISF allows the application developer to send raw sensor output to the host processor without writing a single line of code for the embedded platform. This stage is also an ideal period to assess the capabilities of the ISF middleware for fit in the final product design.

During the prototyping stage of product development, the target environment is rendered using as much off-the-shelf software and hardware as possible in order to reduce the overall development time. One approach is to integrate NXP's Freedom boards into the prototype using the Arduino-compatible interface connectors as a bridge into the prototype platform. This approach allows the developer to leverage the ISF to the maximum extent possible, and to develop software solely at the application layer, during the prototyping phase. Typical goals include reduced system cost, leveraging memory and peripheral options, CPU speed, and interface choices can be easily evaluated using this approach.

3.2 ISF architecture

The ISF v2.2 architecture has been designed to provide rapid generation of embedded application code and customized middleware configurations, based on a set of high-level configuration properties applied to a set of PEx software components. These Processor Expert components relate to each other based on a specific hierarchy. This hierarchy aids the embedded application developer by eliminating omissions that can lead to runtime errors in the system. In addition, PEx technology can error-check and cross-check property settings between the components to guarantee consistency ahead of actually building the application.

The services included in the ISF component are only brought into the application as needed. This allows the user to tailor the features of ISF to their specific application and resource environment.

ISF v2.2 is generated by PEx technology, based on the configuration established by the developer. The software is constructed in layers and uses the Kinetis SDK Drivers, as is practical. The bottom layer of this hierarchy is the configuration of Kinetis microcontroller, communications interfaces, and remote sensors desired by the application developer. The Driver layer consists of existing PEx components (KSDK) that provide the driver-level interfaces to internal hardware peripherals, inside the Kinetis microcontroller. The ISF Services layer is the set of PEx-generated software that provides the services described in the following sections.

Uniform interfaces allow applications to access physical data, measured by the remote sensors. The ISF Core, acting as a server, provides sensor data to registered applications, acting as clients needing that sensor data. The sensor data is applied to the application at various rates and formats.

Intelligent Sensing Framework

The Host Communications service provides the ability to pass data into and out of the Intelligent Sensing Framework via the UART/Serial interface. The Host Interface supports a proprietary, command/response protocol with a set of pre-defined commands. The Host service enables the embedded application developer to extend the command set. This service also provides a flexible, asynchronous data streaming protocol.

The Device Messaging and Protocol Adapter services provide a uniform interface to all communications channels (I^2C, SPI, UART). The Protocol Adaptor layer for each underlying protocol allows the service to run on top of the PEx KSDK drivers, without modification.

The MQX or FreeRTOS operating systems are generated at a source level through the OS Abstraction PEx component. They provide lightweight, real-time scheduling, intertask events, resource semaphores, interrupts, stack and heap management.

Software components from the above-mentioned service families are packaged into libraries that target the various ARM core types in the Kinetis family. The ISF Core packages several software components to form the base set of functionality. Additional functionality is provided by ISF PEx components that generate source code based on the desired developer configuration of each of the component's properties.

The ISF Embedded Application relies on the ISF Core Services for its operation. The Embedded Application is either partially generated by PEx or fully written by the developer. Figure 4 shows a high-level view of the services required by the application.

The application statically registers a callback function with the Host Interface, at compile time. The Host Interface uses the callback to execute commands addressed to the application. The application also uses the Host Interface to send asynchronous packets to the Host. The application establishes the power management scheme to be used, which may also be changed via the Host Interface. The Embedded Application interaction with sensor interfaces is through the DSA-Direct Interface.

The DSA-Direct interface allows the application to control the sensor directly. This method restricts the developer to a single application interacting with the sensors and a single rate/format for sensor data. The Sensor Adapters use the Bus Manager (BM) to create highly accurate, timed callbacks in order to schedule periodic reads of the sensor data. The Bus Manager can also be used by the application to schedule timing events when accuracy down to a few microseconds is required.

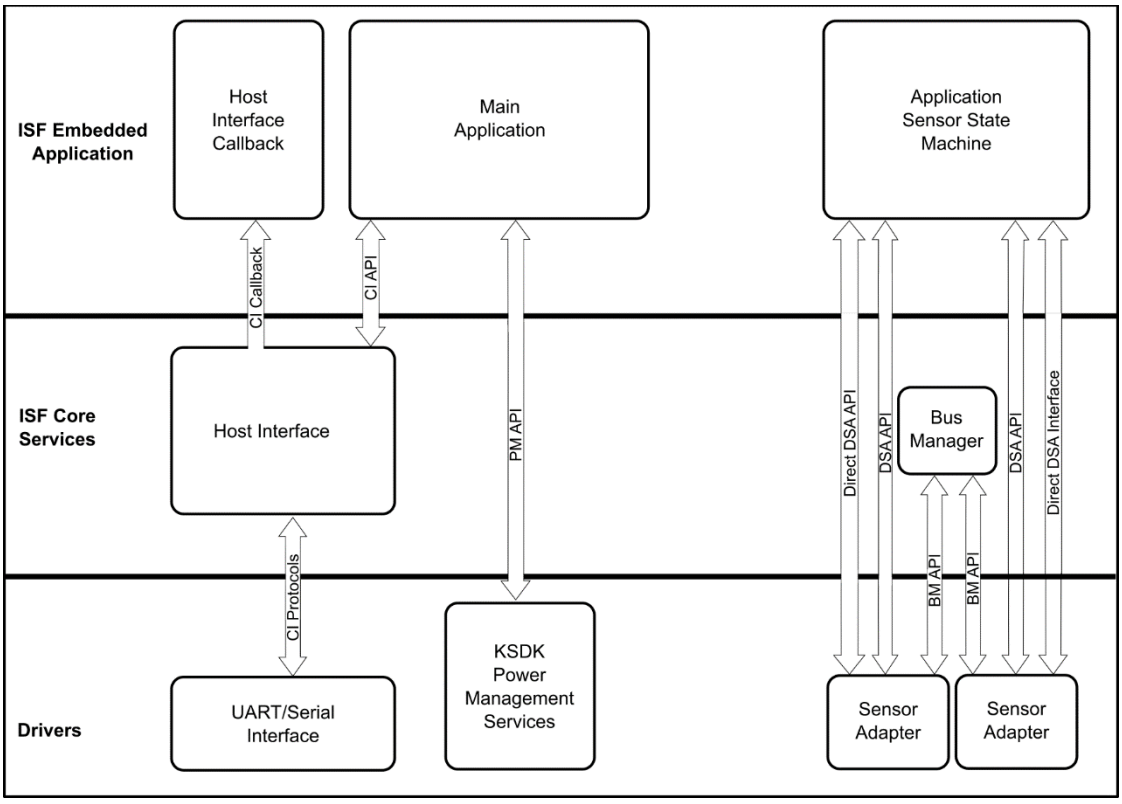


Figure 4. ISF embedded application using ISF core services

3.3 Processor Expert component architecture

ISF v2.2 includes a set of Processor Expert components that work together to generate source code and configure the system to meet the developer's application and hardware requirements. The developer configures two components to start: *ISF_KSDK_Core* and *ISF_KSDK_EmbApp*. Depending upon the services, communications channels, and sensors selected in these components, PEx automatically incorporates additional components that also may be configured. Once a complete set of components is instantiated in the project, code generation can be initiated. The following sections explain the ISF PEx components in more detail.

The *ISF_KSDK_Core* component allows the developer to include from the ISF Core services [Bus Manager (BM) and Command Interpreter (CI)] the desired optional services for the specific Cortex family, for example, Cortex M0+, M4, M4F. In resource-constrained systems, some or all of these services may be excluded in order to save memory or processing cycles. *ISF_KSDK_Core* also provides the developer with the ability to select from a list of supported sensors, the necessary remote sensors to be included in the system.

The *ISF_KSDK_EmbApp* is an optional component that provides the developer with a pre-compiled structure to extract and process raw sensor data from a set of sensors. The component includes a list of subscriptions to types of sensors including accelerometers, magnetometers, gyroscopes, pressure and temperature sensors. The component displays a list of sensor interfaces that match the selected type and correspond to physical sensors in the system. In addition, the *ISF_KSDK_EmbApp* component allows the developer to configure the Host Interface and extend the default set of the host commands.

The *ISF_KSDK_Protocol_Adapter* component provides a collection of interface adapters (I^2C, SPI, UART) along with the Device Messaging services needed. The component automatically configures the desired protocol and instantiates the underlying KSDK drivers. This component is not selected by the developer directly, but is incorporated, based on the Sensor and Host Interface configuration in *ISF_KSDK_Core*.

ISF_KSDK_Sensor_Adapter_Interface is a collection of components that select individual sensor interface components implemented in the ISF system. The component provides for grouping of sensors based on function and type. It also generates the sensor-specific configuration of instantiated sensors in the system. Finally, it generates the global list of sensors used during runtime to access and control sensors.

The *ISF_KSDK_Bus_Manager* component allows the developer to select the specific PIT hardware to be used for the application callback, event timing.

3.4 Core framework component details

3.4.1 Theory of operation overview

ISF is separated into the *ISF Core* component and the *Embedded Application* component. While these components are designed to work together, the ISF Core can be used as a standalone to create the framework, sensor interfaces, and communication channels. ISF Core can then be used with a freeform application, via the exposed runtime APIs.

After creating a project with PEx technology, the developer can import the *ISF_KSDK_Core* component from the PEx component library. This component allows the developer to select the core features of ISF to be included. The *ISF_KSDK_Core* automatically instantiates various other components, based on the developer-selected features. In a fully configured system, the *ISF_KSDK_Core* automatically creates an *ISF_KSDK_Bus_Manager*, *ISF_KSDK_Protocol_Adapter*, and several RTOS tasks.

Once the features of the ISF Core have been selected, the developer can then select the sensors to be included in the system. Each sensor has its own, specific, PEx component. These allow the developer to select or create the desired communication channel interface and to set the sensor-specific, register-level configuration, if necessary.

After the *ISF_KSDK_Core* component has been configured, the developer may use the code generation feature to create the ISF framework.

3.4.2 Framework overview

The Intelligent Sensing Framework is completely generated by the selected Processor Expert (PEx) components at a source code level. Only the necessary components are generated based on the configuration of the components. The optional *ISF_KSDK_EmbApp* component is included to generate an application layer, which can be extended by the customer's application specific code.

3.4.3 Processor Expert component overview

The ISF Core components include the *ISF_KSDK_Core*, I*SF_KSDK_Protocol_Adapter*, I*SF_KSDK_CommChannels*, *ISF_KSDK_Bus_Manager*, and a set of *ISF_KSDK_Sensor_Adapter_Interface* components. Together these components create the baseline core functionality of the ISF.

Figure 5 and subsequent sections explain the Core Component relationships, their underlying functionality, and use cases for including/excluding components.

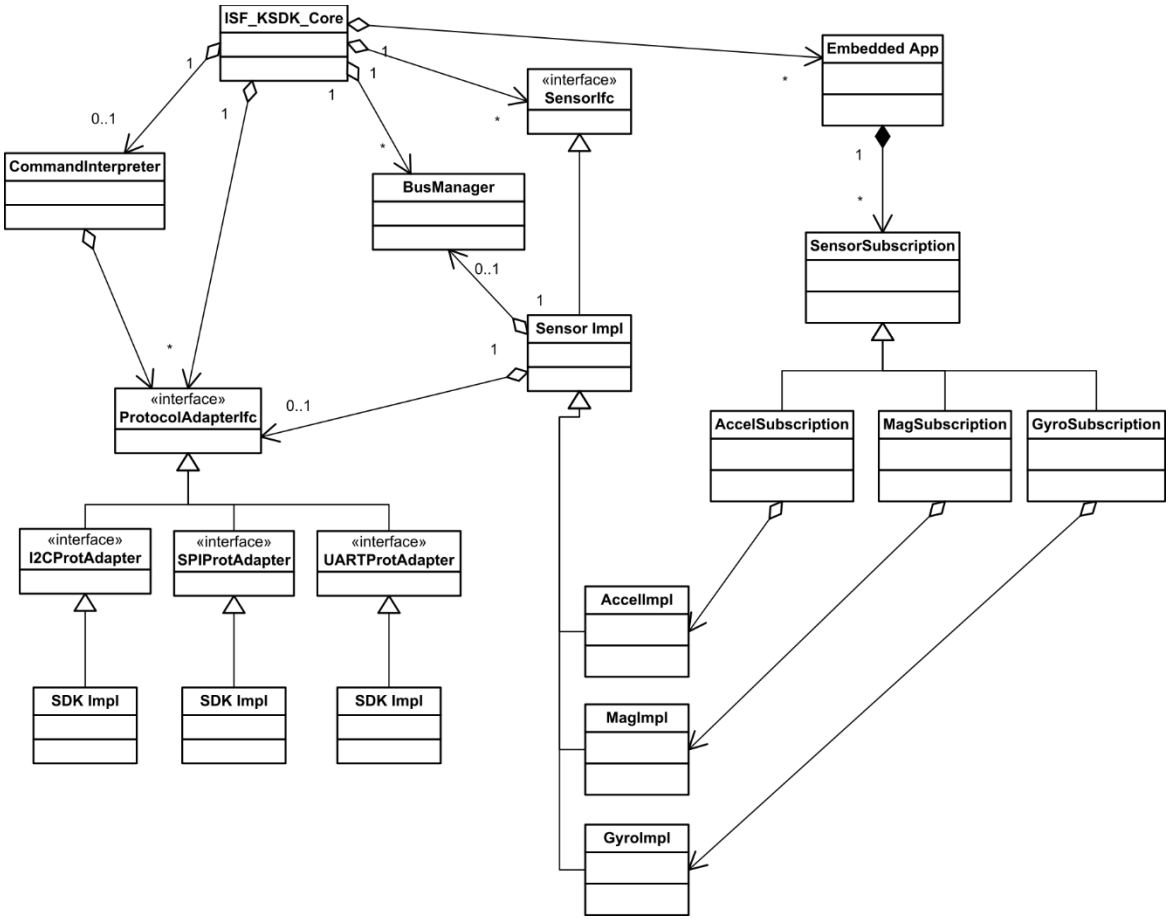


Figure 5. ISF component relationships

3.4.4 Digital Sensor Abstraction (DSA)

3.4.4.1 Theory of operation

The Digital Sensor Abstraction (DSA) is used to expose a standard interface to sensor command and control functionality while maintaining a sensor-specific implementation. The DSA defines interfaces to initialize, configure, start, stop, and shutdown a sensor, to validate sensor settings and to convert native sensor sample data to standard sensor types.

The set of functions implementing these functions for a given sensor is known as a *Sensor Adapter*. The architecture enables the embedded application developer to write new Sensor Adapters and to associate these adapters with existing or new sensors connected to the platform.

The list of the sensors available is maintained in a global list. This list associates each instance of a sensor in the system with a system-unique Sensor ID, a Sensor Adapter, and other specific instance data needed to uniquely address the sensor. The API enables its users to refer to sensors via their assigned Sensor ID when subscribing. Internally, the provided Sensor ID is used to either lookup the sensor configuration information contained in the System Sensor Configuration list or to invoke the appropriate Sensor Adapter functions.

The Digital Sensor Abstraction (DSA) adapter functions to interact and manage its sensors. The Sensor Adapter functions are designed to allow multiple sensor instances of a particular type to all reference the same Sensor 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.

Refer to Figure 5 and Figure 6 to understand the interface between the Sensor Adapter and the application. To prepare to receive sensor data, a software FIFO is created to hold a sample set from each sensor. An event flag is used to signal when new sensor data is available. ISF validates the request parameters with the applicable Sensor Adapter and then configures the sensor via the DSA Sensor Configure interface. The adapter in turn configures the sensor hardware to provide samples at the specified rate. For a Sensor Adapter that polls, a Bus Manager callback is configured to read the sensor data at the specified interval. The Bus Manager invokes its registered callback at the specified intervals. When the adapter's callback is executed, it completes a Device Messaging read call to examine the physical sensor's output registers. Once the read completes, the adapter places the new samples in the FIFO, waits for the FIFO to fill and sets an event flag to signal that new samples are available. Registered subscribers are then notified via their event flags. Upon notification, each embedded application can call the `isf_fifo_el_traverse` method to retrieve the data.

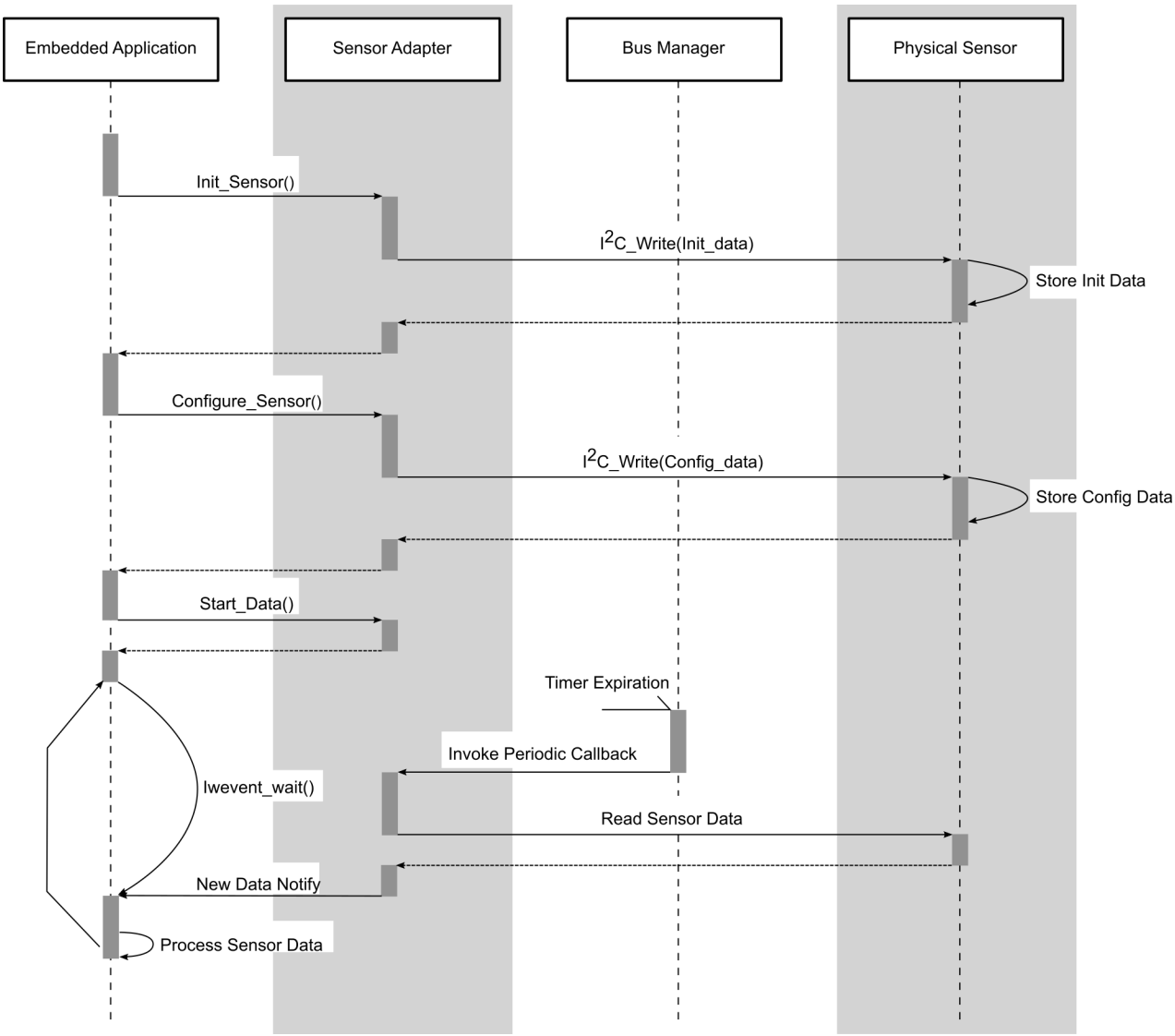


Figure 6. Digital sensor abstraction calling sequence

3.4.4.2 DSA module design

The majority of the sensor-specific software used by ISF v2.2 is included in modules that conform to the DSA interface and implement a reentrant interface to a specific sensor. Besides implementing the sensor interface abstraction, each module also uses register-level configuration information, created by its associated PEx component, to configure the underlying sensor. The sensor-specific module is included only once in the project, based on the selection of the sensor in the *ISF_KSDK_Core* component's System Sensor Configuration.

3.4.4.3 DSA Processor Expert component design

Figure 7 shows the component hierarchy specific to the System Sensor Configuration in the *ISF_KSDK_Core*. The System Sensor Configuration is a list that collects the sensors used in the system. The System Sensor Configuration is aware of all the available sensor types currently supported in the ISF. The developer updates the list to include the desired set of sensors based on those available in the target system. Each type of sensor is classified into generic types: accelerometers, magnetometers, gyroscopes, pressure, orientation, and temperature. This allows the PEx system to present the embedded application with alternative sensors that are interface-equivalent at the application level and can be integrated without changes to the application.

As shown in Figure 7, this is done through the abstract *ISF_KSDK_Sensor_Adapter_Interface* and *ISF_KSDK_Sensor_<Part Number>_<Sensor Type>* PEx components. The specific sensor instance is then created by the sensor-specific components for example, *ISF_KSDK_Sensor_MMA865x_Accelerometer*. These components generate the static configuration data structures for the desired instance of the sensor. They further define the DSA interface for this sensor instance and map the function calls to the sensor specific module's functions.

The sensor-specific components also include properties that expose the register-level interfaces to the sensor. This feature allows advanced embedded application developers to modify specific features of a sensor instance statically if desired. Default values for all of these properties are supplied and conform to the sensor default values as specified in their Data Sheets.

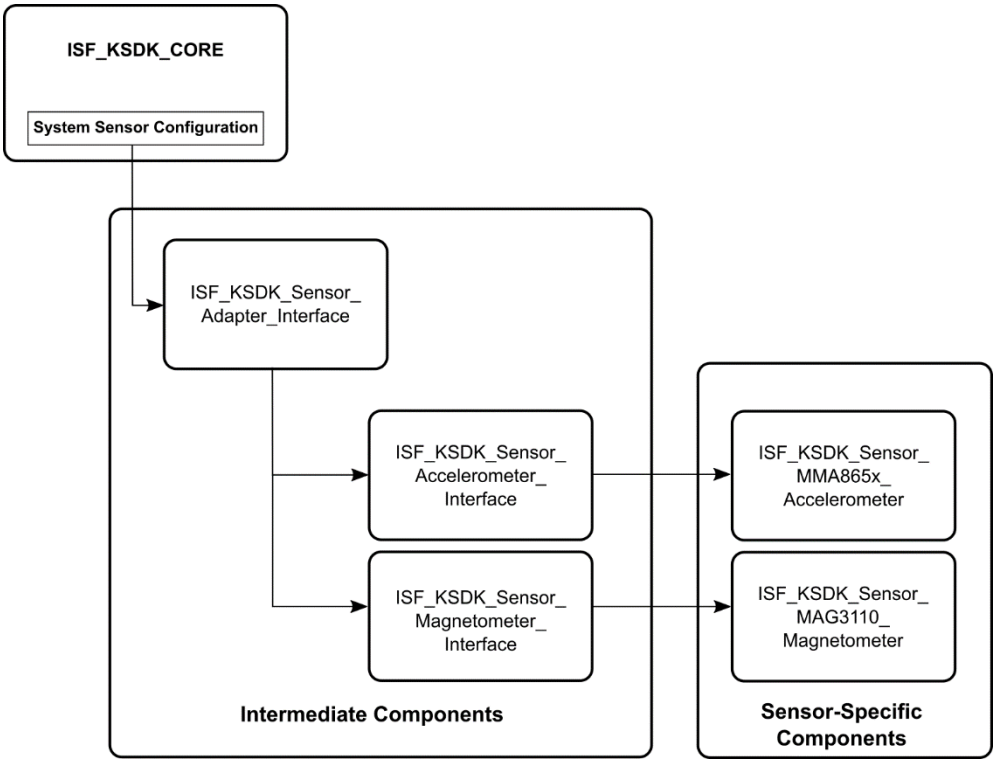


Figure 7. ISF v2.2 component hierarchy

3.4.5 DSA-Direct interface

3.4.5.1 Theory of operation

As described in Section 4.4.4.1, the DSA Interface allows applications to subscribe to raw sensor data. The DSA-Direct interface provides a simplified convenience wrapper around the direct Digital Sensor Abstraction API.

3.4.5.2 DSA-Direct module design

The DSA-Direct interface is a functional API that provides initialization, configuration, operational control (start/stop) and sensor data conversion routines. The DSA-Direct function calls perform error checking and global data structure updates. It makes direct calls to the corresponding sensor's DSA interface functions via the adapter function call interface.

3.4.5.3 Generic sensor types and standard sensor data types

ISF v2.2 supports Generic Sensor Types and Standard Sensor Data Types as part of the subscription interface to the sensor. This step towards standardization for the sensor interfaces continues to preserve the option of the native sensor data format.

The ISF supports the following types of sensors:

- Accelerometer (1 to 3 Axes)
- Analog Sensors other than 5 V (1 to 3 Axes)
- Gyroscope (1 to 3 Axes)
- Magnetometer (1 to 3 Axes)
- Orientation (Up to 10 axes)
- Pedometer Sensor
- Pressure Sensor (1 Axis)
- Temperature Sensor (1 axis)

Each sensor supports its native output data format along with converted sensor output fixed-point and floating-point formats in standard engineering units for example, magnetic field strength in microTeslas (μT) for magnetometers.

Each Sensor Adapter includes a conversion routine that scales the native format to the desired standard format based on the application subscription parameters: resultType and resultFormat.

3.4.6 Bus Manager

3.4.6.1 Theory of operation

The *ISF_KSDK_Bus_Manager* provides a highly accurate, timed, callback service with a resolution down to 1 μsec. Embedded Applications or Sensor Adapters may use the Bus Manager (BM) services to create periodic callbacks at the specified interval[2].

The Bus Manager uses one of the Periodic Interval Timers (PIT) internal to the Kinetis microcontroller. The PIT was chosen because its interval may be loaded while the previous interval is executing on the timer. Figure 8 shows the behavior of the PIT when the **PIT_LDVAL** register is modified while the timer is actively running. The PIT is a countdown timer that generates an interrupt as the count reaches zero. A value loaded into the **PIT_LDVAL** register takes effect at the next interrupt (zero). The BM design relies on the ability to keep the PIT pipeline constantly fed with the next expected interval.

While the actual timed interval is very accurate, the RTOS interrupt handling and *ISF_KSDK_Bus_Manager* service itself introduces some delay between when the timer actually fires and when the registered callback is invoked. Jitter between successive callback invocations is generally low but can be affected by other interrupts including processing of messages from the host application as well as preemption by higher-priority tasks in the system. Applications requiring extreme accuracy must therefore use direct interrupts and/or DMA services.

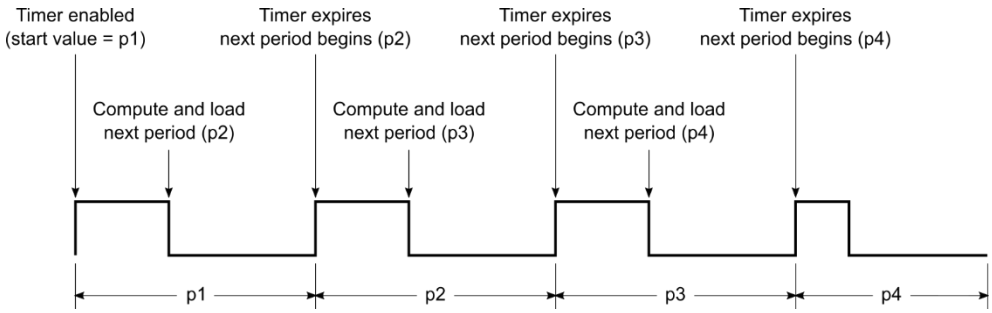


Figure 8. Dynamically setting PIT new load value

3.4.6.2 Bus Manager module design

Figure 9 shows the functional structure of the Bus Manager module. The Embedded Application uses the `Bus Manager` API to first register the periodic callback, providing the function pointer for the callback and the desired period. Next, the operation of the timed callback is started and stopped as required through separate API calls. Once the service is no longer required, the application calls the unregister function.

[2] The Bus Manager services may also be used to create one-shot timing events. In this scenario, the `bm_start()` function may be called to schedule a callback. Inside the callback, the user needs to call `bm_stop()` in order to terminate the callback after a single execution.

The Bus Manager functionality is divided between the BM Task, KSDK PIT driver, and BM Interrupt Service Routine (ISR). The BM Task uses the KSDK PIT driver to initialize and set the period of the PIT timer, based on the complete set of subscribed callbacks. Each interval is determined by examining the list of registered and active callbacks and determining the next closest interval to be scheduled. When an interval is complete, the PIT interrupt is routed to the BM ISR. The ISR signal reloads the PIT timer and sends the events associated with the last interval to the BM Task. The BM Task waits on events indicating that callbacks are pending and sequentially calls the registered callback functions.

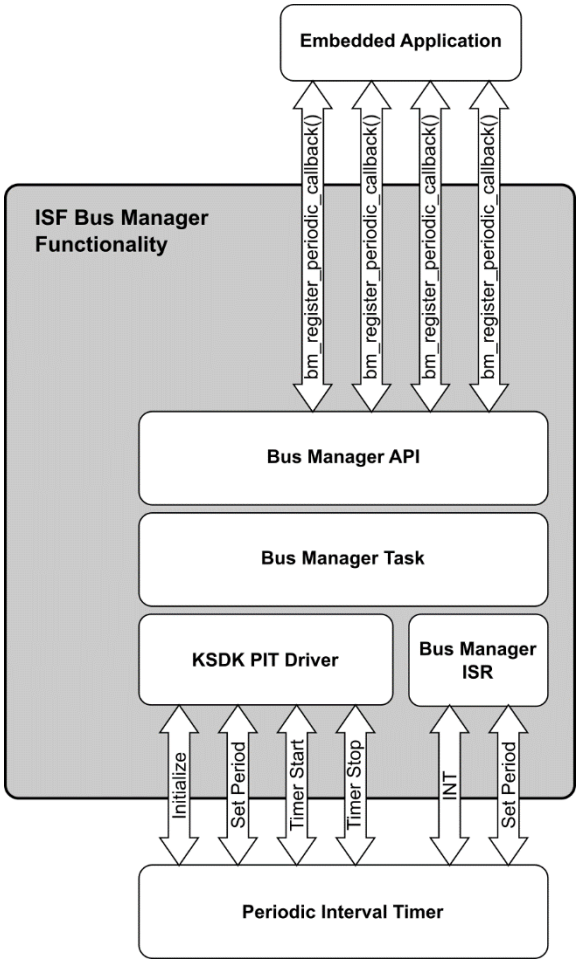


Figure 9. ISF Bus Manager hierarchy

3.4.6.3 Bus Manager Processor Expert component design

The *ISF_KSDK_Bus_Manager* is exposed as a linked PEx component from the *ISF_KSDK_Core* component. See Figure 10. In turn, the Bus Manager creates its own RTOS task. In addition, it links to a PEx *KDSK PIT driver* component and initializes the driver to use the proper PIT timer on the Kinetis microcontroller.

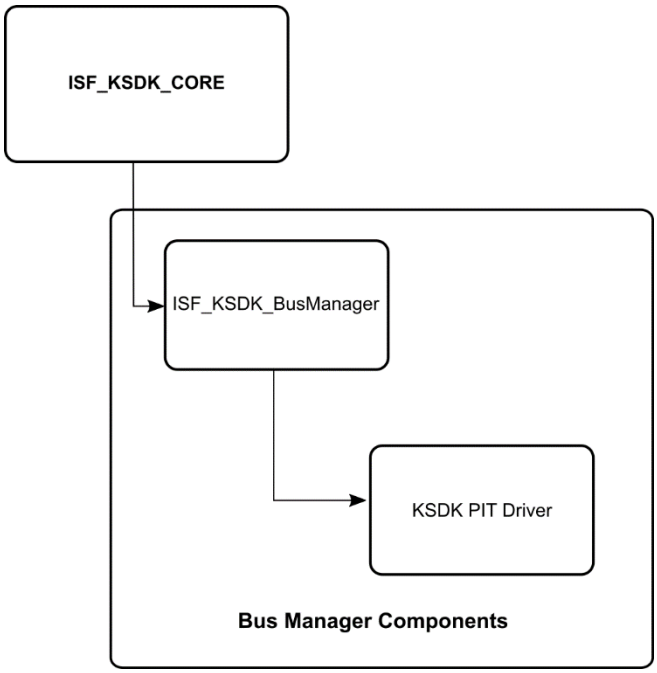


Figure 10. *ISF_KSDK_Core* and Bus Manager components

3.4.7 ISF system configuration

3.4.7.1 Theory of operation

As mentioned previously, the *ISF_KSDK_Core* PEx component uses its System Sensor Configuration properties to generate the global sensor configuration data structures (`gSensorList` and `gSensorHandleList`). The component maintains a list of all the sensors available on the platform along with the data structures necessary to initialize, configure and use the sensors. In turn, these data structures are automatically generated by each of the device-specific Sensor Interface components. The naming convention for the device-specific Sensor Interface components is: *ISF_KSDK_Sensor_<Part Number>_<Sensor Type>*.

The *ISF_KSDK_Core* PEx component also links to an *ISF_KSDK_Protocol_Adapter* component. The *ISF_KSDK_Protocol_Adapter* component contains the list of communication channels to be used in the system and creates the ISF interface files that create the adapters to the KSDK driver components.

3.4.7.2 Sensor Configuration module design

The sensor configuration is captured in the *isf_sensor_configuration.h* and *isf_sensor_configuration.c* files in the generated project. The *isf_sensor_configuration.h* exposes the Sensor Adapter definitions and sensor identifiers for the specified configuration. The *isf_sensor_configuration.c* file creates the `gSensorList`, `gNumSupportedSensors` variable, and the `gSensorHandleList`.

3.4.7.3 Processor Expert component design

The *ISF_KSDK_Core* component uses the **System Sensor Configuration** property to auto generate the *isf_sensor_configuration.h* and *isf_sensor_configuration.c* files. In addition, the Communication Channel list configuration in the *ISF_KSDK_Protocol_Adapter* component is used to generate the global `COMM_CHANNEL_<Channel>` list, the `gSys_ConfiguredChannelList` data structure, as well as the individual channel-specific, global initialization data structures.

3.4.8 Device messaging and protocol adapters

3.4.8.1 Theory of operation

Device Messaging is intimately tied to the individual Protocol Adapters for I²C, SPI, and UART/Serial interfaces. Device Messaging exposes consistent user-level APIs for communicating with external devices. The goal of Device Messaging is to abstract the communications protocol to provide a unified interface for communications, regardless of the underlying transport method used.

Figure 11 depicts the architecture of Device Messaging. Device Messaging depends upon a series of Protocol Adapters. These Protocol Adapters are designed to hide the underlying software driver implementation and manage the multiplexing of those drivers onto specific hardware interfaces. The Protocol Adapters are configured at the system level by their corresponding PEx components. Along with generating the system communication configuration, the components bring in the source code associated with each requested protocol. The KSDK drivers provide for installation of interrupt service routines and tasks required by the protocol.

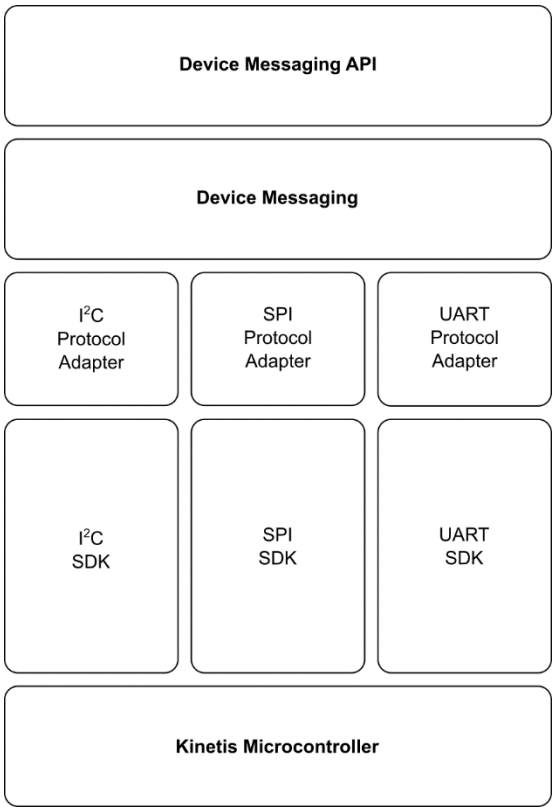


Figure 11. ISF Device Messaging architecture

3.4.8.2 Device Messaging concepts

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

The DM 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 that returns a deviceHandle. The deviceHandle is then passed to the `dm_device_read()` or `dm_device_write()` functions to designate the desired communications endpoint.

The DM component depends on the individual Protocol Adapter implementations to map the DM function calls to a specific function in the KSDK drivers.

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 I^2C transport protocol, a bus object identifies which one of several different I^2C peripherals are used when talking to a particular external I^2C slave.

Using the Device Messaging interfaces, a Device Messaging channel object abstracts the I^2C bus and uses an I^2C Protocol Adapter to communicate with the Device Messaging device endpoints that represent the physical I^2C devices attached to the bus. A global array of the available device messaging channels is generated as part of the ISF system configuration by the *ISF Core PEx* component.

Channel locking

An explicit, channel-locking capability allows extended and 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.

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. A device handle represents a physical device, or communications endpoint. 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 Device Messaging APIs cover channel operations including initialization, locking, reconfiguration, status query and control, as well as device operations including open, close, read and write.

3.4.8.3 Device Messaging module design

The Device Messaging (DM) service provides the `DM API` as a generic interface to any type of underlying protocol. Refer to Figure 12. The `DM API` function calls map directly to a set of function pointers that are autogenerated into the `PROTOCOL` and `gSys_ConfiguredChannelList` data structures. These function pointers are initialized to specific functions inside the corresponding Protocol Adapter for each interface. In addition to the functional interface, the Protocol Adapter creates a Bus Lock for each channel in the system. This Bus Lock is implemented as an OSA Mutex in order to ensure that priority inversion problems can automatically be resolved by the RTOS.

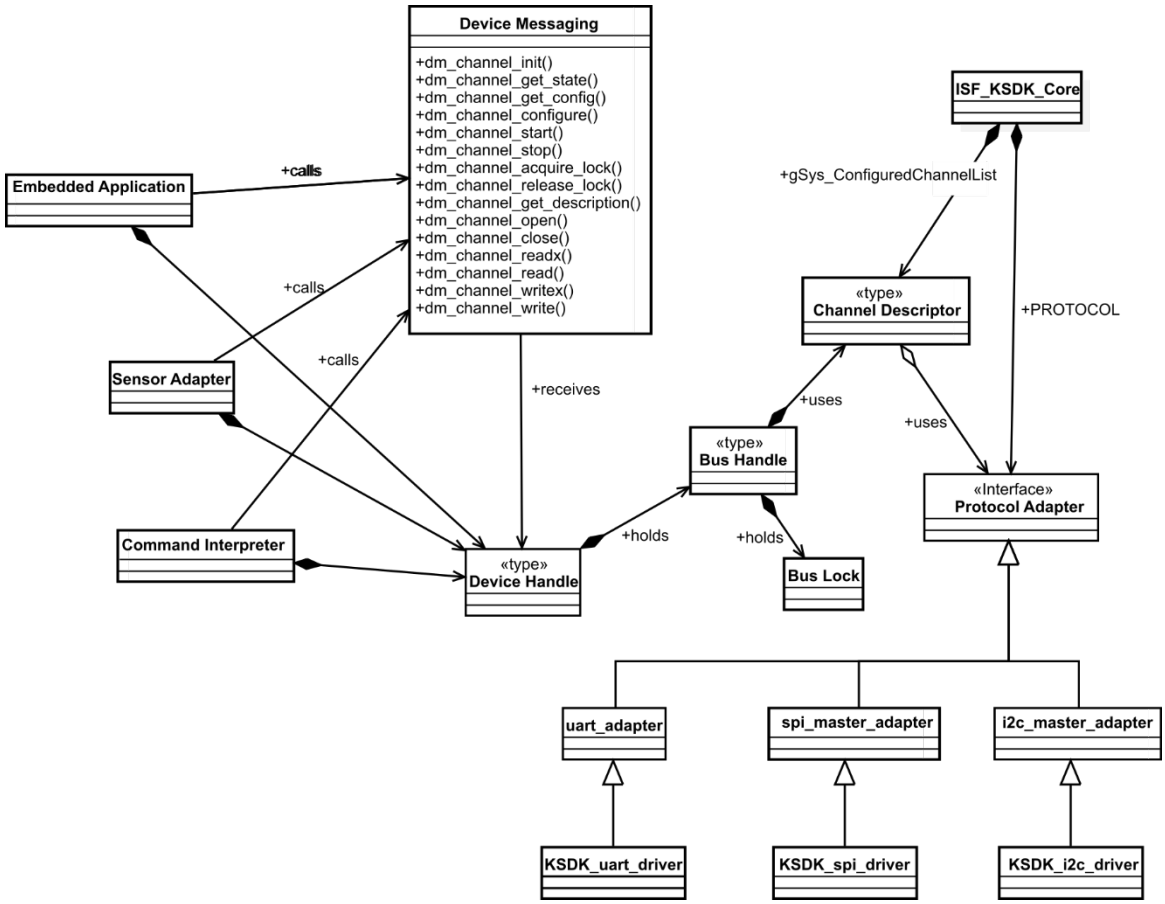


Figure 12. Device Messaging component architecture

3.4.8.4 Device Messaging Processor Expert component design

The Device Messaging services are instantiated in a project by the *ISF_KSDK_Protocol_Adapter* component along with any Protocol Adapter-specific files. Figure 13 illustrates that the *ISF_KSDK_Core* is linked to the *ISF_KSDK_Protocol_Adapter* component. The *ISF_KSDK_Protocol_Adapter* contains the **Comm Channel** property that creates a list of communication channels in the system. Currently, ISF supports I^2C, SPI, and Serial/UART interfaces. Each interface is included via its own unique *ISF_KSDK_CommChannel_*<Interface> component which, in turn, instantiates the underlying PEx KSDK component for that interface. All of the necessary interface files to attach the DM to the specific protocol are also autogenerated within these components.

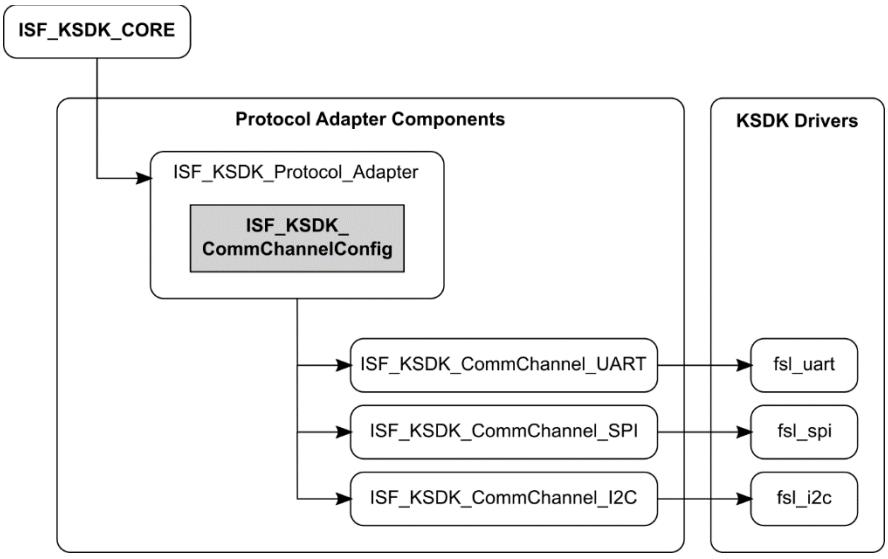


Figure 13. Device Messaging and protocol adapters

3.4.9 Host Interface/Command Interpreter

3.4.9.1 Theory of operation

The Host Interface service is provided by the ISF Command Interpreter (CI), and enables data passing between the host and embedded applications running on the Kinetis microcontroller platform. The interface operates over a UART/serial interface on top of USB or Bluetooth.

The Host Interface currently supports both command/response and streaming protocols over the UART/Serial interface. The Command/Response Protocol is implemented by the Command Interpreter that relies on registered callbacks from the Embedded Application for actually executing commands and producing response data. A working callback is automatically generated by the *Embedded Application* component that implements a set of default commands to allow the host to configure and control sensor subscriptions, read application status, and retrieve raw sensor data. The Streaming Data protocol provides a general purpose mechanism to generate asynchronous data packets to the host whenever a specified set of data elements changes in the Embedded Application.

All of the ISF Host Interface application-level protocols are contained as Data Payloads inside a physical layer, framing format based on the standard High-Level Data Link Control (HDLC) serial packet protocol. This design separates the functionality of the application layer protocols from the packet transport protocol. The protocol uses a Protocol Identifier for routing packets to distinct application-level protocol processing entities. Figure 14 shows the structure of the HDLC protocol. Table 3 provides descriptions of the HDLC protocol packets.

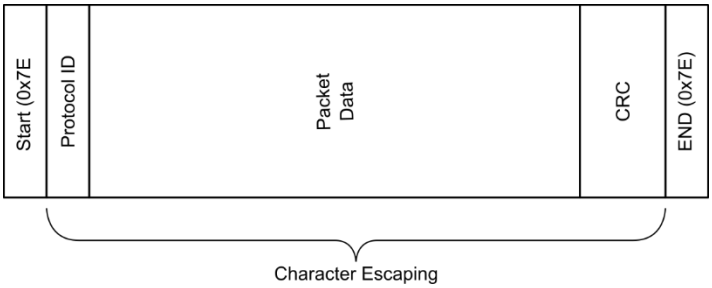


Figure 14. HDLC protocol packet format

The HDLC packet contains the following fields:

Table 3. HDLC data packet descriptions

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x00–Reserved 0x01–CI Protocol also known as Command/Response protocol 0x02–Streaming Protocol 0x03 through 0xFF–Reserved	Protocol identifier, which is the code for predefined protocols; additional protocols can be defined
Packet Data	variable		Packet Data payload
Checksum	2		CCITT-CRC16 of UART Data section (optional)
End Character	1	0x7E	End Marker, delimiter indicating end of packet

The HDLC protocol is applied to packets from the host to ISF and vice versa. Packets from the host to the device or vice versa are marked by a starting and ending byte marker. The byte marker for the start and end of a packet both use the value 0x7E. If the value 0x7E appears as part of the packet data, in between the start and end packet marker, it must be escaped or encoded by the method shown in Table 4.

Table 4. HDLC packet data byte encoding

Data Byte Requiring Encoding	Encode As
0x7D	0x7D followed by 0x5D
0x7E	0x7D followed by 0x5E

When the host or application sends a packet, it is encoded with the escape values that may appear between the start and end packet markers and the checksum and packet markers are added in place. Each side also decodes a received packet and verifies the checksum before interpreting its meaning.

To provide robust communication, a Cyclic Redundancy Check (CRC) feature is optional.

3.4.9.2 Command/Response protocol

The Command Interpreter (CI) provides a general mechanism to accept command packets, and trigger the execution of callback functions registered by the application that handles the 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.

Packets routed between the CI and the host have a specific Packet Data format. This format includes a 4-byte header followed by variable length payload. The Packet Data format is shown in Table 5.

Table 5. CI packet data format

Field Name	Size (bytes)	Value	Protocol layer	Description
Start Character	1	0x7E	HDLC layer	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	ISF layer	CI Protocol also known as Command/Response protocol
AppID	1	variable	CI Protocol	Application Identifier
Command Status	1	Command values: from 0x00 through 0xFF or Response values: from 0x80 through 0xFF		Command or response status
Offset	1 or 2	variable		Offset into the target buffer (depends upon command).
Length	1	variable		Length of the Packet Data payload in bytes
Payload	Length	variable		Packet Data payload
End Character	1	0x7E	HDLC layer	End Marker, delimiter indicating end of packet

The ISF Command Interpreter interface allows an embedded application to interact with the host processor using the Command/Response paradigm—a synchronous interface where the host sends a command packet to the embedded application, which in turn processes the packet and returns a synchronous response to the host.

3.4.9.2.1 Command/Response mode

The CI implements the Command/Response (C/R) mode using a callback design pattern. The C/R protocol typically operates on the serial/UART interface to the host and is assigned protocol ID 0x01. Command/Response packets are handled by the Command/Response handler that is registered by the CI. The C/R protocol handler inspects incoming C/R packets to obtain the AppID field in the command and then passes the whole C/R payload to the corresponding, registered, callback function. Each embedded application typically registers a C/R callback because this is the standard way for an embedded application to receive configuration and control commands from the host. When the callback returns, the C/R protocol handler formats a response packet using data obtained from the callback along with the callback return status, and then sends the response back over the serial/UART interface.

3.4.9.2.2 Command processing

To understand the Command/Response protocol, it is helpful to think of each embedded application running on the device as having two logical buffers, one for input (configuration and control data) and one for the application's output data. Structurally, the buffers can be thought of as having a fixed layout such that a value at a specific location within the buffer always contains the same type of data. In typical usage, the input buffer is often overlaid with a C structure to facilitate use of the input data.

Each application can allocate its own configuration and output buffers. The host can send data to a particular target application by writing data into specific locations within that application's configuration 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. The protocol also supports commands for reading the configuration buffer and the output data buffer.

3.4.9.3 Streaming protocol

An embedded application may also have data sent to the host asynchronously. For example, the application collects data from sensors at a subscribed rate, computes some outputs and sends them to the host. Data from the application's output buffer discussed in the Command/Response protocol, can also be sent to the host asynchronously, whenever it is updated. The Streaming protocol allows the host to specify and subscribe to one or more data elements from an application's output buffer and have those elements sent to the host in an asynchronous message, when it is updated or changed. One usage example for this feature is an embedded application that uses raw sensor data to provide orientation-change information to a host processor. The host sets up three different asynchronous messages for it to receive. The first message, or stream, defines a message containing the raw sensor data that gets sent every time all of the sensors update their data. The second stream is the output of the orientation change algorithm that gets sent to the host only when a change occurs. The third stream is a debug stream that sends data only when an error or other anomalous condition occurs in the application.

In previous releases of ISF, the Command Interpreter implemented a method called Quick-Read (QR) that allowed an embedded application to subscribe to data and have it asynchronously send that data to the host when the data was ready. However, the QR method contained several limitations. The Streaming protocol is designed to overcome these limitations. With the Streaming protocol method, the host can define different sets of data with each set referring to a range of bytes and offsets up to 16 KB in size and it can designate which of these data sets causes data to be sent to the host, when the application updates data.

The Streaming protocol defines a concept called *streams,* which defines a logical data flow of messages, containing a specified set of data that the host can receive in one data packet. Different streams are identified by a unique Stream ID value. A stream is specified using a Stream Configuration object. This object contain two lists:

- Stream Element object list
- Trigger Mask list

Each stream element object describes a slice of an application's output buffer and includes an element ID, the starting offset within the application's output buffer, and the number of bytes to transfer. The Trigger Mask list is a list of bytes that contain information about which stream elements have been updated by the application. This information is used to specify and track when stream data gets sent to the host. The stream data sent to the host is referred to in the following discussion as the *update packet*.

3.4.9.4 Module design

Figure 15 shows the high-level layering and interfaces of the new CI design. The CI uses Device Messaging to interact with the serial/UART interface for character reception and transmission to the host. Use of this abstraction allows the CI to operate over different transports. The CI task state machine receives characters and validates the HDLC framing. Once a complete, valid HDLC frame has been received, it uses the protocol ID in the packet to route the packet to the registered protocol handler. In ISF v2.2 there are two implemented protocols: Command/Response and Streaming. For the Command/Response protocol, the handler interprets the AppID, calls the appropriate application callback function, and returns the formatted response. For the Streaming protocol, the handler interprets the command, formats the response according to the internal protocol state, and returns the response.

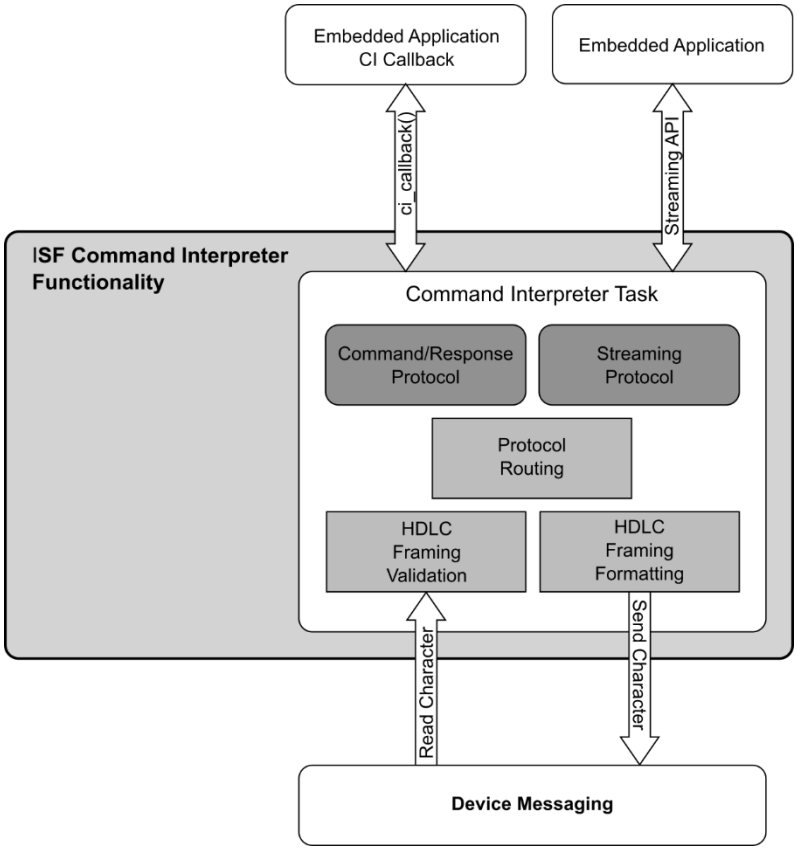


Figure 15. Command Interpreter module design

3.4.9.5 Processor Expert component design

The *Command Interpreter* component is configured via the *ISF_KSDK_Core* component, which contains properties indicating whether the CI service should be instantiated in the project. See Figure 16. If the CI is instantiated, the developer may specify the maximum size of the receive buffer–the default is 34 bytes. The *ISF_KSDK_Core* allows selection of the default CI protocol to run over the serial/UART interface. Additionally, the *ISF_KSDK_Core* component configures the RTOS task for the CI.

Embedded applications can register their Command/Response callbacks via the **CI Callback List** property in the *ISF_KSDK_Core* component.

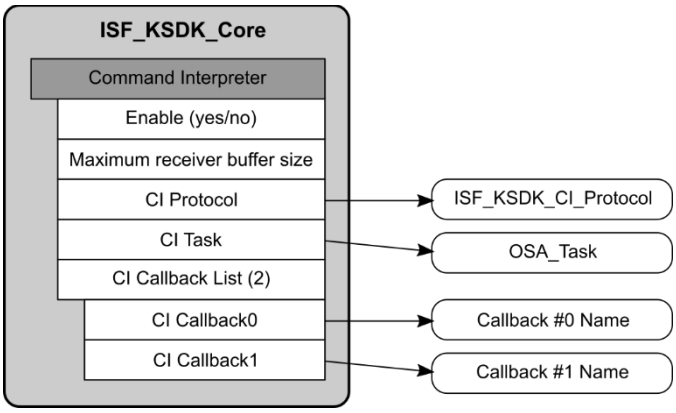


Figure 16. Command Interpreter PEx component hierarchy

3.4.10 Power management

In previous versions of ISF, the Power Manager (PM) provided APIs that allow an embedded application to request changes to the operating power mode of the device. This functionality is now supplied natively by the KSDK *fsl_power_manager* PEx component. The functionality of the ISF Power Manager has been deprecated from ISF v2.2.

3.5 Application support component details

3.5.1 Embedded application component

3.5.1.1 Theory of operation

The *Embedded Application* component allows a developer to configure and generate a complete and working example of an embedded application that subscribes to sensor data and makes it available to the Host Interface without writing a single line of code. Hooks are generated within the code for placement of custom functionality.

A typical workflow for developing a custom embedded application starts with the *Embedded Application* component's default-generated code that can be extended or modified as necessary to achieve the desired custom functionality.

In some cases, the application model constructed in the Embedded Application may be too constraining to be useful in the application developer's design. Developers are not required to use the provided *Embedded Application* component. It is intended to be used as a starting point to understand the available ISF API calls; later it can be replaced with a custom application, linked directly with the ISF embedded middleware.

3.5.1.2 Embedded application module design

The *ISF_KSDK_EmbApp* PEx component accepts a set of configuration properties and generates the software for the Embedded Application task. See Figure 17. The resulting Embedded Application contains three major sections:

- `Host Interface Callback` function
- The Main Application
- An `Application Sensor` State Machine.

Each section is tailored to the developer's application by the properties assigned in the PEx component.

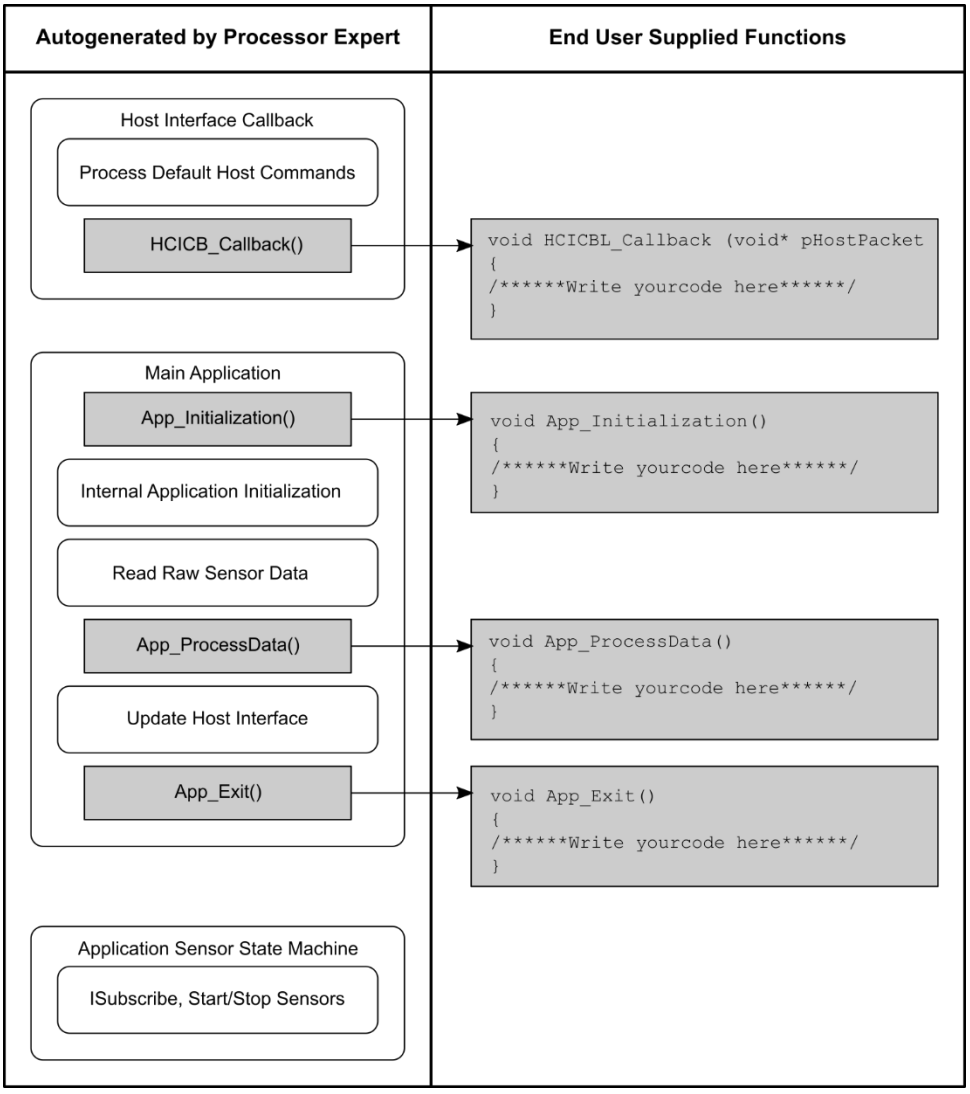


Figure 17. Embedded application module

The Host Interface Callback routine contains the autogenerated default commands. These commands are:

- Read Configuration Data (`CI_CMD_READ_CONFIG`)–Allows the Host to read from the Application Specific configuration data structure.
- Write Configuration Data (`CI_CMD_WRITE_CONFIG`)–Allows the Host to write to the Application Specific configuration data structure.
- Read Application Data (`CI_CMD_READ_APP_DATA`)–Allows the Host to poll the Application Specific Output data structure (including raw sensor samples and processed data).
- Application Reset (`CI_CMD_RESET_APP`)–Causes the Application to return to an initialized state.

In addition to the default commands, each user-defined application is allowed to define its own commands within the valid range specified in the *ISF_KSDK_EmbApp* component. Callback shells are generated in the *Events.c* file.

Within the Main Application, the component provides for initialization of the global data structures and creates a defined set of application resources (events and semaphores used by the task). In addition, it calls a user-definable function that provides for user-defined application-specific data structure declaration and initialization: `App_Initialization()`. The Main Task then falls into a loop that waits on events indicating new raw sensor samples are ready. Samples are in a queue in a FIFO and are ready only when the FIFO is full. Multiple sensor raw data can be read all together or whenever any of the sensors has a new sample. Each time new samples are available, the Main Task calls a user-defined function designed to accept and process samples according the specific application requirements. This function is called `App_ProcessData()` and could implement Motion/Gesture Detection, eCompass, Sensor Function or a wide variety of other features. The Main Task may also be exited for user-definable reasons. In this case, there is an `App_Exit()` function that allows the application to release resources acquired for its own operation.

The Embedded Application contains an autogenerated sensor subscription state machine that accepts a desired operational state and, based on the current Application State, determines and executes a necessary sequence of DSA-Direct calls to bring all of the sensors into that state.

3.5.1.3 Processor Expert component design

The *ISF_KSDK_EmbApp* component contains a set of properties that allows the resulting generated application to be configured and tailored to the embedded application developer's specific needs. See Figure 18. The **Sensor Signaling Method** property allows the developer to select whether to perform sensor data processing every time all of the sensors have completed an update, or when any of the sensors does so. In addition, the Subscription List contains the application level parameters associated with each sensor subscription. These properties determine the sensor type, sensor output format, the specific sensor selected, the desired sampling rate, and the FIFO depth for each subscription. Also, the component allows the developer to define host commands via the User-defined **Host Commands** property. The component links to the underlying *ISF_KSDK_Core* component in order to access information regarding the system sensor configuration. Finally, it creates its own application RTOS Task.

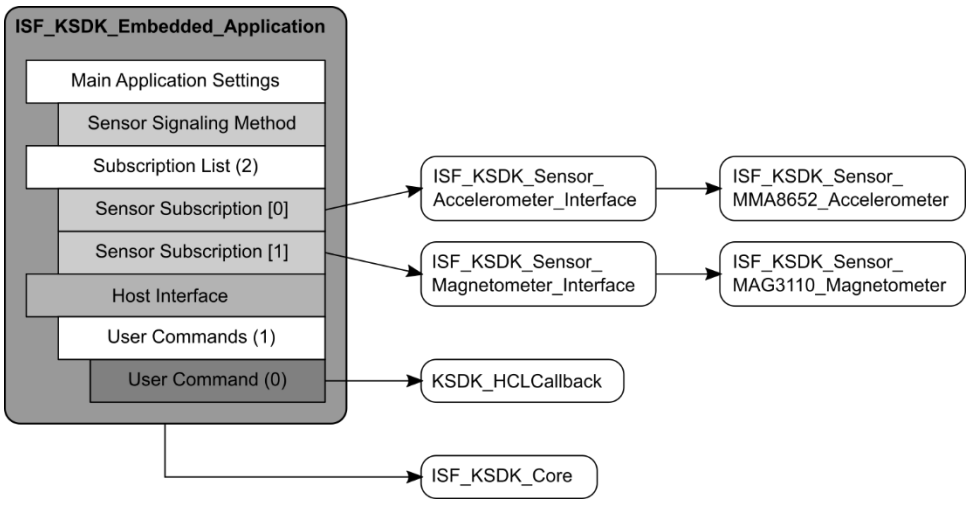


Figure 18. ISF_KSDK_EmbApp hierarchy

3.5.2 Basic Application Component

In addition to the *Embedded Application* component, ISF v2.2 defines the *Basic Application* component which provides a simplified model for embedded application development.

The *ISF_KSDK_BasicApp* PEx component is structured similarly to the *ISF_KSDK_EmbApp* component with a subset of the functionality. *ISF_KSDK_BasicApp* component differs in a few significant ways:

- The Basic Application does not provide for the automatic extension of the CI Command/Response protocol with user-defined commands. Thus, it generates the pre-defined CI commands and then allows the user to edit the CI callback routine directly to generate user-defined CI commands. More information is available in the source file, BasicApp1_Functions.c.
- The Basic Application does not supply or support the autogenerated sensor subscription state machine which is replaced with simplified initialization code.
- The Basic Application provides a user-defined initialization function called BasicApp1_Initialization() and a user-defined sensor data processing function called BasicApp1_OnAnySensor_Data_Ready().

3.5.3 Register Level Interface Application Component

The Register Level Interface (RLI) component (*ISF_KSDK_RLI*) may be optionally included into an ISF v2.2 project. The *ISF_KSDK_RLI* PEx component generates an additional application, which can be addressed separately from an embedded application using its AppID. The RLI application adds the ability to listen for sensor read/write commands from a remote host. Upon receiving a command over the serial interface, the RLI application executes the requested command and returns the result to the remote host. Commands are supported for:

- Selecting the I^2C slave address to use
- Reading one or more bytes from a specified register offset
- Writing one or more bytes to a specified register offset
- Executing the most recent read command periodically at a specified rate and returning the results.

More information regarding the usage of the RLI feature can be found in the ISF v2.2 User's Guide.

3.6 Operating system abstraction

This section provides details of NXP's Operating System Abstraction (OSA) as configured for the ISF implementation in Kinetis by Processor Expert *fsl_os_abstraction* component. The Kinetis SDK supplies the OSA and the KSDK drivers written to the OSA API. ISF v2.2 for Kinetis supports both the MQX RTOS and the open source FreeRTOS at the source code level, as generated by PEx. ISF v2.2 relies on the following services from the OSA:

- Memory Management services
- Task Management services
- Events
- Semaphores
- Mutexes
- Timer services
- Critical Section services

The OSA and the underlying RTOS configuration is managed by Processor Expert components. This generates a configured version of the RTOS for the target system with the processor and board support drivers.

3.6.1 ISF tasks and initialization

ISF has several tasks that must be initialized and running to support its services. These tasks are instantiated by the ISF Processor Expert components automatically through the defined inheritance hierarchy. Processor Expert allows the entry function name, priority, and stack size of the tasks to be modified, but this is not recommended by NXP. The tasks created include, but are not limited to, the Command Interpreter task, the Bus Manager task, and the Initialization task. The Initialization task generates an OSA event called *ISF_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 to be completed prior to execution of their own initializations. The following code has been 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.

4. Protocol definitions

4.1 Command-Response protocol

4.1.1 Built-in commands

ISF provides commands to allow the user to obtain information about the device, the ISF components, the ISF applications, and the user applications. Similarly, other commands allow the user to provide information to the specific applications.

4.1.1.1 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. The `DevInfo` command allows the user to determine information about the target MCU and ISF.

Table 6. DevInfo command packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	0x00	Application Identifier
Command	1	0x00	`CI_CMD_READ_VERSION`
Offset	1	0x00	There is no offset into the data buffer for this command.
Length	1	0x00	The user has no influence on the bytes returned for this command
End Character	1	0x7E	End Marker, delimiter indicating end of packet

DevInfo Command:

The `DevInfo` command is invoked at runtime by sending the following command to the CI,

```
7E 01 00 00 00 00 7E
```

The CI handles the command itself and returns a response packet formatted as shown in Table 7.

Example Response:

```
7E 01 00 80 48 13 00 62 02 02 02 0F 0A 1E 10 28 0A 30 31 32 33 34 35 36 37 00 7E
```

Table 7. DevInfo response packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	0x00	Echoes the Application ID providing the Response
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit, which indicates command completed.
		Bits 6 through 0: variable 0x00 through 0x7F Example Values: 0x80	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS*.
system_device_id	4	variable Example Values: 0x48 0x13 0x00 0x62	32 bit System Device Identification information of the Device (**SIM_SDID** Register) For more details on **SIM_SDID** register, please refer to appropriate Kinetis Sub-Family Reference Manual.
emb_app_present	1	variable from 0x01 to 0xFF Example Value: 0x02	Number of Embedded Applications present in the image Minimum value is 01 for the MBOX App. This example also contains ISFEmbApp
Isf_lib_majorVersion	1	variable Example: 0x02	The ISF major version information
Isf_lib_minorVersion	1	variable Example: 0x02	The ISF minor version information

Protocol definitions

Field Name	Size (bytes)	Value	Description
Isf_lib_buildYear	1	variable where 0x00 corresponds to the year 2000 and 0xFF corresponds to the maximum year of 2255. Example: 0x0F (indicating 2015)	The year of the ISF build
Isf_lib_buildMonth	1	0x00 through 0x0C Example: 0x0A (indicating October)	The month of the ISF build,
Isf_lib_buildDay	1	0x01 through 0x1F Example: 0x1E (indicating 30)	The day of the ISF build,
Isf_lib_buildHours	1	0x00 through 0x17 Example: 0x10 (indicating 16:00)	The time of ISF build in hours,
Isf_lib_buildMinutes	1	0x00 through 0x3B Example: 0x28 (indicating 16:40)	The time of ISF build in minutes,
Isf_lib_buildSeconds	1	0x00 through 0x3B Example: 0x0A (indicating :16:40:10)	The time of ISF build in seconds,
Isf_lib_buildId	8	variable Example: 0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37	A hard-coded sequence used for identifying a valid DevInfo response.
Reserved	0 or 1	If 1 byte, value is 0x00. Example : yes	Byte Alignment Padding. If the value is 00, then it is a padding byte. Any other value is considered to be the next recognized parameter.
End Character	1	0x7E	End Marker, delimiter indicating end of packet

An Embedded Application can also retrieve device information programmatically using the `_fw_device_info_get(device_info_t *info_ptr)` API call. The `_fw_device_info_get()` command fills the memory, at the passed in pointer location, with data according to the following structure:

```
typedef struct {
  uint32          system_device_id;
  uint8           emb_app_present;
  isf_info_t      isf_info;

} device_info_t;
```

The device info metadata includes ISF info as well. The ISF info structure definition is described below:

```
typedef struct
{
  version_info_t  version_info;
  build_info_t    build_info;

} isf_info_t;
```

Where-in, version info structure holds major and minor version information of ISF and build info structure holds ISFbuild year/month/day and build commit_id.

4.1.1.2 ISF Embedded Application Info command

The ISF Application Info (AppInfo) command conforms to the complete Command/Response protocol described previously. Table 8 provides the `AppInfo Command` packet format. The AppInfo command provides the user information about the application associated with AppID value.

Table 8. `AppInfo command` packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	variable	Application Identifier
Command	1	0x00	`CI_CMD_READ_VERSION`
Offset	1	0x00	There is no offset into the data buffer for this command
Length	1	0x00	The user has no influence on the bytes returned for this command
End Character	1	0x7E	End Marker, delimiter indicating end of packet

Protocol definitions

AppInfo Command:

The AppInfo command is invoked at runtime by sending the following command to the CI along with the AppID:

Example AppInfo command for AppID value of 0x01:

```
7E 01 01 00 00 00 7E
```

Example AppInfo command for AppID value of 0x02:

```
7E 01 02 00 00 00 7E
```

The CI handles the command itself and returns a response packet formatted as shown in Table 9.

Example Response:

AppInfo response for AppID value of 01:

```
7E 01 01 80 0E 00 01 00 01 09 4D 42 4F 58 20 41 70 70 00 7E
```

AppInfo response for AppID value of 02:

```
7E 01 02 80 06 00 04 00 01 01 00 7E
```

Table 9. AppInfo response packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	variable Example 1: 0x01 Example 2: 0x02	Echoes the Application ID providing the Response
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit which indicates command completed
		Bits 6 through 0: variable 0x00 through 0x7F	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS*.
Length	1	variable Example 1: 0x0E Example 2: 0x06	Actual number of response payload bytes returned
Length requested	1	0x00	Echoes the input value of bytes to be returned for this command

Field Name	Size (bytes)	Value	Description
appType	1	variable Example 1: 0x01 Example 2: 0x04	8-bit: ISF Application type for example appType is: 01 for MBOX App, 02 for RLI App, 03 for Basic App, 04 for Embedded App.
appMajorVersion	1	variable Example 1 and 2: 0x00	The majorVersion information of the App indicated by the AppID
appMinorVersion	1	variable Example 1 and 2: 0x01	The minorVersion information of the App indicated by the AppID
appNumBytes	1	variable Example 1: 0x09 Example 2: 0x01	Size of the data from the AppID App
appData	appNumBytes	variable Example 1: 4D 42 4F 58 20 41 70 70 00 Example 2: 0x00	Data from the AppID App e.g. interpretation of configuration data. This data can be specified by the user in Processor Expert for the *EmbApp* and *BasicApp* components.
End Character	1	0x7E	End Marker, delimiter indicating end of packet

4.1.1.3 ISF Application Sensor Subscription Info command

The ISF `Application Sensor Subscription Info` command conforms to the complete Command/Response protocol described previously. Table 10 provides ISF Application Sensor Subscription Info command packet format. The Application Sensor Subscription Information provides the user with information about the sensors associated with the AppID.

Table 10. ISF application sensor subscription info command packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	variable	Application Identifier
Command	1	0x09	`CI_CMD_GET_APP_SUBSCRIPTION`
Offset	1	0x00	There is no offset into the data buffer for this command
Length	1	0x00	The user has no influence on the bytes returned for this command
End Character	1	0x7E	End Marker, delimiter indicating end of packet

Example Command:

The Sensor Subscription Info command is invoked at runtime by sending the following command to the CI. The AppID is 02 in the example command shown below:

```
7E 01 02 09 00 00 7E
```

The CI handles the command itself and returns a response packet formatted as shown in Table 11.

First Example Response with a single sensor:

```
7E 01 02 80 0B 01 BC 00 [00 01 66 00 02 BC 08] 00 7E
[] is the single sensor subscription information from the MMA865X accelerometer.
```

Second Example Response with two sensors:

```
7E 01 02 80 11 00 02 30 01 [01 66 00 02 00 08] {02 CA 00 03 CB 14} 00 7E
[] is the first sensor subscription information from the MMA865x accelerometer
and
{} contains the second subscription information for the MAG3110 magnetometer.
```

Table 11. ISF application sensor subscription info response packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	variable	Echoes the Application ID providing the Response
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit which indicates command completed
		Bits 6 through 0: variable 0x00 through 0x7F	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS.*
Length	1	variable Example 1: 0x0B Example 2: 0x11	Actual number of response payload bytes returned
Length requested	1	0x00	Echoes the input value of bytes to be returned for this command
numSensors	1	variable Example 1: 0x01 Example 2: 0x02	Number of Sensors Subscriptions for the App
processedDataBufferOffset	2	variable Example 1: 0xBC00 Example 2: 0x3001	Value that holds offset to the sensor processed data buffer
The following fields apply for each sensor subscribed (numSensors) within the App:			
Reserved	0 or 1	If 1 byte, value is 0x00. Example 1: yes Example 2: no	Byte Alignment Padding. If the value is 00, then it is a padding byte. Any other value is considered to be the next recognized parameter.
sensorId	1	variable Example 1 and 2a: 0x01 Example 2b: 0x02	Sensor Subscription ID for the subscribed sensor within the App

Field Name	Size (bytes)	Value	Description
sensorDataType	2	variable Example1: 0x6600 (3D accel) Example 2a: 0x6600 (3D accel) Example 2b: 0xCA00 (3D mag)	The two bytes form a 16-bit value in little endian format. This value corresponds to the `isf_SensorDataTypes_t` enumeration defined in *isf_sensor_types.h.* Output Data Type Type (Native/Acceleration/Magnetic Field/Temperature/Altitude/Pressure) for the subscribed sensor within the App. e.g. sensorDataTypes: 0x00: Native 0x64 0x00: 1-D Acceleration 0x65 0x00: 2-D Acceleration 0x66 0x00: 3-D Acceleration 0xC8 0x00: 1-D Magnetic Field Strength 0xC9 0x00: 2-D Magnetic Field Strength 0xCA 0x00: 3-D Magnetic Field Strength 0x2C 0x01: 1-D Gyroscope Rotational rate 0x2D 0x01: 2-D Gyroscope Rotational rate 0x2E 0x01: 3-D Gyroscope Rotational rate 0x90 0x01: Quaternion Orientation 0x91 0x01: 1-D Euler Orientation 0x92 0x01: 2-D Euler Orientation 0x93 0x01: 3-D Euler Orientation 0x94 0x01: Direct Cos Matrix Orientation 0xF4 0x01: Temperature 0x58 0x02: Altitude 0x59 0x02: Pressure
sensorResultType	1	0x01 = Raw Counts 0x02 = Fixed Point 0x03 = Floating Point Example 1 and 2a: 0x02 Example 2b: 0x03	Output data format type (output data representation in Raw/Fixed Point/Floating Point) for the subscribed sensor within the App
sampleRateOffset	2	variable	Value that holds offset to the sensor sample rate in µsec for the subscribed sensor within the App
After the final set, there is an end character to mark the end of the response packet			
Reserved	0 or 1	If 1 byte, value is 0x00.	Byte Alignment Padding. If the value is 00, then it is a padding byte. Any other value is considered to be the next recognized parameter.
End Character	1	0x7E	End Marker, delimiter indicating end of packet

4.1.2 Built-in commands for embedded applications

In addition to the built-in mailbox application, the *ISF_KSDK_EmbApp* PEx component defines a default set of commands. This section describes those commands along with usage examples.

Enumerations for these commands are found in the file *isf_ci.h* in the embedded application's *Include* directory.

4.1.2.1 *Read Configuration Data* command (`CI_CMD_READ_CONFIG [0x01]`)

This command returns the desired portion of the application's configuration data buffer, based on the offset and length sent in the command. Table 12 and Table 13 provide Command packet and Response packet Read Configuration data formats, respectively.

Response status:

- `CI_ERROR_NONE` – Success
- `CI_INVALID_COUNT` – Number of bytes requested exceeds size of output data buffer.
- `CI_ERROR_COMMAND` – Offset plus number of bytes is beyond the end of the buffer.

Table 12. *Read Configuration Data*–command packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	variable	Application Identifier
Command	1	0x01 (sets offset size to 1 byte) 0x81 (sets offset size to 2 bytes)	`CI_CMD_READ_CONFIG`
Offset	1 or 2	variable	Offset into the data buffer
Length	1	variable	Number of bytes desired to be returned
End Character	1	0x7E	End Marker, delimiter indicating end of packet

Table 13*. Read Configuration Data*–response packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	variable	Application Identifier

Field Name	Size (bytes)	Value	Description
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit, which indicates command completed
		Bits 6 through 0: variable 0x00 through 0x7F	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS*.
Length requested	1	variable	Number of bytes desired to be returned
Length actual	1	variable	Actual number of bytes returned
Payload	Length actual (plus escape characters when necessary)	variable	Packet Data payload The actual number of bytes of the payload over the wire may be different than the length indicated above because the "Length Actual" is computed against the real payload bytes while the bytes over the wire may include extra escape characters as necessary.
End Character	1	0x7E	End Marker, delimiter indicating end of packet

4.1.2.2 *Write Configuration Data* command `(CI_CMD_WRITE_CONFIG [0x02])`

This command returns the desired portion of the application output data buffer based on an offset and length sent in the command. Table 14 and Table 15 provide Command packet and Response packet Write Configuration data formats, respectively.

Response status:

- `CI_ERROR_NONE` – Success
- `CI_INVALID_COUNT` – Number of bytes requested exceeds size of output data buffer
- `CI_ERROR_COMMAND` – Offset plus number of bytes is beyond the end of the buffer

Table 14. *Write Configuration Data*–command packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	variable	Application Identifier
Command	1	0x02	`CI_CMD_WRITE_CONFIG`
Offset	2	variable	MSB, LSB–Offset into configuration data buffer
Length	1	variable	Number of bytes desired to be written
Data	Length	variable	Data to be written to the configuration buffer
End Character	1	0x7E	End Marker, delimiter indicating end of packet

Table 15. *Write Configuration Data*–response packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	variable	Application Identifier
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit which indicates command completed.
		Bits 6 through 0: variable 0x00 through 0x7F	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS*.
Length requested	1	variable	Number of bytes desired to be written
Length actual	1	variable	Actual number of bytes written
Payload	Length actual	variable	Packet Data payload
End Character	1	0x7E	End Marker, delimiter indicating end of packet

4.1.2.3 *Read Application Data* command (`CI_CMD_READ_APP_DATA [0x03]`)

This command returns the desired portion of the application output data buffer based on an offset and length sent in the command. Table 16 and Table 17 provide Command packet and Response packet Read Application data formats, respectively.

Response status:

- `CI_ERROR_NONE` – Success
- `CI_INVALID_COUNT` – Number of bytes requested exceeds size of output data buffer
- `CI_ERROR_COMMAND` – Offset plus number of bytes is beyond the end of the buffer

Table 16. *Read Application Data*–command packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	variable	Application Identifier
Command	1	0x03 (sets offset size to 1 byte) 0x83 (sets offset size to 2 bytes)	`CI_CMD_READ_APP_DATA`
Offset	1 or 2	variable	Offset into the data buffer
Length	1	variable	Number of bytes desired to be returned
End Character	1	0x7E	End Marker, delimiter indicating end of packet

Table 17. *Read Application Data*–response packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	variable	Application Identifier
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit which indicates command completed.
		Bits 6 through 0: variable 0x00 through 0x7F	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS*.
Length requested	1	variable	Number of bytes desired to be returned
Length actual	1	variable	Actual number of bytes returned
Payload	Length	variable	Packet Data payload
End Character	1	0x7E	End Marker, delimiter indicating end of packet

4.1.2.4 *Read Application Status* command (`CI_CMD_READ_APP_STATUS [0x05]`)

This command returns an application's status information. The command must be explicitly implemented in the embedded application's callback function and the format and contents of the information returned is implementation-specific. Table 18 and Table 19 provide Command packet and Response packet Read Application Status data formats, respectively.

Response status:

- `CI_ERROR_NONE` – Success
- `CI_INVALID_COUNT` – Number of bytes requested exceeds size of output data buffer
- `CI_ERROR_COMMAND` – Offset plus number of bytes is beyond the end of the buffer

Table 18. *Read Application Status*–command packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol
AppID	1	variable	Application Identifier
Command	1	0x05 (sets offset size to 1 byte) 0x85 (sets offset size to 2 bytes)	`CI_CMD_READ_APP_STATUS`
Offset	1 or 2	variable	Offset into the data buffer
Length	1	variable	Number of bytes desired to be returned
End Character	1	0x7E	End Marker, delimiter indicating end of packet

Table 19. *Read Application Status*–response packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	variable	Application Identifier
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit which indicates command completed.
		Bits 6 through 0: variable 0x00 through 0x7F	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS*.
Length requested	1	variable	Number of bytes desired to be returned
Length actual	1	variable	Actual number of bytes returned
Payload	Length	variable	Packet Data payload containing application status data
End Character	1	0x7E	End Marker, delimiter indicating end of packet

4.1.2.5 *Application Reset* command (`CI_CMD_RESET_APP [0x06]`)

This command causes the application to reset its internal state to as close as possible to its initial state out of Power-On Reset. The command returns a confirmation. Table 20 and Table 21 provide Command packet and Response packet Application Reset data formats, respectively.

Response status:

- `CI_ERROR_NONE` – Success

Table 20. *Application Reset*–command packet format

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	variable	Application Identifier
Command	1	0x06 (sets offset size to 1 byte)	`CI_CMD_RESET_APP`
Offset	1	0x00	Offset into the data buffer
Length	1	0x00	Placeholder for compatibility with ColdFire implementations
End Character	1	0x7E	End Marker, delimiter indicating end of packet

Table 21. *Application Reset*–response packet example

Field Name	Size (bytes)	Value	Description
Start Character	1	0x7E	Start Marker, delimiter indicating start of packet
Protocol ID	1	0x01	CI Protocol also known as Command/Response protocol.
AppID	1	variable	Application Identifier
Command Status	1	Bit 7 = 1	This is what is referred to as the COCO bit which indicates command completed.
		Bits 6 through 0: variable 0x00 through 0x7F	These bits contain status of the command. Status = 0b000 0000 indicates successful completion. Any other value indicates an error as described in the ISF API Reference Manual available at *nxp.com/ISF-2.2-KINETIS*.
Length requested	1	0x00	Placeholder for compatibility with ColdFire implementations
Length actual	1	0x00	Actual number of bytes returned
End Character	1	0x7E	End Marker, delimiter indicating end of packet

4.2 Streaming protocol

4.2.1 Introduction

An Embedded Application (EA) may have data to send to the host, asynchronously. For example, the EA collects data from sensors at a subscribed rate. When the sensor produces the data, the EA reads it and sends it to the host. The EA can offer different types of data to the host in a buffer and the host can select which of these data types it needs. A means is needed to allow the host to subscribe to this data and choose which data it wants.

4.2.2 General description

The SP defines a concept called *streams* that encapsulates a set of data that the host can receive in a single data packet. Streams are identified by a unique ID value and contain details of the data pertaining to the stream. A stream is implemented with the Stream Configuration object. This object contain two lists:

- The Stream Element object list
- The Trigger Mask list.

The Stream Element object describes a region of a dataset that includes the Dataset ID, the length, and the offset. The Trigger Mask list is a list of bytes that contains information indicating, which elements have been updated by the EA. This information is used to determine when a stream is sent to the host. The stream data sent to the host is referred to as the *Update packet*.

4.2.2.1 Stream configuration

The Stream Configuration object allows the host to store a Stream Element and Trigger Mask list. The Stream Element list consists of one or more Elements with each Element containing a Dataset ID, length, and offset. The Trigger Mask list is an array of byte(s) with each bit corresponding to a Stream Element. The data structure of a Stream Configuration is implemented as follows:

Field	Size (bytes)	Description
Stream ID	1	ID of this stream
numElements	1	Number of elements in the pElementList
**pTriggerMask*	4	Pointer to an array of trigger byte(s)
**pElementList*	4	Pointer to a list of Stream Element(s)

The stream ID is unique in the system and only one stream can exist with a particular ID value. When an Update packet is sent to the host, the stream ID is included to identify the stream.

The Stream Element list can specify multiple Elements and the same Dataset ID can be specified more than once.

When a stream is created, a set of Trigger Mask byte(s) is required along with Stream Element list information. The number of trigger bytes depends on the number of elements in the stream. Since a byte contains 8 bits, each byte can represent up to 8 elements.

Each bit in the trigger byte corresponds to a stream element. For example, bit 0 of the first byte in the pTriggerMask corresponds to the first Element in the pElementList. In other words, pTriggerMask[0] bit0 corresponds to pElementList[0].

When the EA updates a dataset region which is configured into one or more stream elements, the Trigger Mask bit for those particular elements is cleared in all affected streams. If the Trigger Mask bit is set, it means that the Update packet cannot be sent until the bit is cleared. All bits in the Trigger Mask list must be cleared before the Update Packet is sent to the host.

4.2.2.2 Stream elements

The Stream Element object contains a description of a stream element. The element contains information to describe a set of data that is shown below. The Dataset ID identifies the source element which provides the data.

Field	Size (bytes)	Description
datasetID	1	Identifier for the dataset providing the data
Offset	2	Offset into the dataset's data buffer
Length	2	Number of bytes to take from the dataset

The Dataset ID is defined by the EA. The Dataset specifies a segment of data or all of the data from the data buffer that the EA has to offer to the host.

4.2.2.3 Stream APIs

A summary of the SP APIs are listed here. The detailed description can be found in the *isf_ci_stream.h* file.

4.2.2.3.1 Stream API functions

API Function	Description
`isf_ci_stream_create()`	Create a stream
`isf_ci_stream_update_data()`	Update data of a Dataset
`isf_ci_stream_delete()`	Delete a stream
`isf_ci_stream_reset_trigger()`	Reset the current trigger state to the trigger mask
`isf_ci_stream_get_trigger()`	Get the current trigger state of a given stream
`isf_ci_stream_get_config()`	Get the steam configuration of a stream
`isf_ci_stream_get_num_streams()`	Get the number of streams that currently exists
`isf_ci_stream_get_first()`	Get the first stream in the list of streams
`isf_ci_stream_get_next()`	Get the next stream in the list of streams
`isf_ci_stream_set_CRC()`	Enable or disable CRC code generation and checking

4.2.2.3.2 Stream Protocol APIs

The two functions below are meant to be called by the CI itself. They provide the stream functionality as described in this document. Normally, these functions are specified in the *ISF_KSDK_Core* PEx component and they are placed in the CI protocol list.

API Function	Description
`ci_stream_init()`	Stream initialization that is to be performed before the SP can be used
`ci_protocol_CB_stream()`	Stream Protocol callback function to be registered with the Command Interpreter

4.2.2.3.3 Enable Data Update command

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x01	`CI_CMD_STREAM_ENABLE_DATA_UPDATE` command
3	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1; Status = 0b000 0000 (success)
3	0x01	`CI_CMD_STREAM_ENABLE_DATA_UPDATE` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0x7E	End Marker, delimiter indicating end of packet

4.2.3 Stream host communication

The SP defines three types of packets for communication with the host:

- Command packet
- Response packet
- Update packet

The host sends a command packet and receives a response packet from the SP using the formats defined in Sections 4.2.3.2 and 4.2.3.2.1. For every command packet sent to the SP, the host receives a response packet for that command. The host could also be receiving update packets when the EA has data that is available. A cyclic redundancy feature is available to allow the hosts and EA to verify that a received SP packet is not corrupted.

The packet description refers to a Stream Protocol ID. This ID value is determined by the setup of the EA and depends on how many protocols are registered with the CI and the placement order of the Stream Protocol, relative to other protocols. This value is set at the time the EA is compiled.

4.2.3.1 Cyclic redundancy check (CRC)

To provide robust communication, a cyclic redundancy check (CRC) feature is provided as an option. If the CRC feature is enabled, CRC codes are generated for all packets going to and from the SP. If the host sends a command packet, it is required to generate CRC codes as part of the packet. If the host receives a packet from the SP, the packet contains CRC codes.

The CRC standard used is 16-bit CCITT method. The polynomial used is 0x1021. The two CRC bytes (16-bit) are placed in big endian format at the end of the packet, but before the end marker. The following description of the command and response packets show where the CRC bytes are placed. Note that the CRC code generated does not include the Stream Protocol ID or the start and end marker.

See Section 4.2.7 for the full C code implementation of the CRC standard used by the SP.

4.2.3.2 Host command packet

The host sends a command to the SP in the following format. The start and end marker is always the value 0x7E.

Packet with CRC disabled

Offset	Size (bytes)	Description
0	1	Start Marker, delimiter indicating start of packet (0x7E)
1	1	Stream Protocol ID
2	1	Stream host command
3	X	Data for the command if any is required
3+X	1	End Marker, delimiter indicating end of packet (0x7E)

Packet with CRC enabled

Offset	Size (bytes)	Description
0	1	Start Marker, delimiter indicating start of packet (0x7E)
1	1	Stream Protocol ID
2	1	Stream host command

Offset	Size (bytes)	Description
3	X	Data for the command if any is required
3+X	2	16-bit CRC, [3+X] - msb, [3+X+1] - lsb
5+X	1	End Marker, delimiter indicating end of packet (0x7E)

The Stream Protocol ID value is dependent on where the SP is placed in the CI protocol list in the ISF PEx component. The ID value becomes fixed when the EA is compiled. Refer to Section 4.2.4 for more details on Stream Host commands.

4.2.3.2.1 Command response packet

The SP sends a response packet to the host in the following formats. If the response packet contains data to return to the host, the packet will contain additional information as well as the data itself. The start and end marker is always the value 0x7E. The following response packet is received by the host.

Packet with CRC disabled

Offset	Size (bytes)	Description
0	1	Start Marker, delimiter indicating start of packet (0x7E)
1	1	Stream Protocol ID
2	1	b[7] - Command Complete (COCO), b[6:0] - status
3	1	Stream host command echo
4	2	Length of data in big endian: [4] - msb, [5] - lsb
6	X	Data in response to the command if any
6+X	1	End Marker, delimiter indicating end of packet (0x7E)

Packet with CRC enabled

Offset	Size (bytes)	Description
0	1	Start Marker, delimiter indicating start of packet (0x7E)
1	1	Stream Protocol ID
2	1	b[7] - Command Complete (COCO), b[6:0] - status
3	1	Stream host command echo
4	2	Length of data in big endian: [4] - msb, [5] - lsb
6	X	Data in response to the command if any
6+X	2	16-bit CRC, [6+X] - msb, [6+X+1] - lsb
8+X	1	End Marker, delimiter indicating end of packet (0x7E)

The COCO/status byte at offset 2 provides the host with the status of the command. The COCO bit is set if the command was received and executed. The status portion indicates the result of executing the command. The exact status is dependent on the command and is detailed in Section 4.2.4.

The command echo at offset 3 is a copy of the Stream host command in the command packet. The command echo allows the host to verify, which command this response packet matches.

The length of the data at offset 4/5 is in big-endian format and specifies the number of data bytes following but does NOT include the end marker (0x7E).

4.2.3.3 Update packet

The host can receive update packets asynchronously from the SP. Update packets are data from a stream that the host has requested.

Update packets contain the COCO bit along with the Update packet status at offset 2. The COCO/status byte contains the value 0x82 (COCO = 1 and status = 0b000 0010) to indicate that the packet is an update packet. The host uses this information to distinguish this update packet from a command response packet.

The length of the data at offset 4/5 is in big endian format and specifies the number of bytes following it that includes the Element ID(s) and the data bytes for each Element, but does NOT including the end marker (0x7E).

Packet with CRC disabled

Offset	Size (bytes)	Description
0	1	Start Marker, delimiter indicating start of packet (0x7E)
1	1	Stream Protocol ID
2	1	b[7] - Command Complete (COCO), b[6:0] – status. COCO = 1; Status = 0b000 0010 indicates the packet is an update packet.
3	1	Stream ID of the Update packet
4	2	Length of the following data which includes the IDs and data for all elements of this stream, [4] - msb, [5] - lsb
6	1	Element ID
7	X	Data
7+X	1	Element ID
8+X	Y	Data
Z	1	End Marker, delimiter indicating end of packet (0x7E)

Packet with CRC enabled

Offset	Size (bytes)	Description
0	1	Start Marker, delimiter indicating start of packet (0x7E)
1	1	Stream Protocol ID
2	1	b[7] - Command Complete (COCO), b[6:0] – status. COCO = 1; Status = 0b000 0010 indicates the packet is an update packet.
3	1	Stream ID of the Update packet
4	2	Length of the following data which includes the IDs and data for all elements of this stream, [4] - msb, [5] - lsb
6	1	Element ID
7	X	Data
7+X	1	Element ID
8+X	Y	Data
Z-2	2	16-bit CRC, [Z-2] - msb, [Z-1] - lsb
Z	1	End Marker, delimiter indicating end of packet (0x7E)

4.2.4 Stream host commands

The SP implements a series of host commands that map directly to the stream APIs. The host commands are sent to the EA in the command packet format as defined in Section 4.2.3.2.

4.2.4.1 Command list summary

The available commands are listed here. They can be found in the file *isf_ci_stream.h*

- `CI_CMD_STREAM_RESET`
- `CI_CMD_STREAM_ENABLE_DATA_UPDATE`
- `CI_CMD_STREAM_DISABLE_DATA_UPDATE`
- `CI_CMD_STREAM_CREATE_STREAM`
- `CI_CMD_STREAM_DELETE_STREAM`
- `CI_CMD_STREAM_RESET_TRIGGER`
- `CI_CMD_STREAM_ENABLE_CRC`
- `CI_CMD_STREAM_DISABLE_CRC`
- `CI_CMD_STREAM_GETINFO_NUMBER_STREAMS`
- `CI_CMD_STREAM_GETINFO_TRIGGER_STATE`
- `CI_CMD_STREAM_GETINFO_STREAM_CONFIG`
- `CI_CMD_STREAM_GETINFO_GET_FIRST_STREAMID`
- `CI_CMD_STREAM_GETINFO_GET_NEXT_STREAMID`

4.2.4.2 Command description

This section describes each host command with details.

Note: The description provides example command and response packets for each command and they make the following assumption:

- The Stream Protocol ID is 2. As noted, the actual protocol ID value is dependent on the placement of the protocol in the CI protocol list.
- The Cyclic Redundancy Check (CRC) feature is disabled as the default state, unless noted otherwise in the example.

4.2.4.2.1 Reset Command

Command:	`CI_CMD_STREAM_RESET`
Description:	This command resets the stream protocol. All streams are deleted. The CRC is set to disabled state, update packets are disabled, and internal states are set to default values.
Value:	0x00
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet (if CRC is enabled)

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x00	`CI_CMD_STREAM_RESET` command
3	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x00	`CI_CMD_STREAM_RESET` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.2 Disable Data Update command

Command:	`CI_CMD_STREAM_DISABLE_DATA_UPDATE`
Description:	This command disables the SP from sending update packets to the host.[1]
Value:	0x02
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet (if CRC is enabled)

1. Regardless of whether stream update is enabled or disabled, the applications or tasks running in the EA can always update data in a dataset using the i`sf_ci_stream_update_data()` API. The difference is that if update is disabled and the conditions exists for a stream to send data, the update packet will NOT be sent.

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x02	`CI_CMD_STREAM_DISABLE_DATA_UPDATE` command
3	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x02	`CI_CMD_STREAM_DISABLE_DATA_UPDATE` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.3 Create Stream command

Command:	`CI_CMD_STREAM_CREATE_STREAM`
Description:	This command creates a stream with the given parameters. Memory is allocated from the system to store the stream information including the each element's data.
Value:	0x03
Parameters:	The parameters required to create a stream is the same as for the API function `isf_ci_stream_create()`. The parameters are listed here in the order that they appear in the command packet.
Stream ID	A unique stream ID value
Number of Elements	Number of elements in the element list
Trigger Mask bytes	A list of bytes with each bit in each byte representing one element in the element list. Bit 0 of the first trigger byte corresponds to the first element in the element list. Bit 1 of the first trigger byte corresponds to the second element in the element list, and so on.
Element list	A list of bytes that define one or more datasets in the stream. Each element is defined in a list as follows:

Offset	Size (bytes)	Description
0	1	Dataset ID
1	2	Length, [1] - msb, [2] - lsb
3	2	Offset, [3] - msb, [4] - lsb

Response status:	CI_STATUS_STREAM_SUCCESS – Success
	CI_STATUS_STREAM_ERR_CRC – CRC error in the packet (if CRC is enabled)
	CI_STATUS_STREAM_ERR_STREAMID_EXISTS – The stream ID value is already used by an existing stream.
	CI_STATUS_STREAM_ERR_INVALID_NUM_PARM – The number of parameters provided to create the stream is insufficient. An example is the number of trigger bytes is not sufficient to represent the number of elements in the element list. Another example is the number of bytes in the element list is not sufficient to define all the number of elements specified.
	CI_STATUS_STREAM_ERR_NUMELEMENTS_INVALID – The number of element values is zero. A stream must have at least one element.
	CI_STATUS_STREAM_ERR_OUT_OF_MEMORY – The system is out of memory and the stream cannot be created.
	CI_STATUS_STREAM_ERR_NULL_POINTER – The trigger mask or element list buffer is NULL.

Command Packet example:

Command packet to create a stream:

Stream ID:	0xF0
Number of elements:	2
Trigger mask bytes:	0x03, b[0] – Element 1, b[1] – Element 2:
Element list:	Element 1: / ID / Length / Offset: 0x10, 0x0004, 0x0012
	Element 2: / ID / Length / Offset: 0x11, 0x0345, 0x0513

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x03	CI_CMD_STREAM_CREATE_STREAM command
3	0xF0	Stream ID
4	0x02	Number of elements
5	0x03	Trigger byte (for 2 elements)
6	0x10	Element1 ID
7	0x00	Length MSB
8	0x04	Length LSB
9	0x00	Offset MSB
10	0x12	Offset LSB
11	0x11	Element2 ID
12	0x03	Length MSB
13	0x45	Length LSB
14	0x05	Offset MSB
15	0x13	Offset LSB
16	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x03	`CI_CMD_STREAM_CREATE_STREAM` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.4 Delete Stream command

Command:	`CI_CMD_STREAM_DELETE_STREAM`
Description:	This command deletes a stream with the given stream ID. Memory used to store the stream and its data is released back to the system.
Value:	0x04
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet (if CRC is enabled)
	`CI_STATUS_STREAM_ERR_STREAM_NOEXISTS` – The given stream ID does not exists.

Command Packet example:

Command packet to delete stream ID 0xF0.

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x04	`CI_CMD_STREAM_DELETE_STREAM` command
3	0xF0	Stream ID to delete
4	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x04	`CI_CMD_STREAM_DELETE_STREAM` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.5 Reset Trigger command

Command:	`CI_CMD_STREAM_RESET_TRIGGER`
Description:	This command resets the trigger state of the given stream ID. The current trigger state of the stream is set to the trigger mask.
Value:	0x05
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet (if CRC is enabled)
	`CI_STATUS_STREAM_ERR_STREAM_NOEXISTS` – The given stream ID does not exist.

Command Packet example:

Command packet to reset stream ID 0xF0.

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x05	`CI_CMD_STREAM_RESET_TRIGGER` command
3	0xF0	Stream ID to reset trigger
4	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x05	`CI_CMD_STREAM_RESET_TRIGGER` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.6 Enable CRC command

Command:	`CI_CMD_STREAM_ENABLE_CRC`
Description:	This command enables CRC code generation and checking. Note that the response packet for this command contains two additional bytes for the CRC code. Any packets that the host sends to the SP after this command is required to have the CRC codes. The SP uses the CRC to check for data corruption. If corruption occurs, the response packet contains the status error `CI_STATUS_STREAM_ERR_CRC`.
Value:	0x06
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet if CRC is enabled.

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet r
1	0x02	Stream protocol ID
2	0x06	`CI_CMD_STREAM_ENABLE_CRC` command
3	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Note: the response packet contains two bytes for CRC codes.

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x06	`CI_CMD_STREAM_ENABLE_CRC` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0xDA	CRC MSB (SP generated)
7	0xD5	CRC LSB (SP generated)
8	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.7 Disable CRC command

Command:	`CI_CMD_STREAM_DISABLE_CRC`
Description:	This command disables CRC code generation and checking. The response packet for this command does NOT contain CRC codes as it is disabled.
Value:	0x07
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet (if CRC is enabled)

Command Packet example:

Assume that CRC is currently enabled. The host generates CRC codes as part of the packet.

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x07	`CI_CMD_STREAM_DISABLE_CRC` command
3	0xBC	CRC MSB (host generated)
4	0x7B	CRC LSB (host generated)
5	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x07	`CI_CMD_STREAM_DISABLE_CRC` command echo
4	0x00	Length MSB
5	0x00	Length LSB
6	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.8 Get Number of Streams command

Command:	`CI_CMD_ STREAM_GETINFO_NUMBER_STREAMS`
Description:	This command returns the number of streams that currently exist.
Value:	0x08
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet if CRC is enabled

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x08	`CI_CMD_STREAM_GETINFO_NUMBER_STREAMS` command
3	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

The example assumes that five streams exist in the system.

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x08	`CI_CMD_STREAM_GETINFO_NUMBER_STREAMS` command echo
4	0x00	Length MSB
5	0x01	Length LSB
6	0x05	Number of streams that currently exists
7	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.9 Get Trigger State command

Command:	CI_CMD_ STREAM_GETINFO_TRIGGER_STATE
Description:	This command returns the current trigger state of a given stream.
Value:	0x09
Parameters:	None
Response status:	CI_STATUS_STREAM_SUCCESS – Success
	CI_STATUS_STREAM_ERR_CRC – CRC error in the packet if CRC is enabled.
	CI_STATUS_STREAM_ERR_STREAM_NOEXISTS – The given stream ID does not exist.

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x09	CI_CMD_STREAM_GETINFO_TRIGGER_STATE command
3	0xF0	Stream ID to get trigger state
4	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

This example assumes that stream 0xF0 has 10 elements (10-bits, two trigger bytes) with current state of:

- 0xF9 – Elements 0 to 7, b[7:0]
- 0x02 – Elements 8 to 9, b[1:0]

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x09	CI_CMD_STREAM_GETINFO_TRIGGER_STATE command echo
4	0x00	Length MSB
5	0x02	Length LSB
6	0xF9	Trigger state of elements 0-7
7	0x02	Trigger state of elements 8-9
8	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.10 Get Stream Configuration command

Command:	`CI_CMD_ STREAM_GETINFO_STREAM_CONFIG`
Description:	This command returns the configuration of a given stream.
Value:	0x0A
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet if CRC is enabled
	`CI_STATUS_STREAM_ERR_STREAM_NOEXISTS` – The given stream ID does not exist.

Command Packet example:

Assume that a stream exists with the following attributes:

- Stream ID: 0xF0
- Number of elements: 2
- Trigger mask bytes: 0x03, b[0] – Element 1, b[1] – Element 2
- Element list:
  — Element 1 / DatasetID / Length / Offset: 0x10, 0x0004, 0x0012
  — Element 2 / DatasetID / Length / Offset: 0x11, 0x2345, 0x9513

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x0A	`CI_CMD_STREAM_GETINFO_STREAM_CONFIG` command
3	0xF0	Stream ID to get trigger state
4	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x0A	`CI_CMD_STREAM_GETINFO_STREAM_CONFIG` command echo
4	0x00	Length MSB
5	0x0D	Length LSB (13 bytes)
6	0xF0	Stream ID
7	0x02	Number of elements
8	0x03	Trigger byte (for 2 elements)
9	0x10	Element 1 Dataset ID
10	0x00	Length MSB
11	0x04	Length LSB

Byte	Value	Description
12	0x00	Offset MSB
13	0x12	Offset LSB
14	0x11	Element 2 Dataset ID
15	0x23	Length MSB
16	0x45	Length LSB
17	0x95	Offset MSB
18	0x13	Offset LSB
19	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.11 Get First Stream ID command

Command:	`CI_CMD_ STREAM_GETINFO_GET_FIRST_STREAMID`
Description:	Streams are stored in a linked list and this command returns the ID of the first stream in the list.
Value:	0x0B
Parameters:	None
Response status:	`CI_STATUS_STREAM_SUCCESS` – Success
	`CI_STATUS_STREAM_ERR_CRC` – CRC error in the packet if CRC is enabled
	`CI_STATUS_STREAM_ERR_STREAM_NOEXISTS` – No streams exist

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x0B	`CI_CMD_STREAM_GETINFO_GET_FIRST_STREAMID` command
3	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Assume that the ID of the first stream is 0xF0.

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x0B	`CI_CMD_STREAM_GETINFO_GET_FIRST_STREAMID` command echo
4	0x00	Length MSB
5	0x01	Length LSB
6	0xF0	Stream ID
7	0x7E	End Marker, delimiter indicating end of packet

4.2.4.2.12 Get Next Stream ID command

Command:	CI_CMD_ STREAM_GETINFO_GET_NEXT_STREAMID
Description:	Streams are stored in a linked list and this command returns the ID of the stream that is next in the list from the previous get first or get next stream ID. If this command is issued without calling the CI_CMD_STREAM_GETINFO_GET_FIRST_STREAMID, the SP returns the status CI_STATUS_STREAM_STREAM_END_OF_LIST.
Value:	0x0C
Parameters:	None
Response status:	CI_STATUS_STREAM_SUCCESS – Success
	CI_STATUS_STREAM_ERR_CRC – CRC error in the packet if CRC is enabled
	CI_STATUS_STREAM_ERR_STREAM_NOEXISTS – No streams exist
	CI_STATUS_STREAM_STREAM_END_OF_LIST – End of stream list

Command Packet example:

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x0C	CI_CMD_STREAM_GETINFO_GET_NEXT_STREAMID command
3	0x7E	End Marker, delimiter indicating end of packet

Response Packet example:

Assume that the ID of the first stream is 0xF1.

Byte	Value	Description
0	0x7E	Start Marker, delimiter indicating start of packet
1	0x02	Stream protocol ID
2	0x80	COCO = 1, status = 0b000 0000 (success)
3	0x0C	CI_CMD_STREAM_GETINFO_GET_NEXT_STREAMID command echo
4	0x00	Length MSB
5	0x01	Length LSB
6	0xF1	Stream ID
7	0x7E	End Marker, delimiter indicating end of packet

4.2.5 Triggers, Elements, and updates

The SP sends stream data to the host with an update packet, when the data in the stream is ready. The condition under which the stream data is ready depends on two factors:

- The trigger state
- The stream-enable state.

This section discusses the factors that affect the trigger state.

4.2.5.1 Elements

Streams are created with an element list consisting of one or more elements. Element information contains a Dataset ID, length, and offset. The length and offset specify the region of interest to the host within the specified dataset. Memory is allocated as part of the stream to store the length of the data. For example, if the dataset has a length of 100 bytes and an offset of 500, 100 bytes is allocated to store the data.

When an EA updates data using the `isf_ci_stream_update_data()` API, it specifies the dataset ID, length, offset, and a pointer to the source data. The source region is compared to the elements in all currently configured streams. If a stream element requires data from the updated dataset, only that portion of the dataset required to satisfy the configured stream elements is copied. If at least one byte overlaps, the data is copied from the source to the element's data area in the stream. When this happens, the element is considered to be updated. This event causes the trigger state to change and is discussed in Section 4.2.5.2.

4.2.5.2 Trigger states

Streams are created with a set of trigger masks. Each element in the element list has a corresponding bit in the trigger mask. If the mask bit for the element is set to one, then the data for that element must be updated by the EA, before the update packet belonging to the stream can be sent. If the mask bit is set to zero, then the element is not required to be updated before the update packet is sent. In addition, all trigger bits of the stream must be set to zero before the update packet is sent.

Each stream keeps the current state of the trigger bits with a trigger state byte. When the stream is created, the stream initializes the trigger state to be the same as the trigger mask. If an element is updated, the corresponding bit in the trigger state is set to zero. Refer to Section 4.2.5.2.

The stream trigger state is initialized to the trigger mask when the stream is created, and it can also be initialized when the EA or host resets the trigger of the stream.

In the case where the trigger mask byte(s) provided are all zeros, any update to any elements in the stream cause an update packet to be sent to the host, provided that the source dataset's region overlaps with the element's region.

Note: There may be unused bits in the trigger masks. For example, if there are five elements in the element list, only bits b[4:0] of the trigger byte are needed. The SP disregards the values in the unused trigger bits, b[7:5] in its processing. This means that unused, trigger mask, bits provided to the SP during stream creation, have no effect. Only the used trigger bits are processed by the SP.

4.2.5.3 Update conditions

The SP sends an update packet of a stream when these conditions are met:

- All the trigger state byte(s) of the stream are zeros.
- The stream update is enabled. The host enables stream update by using the host command `CI_CMD_STREAM_ENABLE_DATA_UPDATE`.

Note: Regardless of whether stream update is enabled or disabled, the EA can always call the `isf_ci_stream_update_data()` API to update data. If the regions of data overlap, then the data is updated. Whether the update packet is sent or not depends on the trigger state and the stream update enable state.

4.2.5.4 Update example

The EA or host creates a stream with the following elements.

<table>
<tr><th>Stream Element Description</th><th>Value</th></tr>
<tr><td>Number of elements:</td><td>3</td></tr>
<tr><td>Trigger mask byte value:</td><td>0x05</td></tr>
<tr><td>b[0] =</td><td>1 for Element 1</td></tr>
<tr><td>b[1] =</td><td>0 for Element 2</td></tr>
<tr><td>b[2] =</td><td>1 for Element 3</td></tr>
<tr><td>b[7:3]</td><td>are unused (values are disregarded)</td></tr>
<tr><td>Element list:</td><td>
<table>
<tr><th>–</th><th>Dataset ID</th><th>Length</th><th>Offset</th></tr>
<tr><td>Element 1</td><td>0x10</td><td>0x0008</td><td>0x0010</td></tr>
<tr><td>Element 2</td><td>0x11</td><td>0x0200</td><td>0x0500</td></tr>
<tr><td>Element 3</td><td>0x12</td><td>0x0100</td><td>0x0000</td></tr>
</table>
</td></tr>
</table>

The internal trigger state is initialized to the value 0x05, same as the trigger mask value. Consider the following cases of events and the outcomes.

Event 1: EA calls `isf_ci_stream_update_data()` API to update Dataset 0x12.

Field	Value
Dataset ID:	0x12
Length:	0x0008
Offset:	0x0000

The EA source data region being updated overlaps with Element 3's data region. The portion of data that overlaps is copied over and the trigger state bit for Element 3 is set to zero.

Current trigger state value: 0x01

1. Overlapped source region copied to stream element buffer
2. b[2] is set to zero

New trigger state value: 0x01

No update packet is sent because the trigger state byte is not zero.

Field	Value
Current trigger state value :	0x01 Overlapped source region copied to stream element buffer
b[2]	is set to zero
New trigger state value:	0x01 No update packet is sent because the trigger state byte is not zero.

No update packet is sent because the trigger state byte is not zero.

Event 2: EA calls `isf_ci_stream_update_data()` API to update Dataset 0x11.

Field	Value
Dataset ID:	0x11
Length:	0x0200
Offset:	0x0400

The region being updated overlaps with Element 2 region. Note that a portion of the EA source region being updated is outside of Element 2's region. In this case, only the source region that overlaps with Element 2's region is updated. The overlapped region starts at offset 0x500 with a length of 0x0100 bytes and only this region is copied to the dataset.

Because the trigger state bit for Element 2 is already cleared, there is no change in the trigger state.

Current trigger state value: 0x01

1. Overlapped source region copied to the stream element buffer.
2. b[1] set to 0 (already 0 so no change)

New trigger state value: 0x01

No update packet is sent because the trigger state byte is not 0.

Event 3: EA calls `isf_ci_stream_update_data()` API to update Dataset 0x10.

Field	Value
Dataset ID:	0x10
Length:	0x0020
Offset:	0x0030

The EA source data region being updated does NOT overlap with Element 1's region. No data is copied. The trigger bit for Element 1 remains a value of one.

Current trigger state value: 0x01

New trigger state value: 0x01

No update packet is sent because the trigger state byte is not 0.

Event 4: EA calls `isf_ci_stream_update_data()` API to update Element 1.

Field	Value
Dataset ID:	0x10
Length:	0x004
Offset:	0x0010

The EA source data region being updated overlaps with Element 1's region. The source data is copied to the dataset. The trigger state bit for Element 1 is set to zero.

Current trigger state value: 0x01

1. b[0] is set to 0
2. Trigger state value: 0x00
3. Update packet for the stream is sent to the host if stream update is enabled.
4. Trigger state is reset to the trigger mask value.
5. New trigger state value: 0x05

In this situation, the trigger state becomes zero after the element data is updated that causes an update packet to be sent if stream update is enabled. The trigger state is then reset to the trigger mask value.

4.2.6 Internal design

This section discusses the internal design of the SP. The stream instance and configuration are described along with the handling of the instance linked list.

4.2.6.1 Stream instance

Stream information is stored in a singly-linked list of stream instances. The placement of the stream in the linked list is in the order in which they are created by the EA or host. The last stream in the list points to a NULL stream. Each stream instance encapsulates the following information about the stream:

- Stream configuration that includes stream ID, trigger mask, and element(s)
- Current trigger states
- Stream buffer; for more details see Section 4.2.6.2.
- Pointer to the next stream in the linked list

The C code definition of the stream instances is as follows:

```
typedef struct  __attribute__ ((__packed__)) _ci_stream_instance
  {
    ci_stream_config_t *pStreamConfig;
    uint8     *pStreamBuffer;
    uint8     *pTriggerState;
    struct    _ci_stream_instance *pNextInstance;
  } ci_stream_instance_t;
```

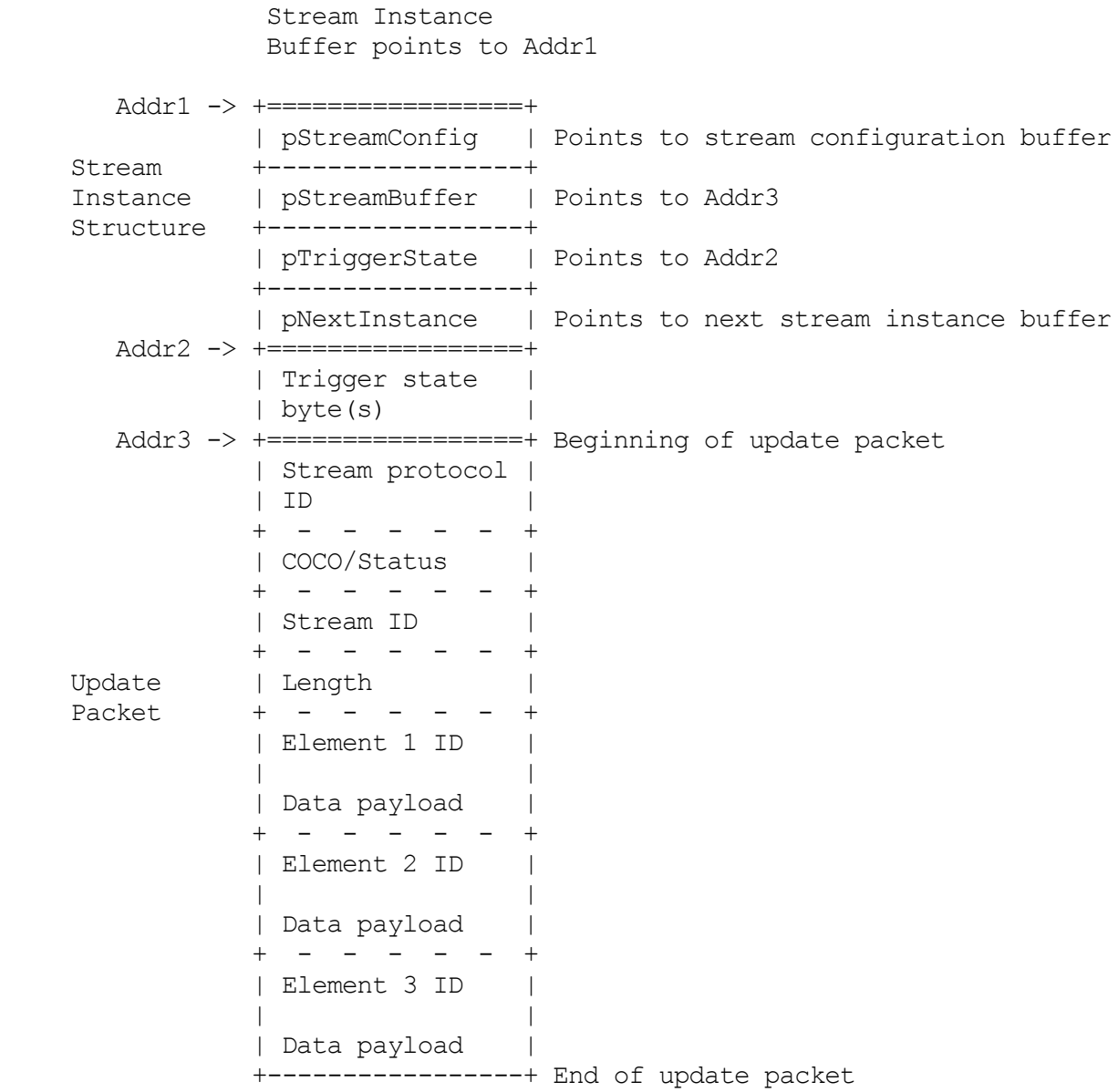
4.2.6.2 Stream instance buffer

Streams contain stream information and the data for each element in the stream. Stream information consists mainly of the stream instance structure plus the trigger states. When an update packet is sent to the host, the typical method is to copy the element information from the stream to an update packet and send it. However, the stream may contain large amounts of data and the system may not have sufficient memory for the update packet, even only temporarily. In addition, copying the data adds to the latency of processing an update packet.

In order to use memory efficiently and reduce latency, a single, dynamically-allocated stream instance buffer is created for each stream to hold the stream information and the update packet. The update packet portion of the buffer contains all the information needed, including the data for each element. With this method, only one memory buffer is required to store data for the element and the SP can send the update packet with one contiguous buffer, saving memory and time.

Protocol definitions

The stream instance buffer is structured as follows:

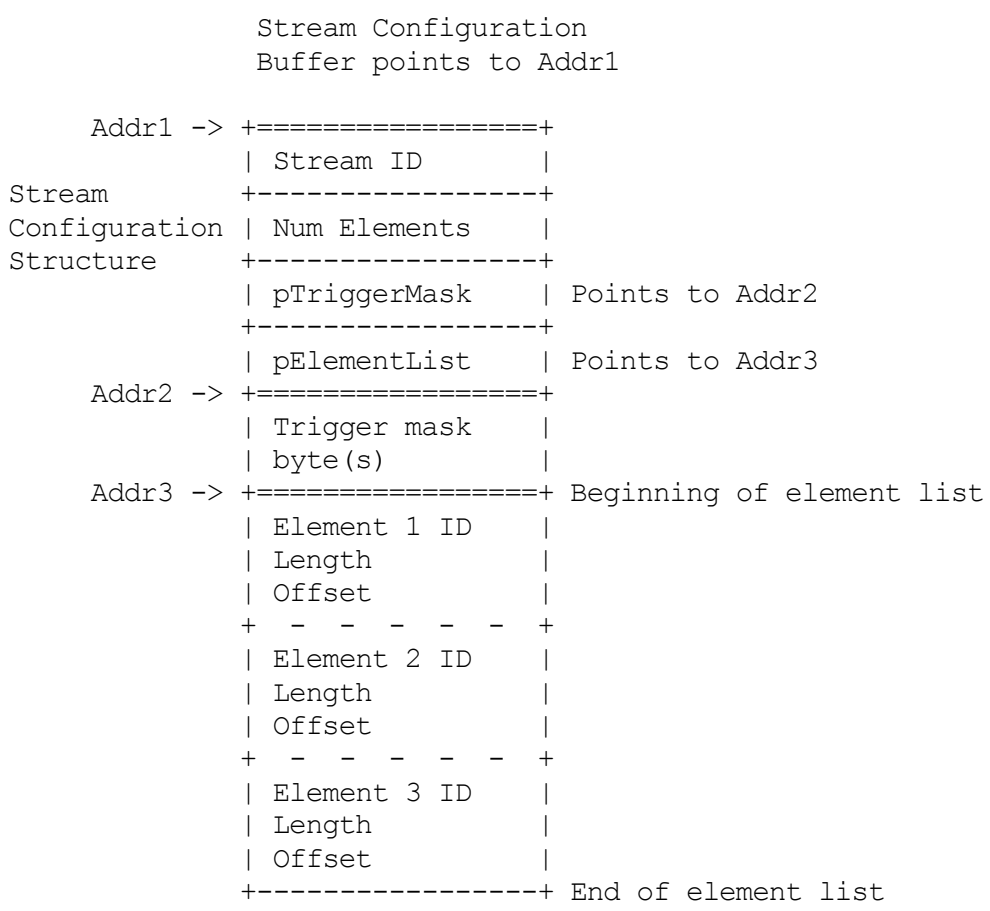


During the stream creation process, the static information of the update packet is initialized, including the protocol ID, COCO/status (0x82 value), stream ID, length, and element ID(s). Each element's data is initialized to zero. Populating these fields is done at initialization to ensure the update packet is ready to be sent to the host, when trigger conditions allow.

4.2.6.3 Stream configuration buffer

A stream instance holds information about the stream in a stream configuration object. The Stream configuration contains information such as stream ID, trigger mask byte(s), and the element list consisting of element(s) information. Each element has a Dataset ID, length, and offset that are stored in the element list.

Similar to the stream instance buffer, the stream configuration is stored in a stream configuration buffer that is designed to minimize memory usage. The stream configuration buffer layout is as follows:



4.2.6.4 Stream instance linked list modification

Modification of the linked list is handled in a typical fashion as described in the following cases:

Case 1: Adding Stream3 to the list.

Before:

Stream1 -> Stream2 - >NULL

After:

Stream1 -> Stream2 -> Stream3 -> NULL

Case 2: Deleting the first stream, Stream1, from the list.

Before:

Stream1 -> Stream2 -> Stream3 -> NULL

After:

Stream2 -> Stream3 -> NULL

Case 3: Deleting the middle stream, Stream2, from the list.

Before:

Stream1 -> Stream2 -> Stream3 -> NULL

After:

Stream1 -> Stream3 -> NULL

4.2.7 CRC implementation

The 16-bit, CCITT CRC standard is implemented in ISF v2.2 using the following code:

```
#define POLY_CRC16_GENERATOR      0x1021


uint16 ccitt_crc16_cal(uint32 anumBytes, uint8 *apBuf)
{

  uint16 crc16 = 0xffff;
  uint8 *p8 = (uint8*)apBuf;
  uint8 bit;
  uint16 xor_flag;

  while(anumBytes--)
  {

    uint8 v;

    //  Align test bit with leftmost bit of the message byte.
    v = 0x80;

    bit = 0;
    do
    {
        if (crc16 & 0x8000)
        {
            xor_flag= 1;
        }
        else
        {
            xor_flag= 0;
        }
        crc16 = crc16 << 1;


        if (*p8 & v)
        {
```

```
            // If not zero, then append the next bit of the message
            // to the end of the CRC. The zero bit placed there by
            // the shift above need not be changed if the next bit of
            // the message is zero.
            crc16 = crc16 + 1;
        }

        if (xor_flag)
        {
            crc16 = crc16 ^ POLY_CRC16_GENERATOR;
        }

        // Align test bit with next bit of the message byte.
        v = v >> 1;

    } while(++bit < 8);

    p8++;
  }


  bit = 0;
  do
  {
    if (crc16 & 0x8000)
    {
      xor_flag= 1;
    }
    else
    {
      xor_flag= 0;
    }
    crc16 = crc16 << 1;

    if (xor_flag)
    {
      crc16 = crc16 ^ POLY_CRC16_GENERATOR;
    }
  } while(++bit < 16);

  return crc16;
}
```

5. References

Resource	Description	Link
MQX RTOS Reference Manual	Documentation	nxp.com/files/32bit/doc/ref_manual/MQXRM.pdf
MQX RTOS User Guide	Documentation	nxp.com/files/32bit/doc/user_guide/MQX_USER_GUIDE.pdf
FreeRTOS	Web page	nxp.com/FREERTOS
Kinetis Design Studio Downloads	Tool Summary page	nxp.com/KINETIS-IDE/DOWNLOADS
Kinetis Software Development Kit	Documentation	nxp.com/KINETIS-SDK/DOCS
Freedom Development Platform	Tool Summary page	nxp.com/FREEDOM
Processor Expert Processor Expert Drivers	Documentation	nxp.com/PROCESSOREXPERT/DOCS nxp.com/PROCESSOREXPERT-MICRODRIVER/DOCS
FRDM-KL25Z	Documentation	nxp.com/FRDM-KL25Z/DOCS
FRDM-KL26Z	Documentation	nxp.com/FRDM-KL26Z/DOCS
FRDM-K22F	Documentation	nxp.com/FRDM-K22F/DOCS
FRDM-K64F	Documentation	nxp.com/FRDM-K264F/DOCS
Sensor Fusion	Tool Summary Page	nxp.com/SENSORFUSION
ISF v2.2 Kinetis User Guide	Documentation	nxp.com/isf-2.2-KINETIS
ISF v2.2 API Reference Manual	Documentation	nxp.com/isf-2.2-KINETIS
ISF v2.2 Release Notes	Documentation	nxp.com/isf-2.2-KINETIS

6. Revision history

Rev. No.	Date	Description
1.0	2/2016	Initial public release

How to Reach Us:

Home Page:
NXP.com

Web Support:
NXP.com/support

Document Number: ISF2P2_KINETIS_SWRM
Revision 1.0, 2/2016