Document Number: ISF2P2_UG
Rev. 1.0, 2/2016

Intelligent Sensing Framework v2.2 User Guide

For the Kinetis Family of Microcontrollers

Contents

1 Introduction

This User Guide is intended to provide answers to practical questions about the use of the Intelligent Sensing Framework (ISF) along with example code snippets that demonstrate the functionality discussed. It is not designed to be read cover-to-cover, but is intended to be a browsable reference for developers.

Section 2 discusses the overall hardware environment and the sensors operating with ISF.

Section 3 discusses the direct use of some ISF Components such as the Bus Manager and Event Handler.

Section 4 discusses communications with a PC using the ISF Command Interpreter.

Section 5 discusses the main application flow and some alternate ways to code a main sensor loop including using either of the provided application components or integrating ISF into a user's existing MQX™ or FreeRTOS-based application.

Section 6 discusses how to use Processor Expert in conjunction with ISF and covers concepts such as:

- Generating an initial code base and then modifying it manually
- Logical device drivers versus the Kinetis Software Development Kit
- Configuring and using FreeRTOS
- Differences between MQX and FreeRTOS

Note: The code examples in this document use both Operation System Abstraction (OSA) and MQX function calls interchangeably. It is left to the reader to adopt their preferred method. ISF v2.2 uses the OSA layer exclusively.

2 Hardware and sensors

2.1 Interacting with sensors

There are four ways to interact with sensors using ISF. One way is from a remote host and three ways are with an embedded app.

- The Register Level Interface (RLI) component can be used to send read/write sensor register commands over the MCU's serial interface.

  The RLI interface component adds the ability to listen for sensor read/write register commands from a remote host. Upon receiving a command over the serial interface, the RLI component 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.

Kinetis Interface Tool (KIT) PC GUI provides a tab that implements the remote host side of the RLI protocol. This allows simple register level access to any connected I^2C sensor without any programming required. One advantage of the RLI interface is that it can be used to interact with sensors that are otherwise not supported by ISF.

- An embedded application can subscribe to sensors using the interfaces declared in *isf_dsa_direct.h*:

```
int32 init_sensor();
int32 configure_sensor();
int32 start_sensor();
int32 stop_sensor();
int32 shutdown_sensor();
int32 convert_sensor_data();
```

  The Digital Sensor Abstraction (DSA) direct APIs provide a thin, but convenient wrapper around the native Sensor Adapter DSA compliant interface. The wrapper handles proper initialization of the `isf_SensorHandle_t` data structure and simplifies access to the underlying Sensor Adapter interface functions.
- An embedded application can use a sensor adapter directly using the interfaces declared in *isf_dsa_adapter.h*.

  For an example of how to use these interfaces, look at the *isf_dsa_direct.c* code in the *Generated_Code/ISF/Core/Source* directory of an ISF project.
- An embedded application can use the Device Messaging APIs to directly read or write to a sensor.

  The Device Messaging (DM) APIs provide for talking to sensor devices over I^2C or SPI. Succinctly, the DM provides a common interface for reading and writing sensors using either I^2C or SPI protocols.

  For an example of DM usage, look at the *rli_project.c* in the *Generated_Code/ISF/RLI/Source* directory.

2.2 Subscription options

2.2.1 Native subscriptions and explicit converts

ISF Sensor Adapters are designed to provide direct control of the sensor. An application subscribes to a sensor and configures that sensor using the adapter's `Configure` method, either directly or via the *isf_dsa_direct* `configure_sensor()` API. These configuration parameters are written to the sensor as a result. If an application subscribes to the same sensor multiple times each subsequent subscription overwrites the previous subscription's configuration. A developer might run into this situation when dealing with combination sensors, such as the FXOS8700C (accelerometer + magnetometer) and MPL3115A2 (pressure/altimeter + temperature).

When multiple subscriptions are configured for the same sensor, the following actions occur internally:

- The sensor configurations are both written to the sensor such that the last one written takes precedence.
- Multiple periodic callbacks are registered with the Bus Manager's periodic timer function to read sensor data. This is inefficient because not only does it force two separate bus transactions, but managing the additional callback adds unnecessary overhead to the Bus Manager as well.

The most efficient solution in this situation is to use a single subscription that returns both sensor quantities together. This is easily done by configuring the sensor subscription in Processor Expert to use a **Sensor Data Format** of `Raw Sensor Format` and a **Sensor Data Type** of `Native Sensor Output`.

To retrieve the desired individual sensor quantities from this **native sample**, the Sensor Adapter's `Convert()` method is used. Each Sensor Adapter's `Convert()` method takes in a **native sample**, a requested data type and format, and returns a converted sample in the format requested.

If coding by hand, this might look like:

```
isf_SubscriptionSettings_t mySettings;
isf_SensorHandle_t         mySensorHandle;
uint8                      mySensorId = 1;
LWEVENT_STRUCT             mySensorEvent;
uint32                     myEventField = 1;
mpl3115_DataBuffer_t       myRawSampleData;

_lwevent_create(&mySensorEvent, LWEVENT_AUTO_CLEAR);

init_sensor(mySensorId, &mySensorHandle & mySensorEvent, myEventField);

mySettings.resultType     = TYPE_NATIVE_SENSOR_DATA_TYPE;
mySettings.nSamplePeriod  = 10000;   // 100 Hz in microseconds period
mySettings.nFifoDepth     = 1;

configure_sensor(
  &mySensorHandle,
  &mySettings,
  &myRawSampleData
);

start_sensor(&mySensorHandle);

for (;;)
{
  isf_KiloPascals1D_float_t     myPressureSample;
  isf_DegreesCelsius1D_float_t  myTemperatureSample;

  _lwevent_wait_for(&mySensorEvent, myEventField, FALSE, NULL);

  convert_sensor_data(
          &mySensorHandle,
          TYPE_PRESSURE,
          DSA_RESULT_TYPE_ENG_FLOAT,
          &myPressureSample,
          &myRawSampleData
  );
  convert_sensor_data(
          &mySensorHandle,
          TYPE_TEMPERATURE,
          DSA_RESULT_TYPE_ENG_FLOAT,
          &myTemperatureSample,
          &myRawSampleData
  );
 }
```

2.2.2 Any sensor versus all sensors

When entering the `_lwevent_wait_for()` loop in the embedded application's main loop, there is a choice between waiting for all event flags to be set or waiting for any individual event flag in the event group to be set. Addition information is available in the reference manual for the underlying RTOS.

With respect to ISF and waiting for sensor data, this translates to waiting for all sensors to have new data available or coming out of the loop when any single sensor has new data available.

Consider the example where we have subscribed to accelerometer, magnetometer, and gyroscope data and would like to know there is new data for each of these sensors before calling the sensor computation routines. Of course, this works best when all three sensor subscriptions are at the same rate and we just want to wait for all sensor data to arrive for that period before starting the computations. In that case, we can configure the **Sensor Signaling Method** in the Main Application Settings group of the EmbeddedApp component in Processor Expert, to a value of `AllSensors`.

This generates a call to `_lwevent_wait_for()` with the EventFlags for all the sensor subscriptions OR'd together.

If, however, we have subscribed to sensors at different rates and wish to handle each sensor's data as it appears, we can configure the **Sensor Signaling** to a value of `AnySensor`. This causes the boolean flag passed to `lwevent_wait()` to be set to FALSE. This means wait for any bit in bit_mask to be set.

To determine which flag (or flags- if more than one flag is set at one time) the function `_lwevent_get_signalled()` can be called to determine which code to execute.

2.3 Using Device Messaging directly

To use Device Messaging, a device channel must be initialized. A channel corresponds to a physical connection instance in the device. For instance, if there are two I^2C buses, and one SPI bus, then there would be three channels configured in *isf_sysconf_comms.c*. To work with a channel, a channel descriptor is required. To obtain a channel descriptor, allocate a `dm_ChannelDescriptor_t` variable and pass it, along with the channel ID of the channel to be used, to the `dm_channel_init()` function.

```
dm_ChannelDescriptor_t  myChannelDescriptor;
uint32                  channelID = 1;

dm_channel_init(channelID, &myChannelDescriptor);
```

Once the channel is open, the `dm_device_open()` function can be used to select a device on that channel for the communication.

```
i2c_device_t           deviceInfo = {0};
dm_DeviceDescriptor_t  myDeviceDescriptor;

deviceInfo.address = 0x1E;  //FXOS8700 7-bit I2C address


dm_device_open(&myChannelDescriptor, &deviceInfo, &myDeviceDescriptor);
```

The device descriptor can now be used to read and write to the device.

```
dm_device_write(
    &myDeviceDescriptor, startAddress, apWriteBuffer, aNbyteWrite, aNbyteWrite
);

dm_device_read(&myDeviceDescriptor, startAddress, apReadBuffer, aNbyteRead, aNbyteRead);
```

The code in the RLI component is a good example demonstrating many of the features discussed in this User Guide. To view this code, generate a project with the RLI component added. The code is in the *Generated_Code/ISF/RLI/Source/rli_project.c* file.

2.4 Configuring the Generic Analog Sensor Adapter

The *Generic Analog Sensor Adapter* component is designed to support a number of analog sensors by using GPIOs to control signal inputs and the Analog-to-Digital converter (ADC) to sample the analog signals from the sensor. In general, the *Generic Analog Sensor Adapter* supports analog accelerometers (for example, the FXLN83xx family of 3-axis accelerometers) and pressure sensors (for example, the MPXV5004DP).

The *Generic Analog Sensor Adapter, or ISF_KSDK_Sensor_Generic_Analog_Adapter,* is brought into an ISF Processor Expert (PEx) project like any other Sensor Adapter (via the System Sensor Configuration in the *ISF_KSDK_Core* component). The *Generic Analog Sensor Adapter* component provides a pull-down list of analog sensor types, which are the currently supported sensors. Selecting one of those sensors causes additional automated inclusion of the GPIOs for the specific control signals for the chosen sensor. Subscription to the sensors by the *ISF_KSDK_EmbApp* is also consistent with other sensors (although the sensor type is **Analog**).

The *Generic Analog Sensor Adapter* component allocates the KSDK *fsl_adc16* driver through an inherited component. This is the driver interface to the ADC, which is used by the generated code.

Code for the *Generic Analog Sensor Adapter* is created in the *Sources* directory. Typically, it is named *fsl_Sensor_<Component Name>_Functions.c* and *fsl_Sensor_<Component Name>_Functions.h*. These files provide the mapping between the generic adapter and the specific sensor selected. One example of this is the generation of the `apply_sensor_specific_configurations()` function. This function tailors the configuration table for the specific mapping between the ADC outputs and the acceleration or pressure units. It is called from the Sensor Adapter `Configure()` function**.**

NOTE:

Files in the *Sources* directory are not necessarily deleted or updated properly if the *Generic Analog Sensor Adapter* component sensor type is changed. In that case, the user should manually delete these files associated with the component and regenerate code.

The target sensor is sampled based on an arbitrary sampling period in the sensor subscription. The callback initiates a sampling by the ADC, reads the results, converts them to the desired units, and then pushes them into the software FIFO.

2.4.1 Modifying ISF for an unsupported analog sensor

It is possible to start with one of the supported sensors (for example, FXLN83xxQ), generate the code via PEx, then manually modify the resulting application to work with an otherwise unsupported sensor.

The basic steps to do this are:

1. Select the sensor type that is closest to the desired sensor
2. Modify the ADC and GPIO configurations to match the pin configurations of the sensor
3. Generate code using Processor Expert.
4. Modify the conversion factors in the `apply_sensor_specific_configurations()` function to match the expected output ranges of the sensor.

2.5 Interrupt-driven sensor subscriptions

This section describes the FXLS8952C Sensor Adapter (*ISF_KSDK_Sensor_FXLS8952_Accelerometer*), the infrastructure it uses in ISF and the KSDK to support interrupt-driven sensor reading, and then how the user could extend a given sensor to be interrupt-driven by manually utilizing the same infrastructure.

2.5.1 FXLS8952C in interrupt driven mode

ISF v2.2 supports the FXLS8952C[1] through either polled or interrupt-driven sensor data reading. In the *PEx* component, there is a **Use Interrupts** property which can be set to TRUE to enable interrupt-driven reading. This action creates an inherited *fsl_gpio* PEx component, which has to be manually initialized to the pin associated with the INT1 from the FRDM-STBC-AGM02 shield board. In addition, the configuration enables the Event Handler Service in the *ISF_KSDK_Core*. Plus, the user needs to update the *fsl_gpio* component to generate the PORTD IRQ (which is the port associated with INT1) handler, enable the installation of the ISR, and then manually add the call in *Sources/Events.c* as follows:

```
void ExtInt1_PORTD_IRQHandler(void)
{
    PORT_HAL_ClearPortIntFlag(PORTD_BASE_PTR);
    ExtInt1_IRQHandler();
}
```

By tracing through the generated ISF code, it is easy to see the code involved with sensor interrupt handling. During system initialization, the interrupt handler is installed by this code in *Cpu.c*:

```
/*! ExtInt1 Auto initialization start */
OSA_InstallIntHandler(PORTD_IRQn,  ExtInt1_PORTD_IRQHandler);
GPIO_DRV_Init(ExtInt1_InpConfig0,NULL);
/*! ExtInt1 Auto initialization end */
```

1 This part will be released in the future.

During sensor configuration, the FXLS8952C Sensor Adapter registers its periodic callback, which reads the sensor data, with the *Event Handler* service.

```
isf_EventHandler_RegisterEvent(&fsl_fxls8952_i2c_3D_accel_PeriodicCallback,
pSensorHandle);
```

Once the sensor is configured and running, it generates an interrupt signal every time a sample is ready to be read. The microcontroller vectors that interrupt to the `ExtInt1_PORTD_IRQHandler`, which clears the interrupt and calls the `ExtInt1_IRQHandler` (i.e., the ISF generated handler). This ISF handler sends an OSA Event signal to the Event Handler to notify the previously registered callback. The Event Handler task waits on these events and calls the corresponding registered callback.

2.5.2 Manually converting a sensor adapter to an interrupt-driven adapter

In addition, any of the existing Sensor Adapters can be manually configured to run in an interrupt-driven mode, as described for the FXLS8952C in Section 2.5.1. This section outlines the steps involved with using the FXOS8700C as an example.

1. Instantiate the FXOS8700C in the default project through the *ISF_KSDK_Core* and *ISF_KSDK_EmbApp* components as normal.
2. Add the *fsl_gpio* PEx component to the project and configure for the specific port/pin assigned to the interrupt pin from the FXOS8700C sensor.
   a. Select the desired pin. For example, there is a defined pin called FXOS8700CQ_INT2 that maps to PTC13 on the FRDM-K64F.
   b. In the **Events** section, select **code generation for the corresponding IRQ handler** and select **Install Interrupts**.
3. Generate the code and compile to make sure there are no errors.
4. Edit *Events.c* to add an interrupt handler which sends a unique signal to the Event Handler service. The interrupt handler is modelled after `ExtInt1_IRQHandler()`, as shown in Section 2.5.1.
5. In the *fsl_FXOS8700_i2c_3D_accel.c* file and the `Configure()` function, replace the call to `bm_register_periodic_callback` with a registration to the Event Handler for the equivalent callback using the selected, unique signal identified in step 4.
6. Update the register configuration in the *FXOS8700_1.c* file, `FXOS8700C_1_Sensor_Specific_Config` to enable the generation of the interrupt.

2.6 Using Register-Level Interface (RLI)

The ISF Register-Level Interface is used to read and write arbitrary sensor registers for any connected I^2C sensor.

RLI is implemented as an ISF application that defines two custom commands, one implementing a read command and the other a write command.

2.7 Using KIT tool for RLI

The Kinetis Interface Tool (KIT) PC GUI provides a tab that allows use of the commands to select a sensor device as well as read and write registers. For example, the FXAS21002C gyroscope has an I^2C 7–bit slave address of 0x20. On the KIT's **RLI** tab, enter **20** in the **Choose part-enter part slave address** field, and then press **Choose**.

To read the part's **WHO_AM_I** register (FXAS21002 WHO_AM_I is 0x0C), enter **# of Bytes to Read** as **1**, with **Start Address** as **C**, and then select **Read Register(s)**.

The read response displays in the **Register Responses** text box.

1 Registers read
D7

2.8 General serial programming against the RLI app

RLI is not limited for use with the KIT only. Anything capable of reading and writing serial data can also be used.

The protocol is as follows:

- All ISF packets are encapsulated between 0x7E bytes.
- The first byte after the 0x7E in a packet determines the packet protocol type. 0x01 is the Command/Response Protocol. The RLI commands use the Command/Response protocol.
- The next byte in a packet is the AppID. This is used by the ISF Command Interpreter to direct the packet to the correct application. Typically, an ISF executable image contains a mailbox app with AppID = 1, and an EmbeddedApp with AppID = 2. If an RLI component is used, the RLI app has AppID = 3.

  Note, though, the AppIDs are assigned in their order of addition to the project. If a developer adds the RLI component to their Processor Expert project prior to adding their Embedded App component, then the AppIDs are assigned as RLIApp = 2, and EmbApp = 3.

- Following the AppID byte in the packet is a Command byte. The commands are enumerated in *isf_ci.h.*

```
typedef enum
{
  CI_CMD_READ_VERSION     = 0,
  CI_CMD_READ_CONFIG,
  CI_CMD_WRITE_CONFIG,
  CI_CMD_READ_APP_DATA,
  CI_CMD_UPDATE_QUICKREAD,
  CI_CMD_READ_APP_STATUS,
  CI_CMD_RESET_APP,
  CI_CMD_DEVICE_WRITE,
  CI_CMD_DEVICE_READ,
  CI_CMD_GET_APP_SUBSCRIPTION,
  CI_CMD_WRITE_SREC_FLASH,
  CI_CMD_MAX = 128
} ci_commands_enum;
```

The RLI App uses the `CI_WRITE_CONFIG` command to configure the slave address for communication.

The `CI_CMD_WRITE_CONFIG` command takes an offset into the configuration register, a number of bytes to write, and the actual bytes to write.

NOTE:
All register values are expressed in hexadecimal notation.

Therefore, the command packet that must be sent to the RLI to choose the slave address is:

```
7E 01 03 02 00 02 20 00 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
02	CI_CMD_WRITE_CONFIG
00	Configuration register, offset zero
02	Writing two bytes
20	16-bit device address LSB
00	16-bit device address MSB
7E	End of packet

The RLI app responds with:

```
7E 01 03 80 02 02 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
80	Cmd status value
02	Command word echo
02	Number of bytes written
7E	End of packet

The Command status value consists of a Command-Complete bit in bit 7 (MSBit) and a status value in bits 0–6, where a zero status value indicates **SUCCESS/NO_ERROR**.

Once the slave address has been successfully selected, sensor read/write operations can be performed. To read the **WHO_AM_I** value from the FXAS21002C, a 1-byte read from register 0x0C must be performed.

The command the RLI App uses to perform device read operations is `CI_CMD_DEVICE_READ` with value of 8.

In order to read from an I²C device, it is necessary to write the address/register-offset from which to read to the device so it can fetch the value and return it. There the register read command requires a 1 byte write followed by an N-byte read.

Therefore, the command that must be sent is:

```
7E 01 03 08 01 01 0C 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
08	CI_CMD_DEVICE_READ
01	Number of I²C bytes to write
01	Number of I²C bytes to read
0C	The byte to write (the register offset to read from)
7E	End of packet

The RLI app responds with:

```
7E 01 03 80 01 01 D7 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
80	Cmd status value
01	Number of bytes requested
01	Number of bytes returned
D7	The register value read
7E	End of packet

To write a register, the `CI_CMD_DEVICE_WRITE` command is used.

The read and write commands use the same command format. The write command is accomplished by sending an N-byte write followed by a 0 byte read. To write the value 0x02 to the **CTRL_REG1** register at 0x13 using the `CI_CMD_DEVICE_WRITE` command, with enum value of 7, the command is:

```
7E 01 03 07 02 00 13 02 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
07	CI_CMD_DEVICE_WRITE
02	Number of I^2C bytes to write
00	Number of I^2C bytes to read
13	The first byte to write (the register offset to write to)
02	The register value to write
7E	End of packet

The RLI app responds with:

```
7E 01 03 80 00 00 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
80	Cmd status value
00	Number of bytes requested
00	Number of bytes returned
7E	End of packet

As mentioned previously, this serial communication can be performed by any capable connected device.

The following example contains a simple Python 2.7 program that performs the reads and writes described in this section. Note that this is not intended to be production-quality code, but merely gives an example of using the PySerial module in Python to interact with the RLI app. It is important to note that the full packet escaping logic is not implemented in this example. Full escaping would require checking each byte transmitted to ensure that a raw 0x7E is not included as part of a packet's payload. Any 0x7E bytes occurring in the data must be escaped by sending a 0x7D before the byte and by clearing bit 6 in the byte. Any 0x7D in the payload must be escaped in the same manner. When reading bytes from the ISF Command Interpreter including data from the RLI app, incoming bytes must be checked for escaped characters by removing any 0x7Ds received in the input stream and by setting bit 6 in the following byte.

```
import serial
import time
import binascii

def signed16(msb, lsb):
    v = (msb<<8) + lsb
    return -(v & 0x8000) | (v & 0x7FFF)

def get_packet() :
    t=0
    r=bytearray()

    while True:
        n = ser.inWaiting()
        if n>0 :
            r += ser.read(n)
            t += n
            if (t > 2) and (r[t-1] == 0x7E):
                break
    return bytearray(r)

selectSlave = bytearray(b'\x7E\x01\x03\x02\x00\x02\x20\x00\x7E')
readWhoAmI  = bytearray(b'\x7E\x01\x03\x08\x01\x01\x0C\x7E')
writeActive = bytearray(b'\x7E\x01\x03\x07\x02\x00\x13\x02\x7E')
readData    = bytearray(b'\x7E\x01\x03\x08\x01\x06\x01\x7E')

ser = serial.Serial(port='COM53',baudrate=115200)

ser.flushInput()

print "Setting RLI Slave"
ser.write(selectSlave)

b = get_packet()
print "Slave Set returned: ",binascii.hexlify(b)

print "Reading WHO AM I  value"
ser.write(readWhoAmI)
b = get_packet()
print "WHO AM I returned: 0x%02x" % b[6]

print "Setting device to active"
ser.write(writeActive)
b = get_packet()
print "Active command returned ", binascii.hexlify(b)

print "Reading XYZ data"
ser.write(readData)
b = get_packet()
print "Read:", signed16(b[6],b[7]), signed16(b[8],b[9]), signed16(b[10],b[11])

print "Done."
ser.close()
```

The RLI app supports two additional commands, which have command values 0x0B and 0x0C, hexadecimal values for 11 and 12.

These two commands provide a periodic read capability. Sending command 0x0B requests that the most recently executed `CI_CMD_DEVICE_READ` command be executed at the requested frequency.

Command 0x0C requests that the periodic reads be discontinued.

The **Start Periodic Reads** command requires specification of the read period in microseconds and the Command Interpreter Stream number in which to return the data.

The **Start Periodic Reads** command is composed as follows:

```
7E 01 03 0B 00 05 00 00 27 10 01 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
0B	Start periodic reads via Command = `CONFIGURE_PERIODIC_READS` (0B)
00	Offset
05	Length
00	32 bit period MSB
00	32 bit period MCB
27	32 bit period LCB
10	32 bit period LSB
01	Stream number to use
7E	End of packet

The command is asking to repeat the previous read command at a rate of 100 Hz (0x0000 2710 microseconds period) using Stream 1.

Table 1. Example **Start Periodic Read** commands at various periods/frequencies for Stream 1

Command	RLI Period (µs)		RLI Frequency (Hz)
	Decimal	Hex	
7E 01 03 0B 00 05 **00 00 04 E2** 01 7E	1250	(0x0000 04E2)	800
7E 01 03 0B 00 05 **00 00 09 C4** 01 7E	2500	(0x0000 09C4)	400
7E 01 03 0B 00 05 **00 00 13 88** 01 7E	5000	(0x0000 1388)	200
7E 01 03 0B 00 05 **00 00 27 10** 01 7E	10,000	(0x0000 2710)	100
7E 01 03 0B 00 05 **00 00 4E 20** 01 7E	20,000	(0x0000 4E20)	50
7E 01 03 0B 00 05 **00 00 9C 40** 01 7E	40,000	(0x0000 9C40)	25
7E 01 03 0B 00 05 **00 01 38 80** 01 7E	80,000	(0x0001 3880)	12.5
7E 01 03 0B 00 05 **00 02 71 00** 01 7E	160,000	(0x0020 7100)	6.25

To discontinue performing periodic reads, the following command is sent:

```
7E 01 03 0C 00 00 7E
```

Value	Description
7E	Start of packet
01	Command/Response protocol
03	AppID 3 – RLI App
0C	Stop periodic reads
00	Offset
00	Length
7E	End of packet

2.9 Creating your own sensor adapter

Often, it is easier to use the Device Messaging interface to talk directly to new or unsupported sensors, but you can also create a full ISF Sensor Adapter for any sensor.

The easiest way to create your own Sensor Adapter for a new or unsupported sensor, is to start with one of the provided adapters that most closely matches the new sensor. In other words, if a new I^2C accelerometer is to be integrated, choose one of the existing accelerometer adapters to copy and modify. To find this code, generate a project that includes the adapter to be copied. After Processor Expert code generation is complete, the adapter code can be found in the *Generated_Code/ISF/Sensor-Specific* directory.

The adapters are implemented using the ISF Digital Sensor Abstraction interface as defined in *isf_dsa_adapter.h* and consist of routines to be called for initialization, configuration, start, periodic read, stop, and shutdown.

Initialization should contain code to perform any one-time, initial processing. This typically includes initialization of internal data structures. In addition, it may sometimes include some interrogation of the sensor itself to determine available feature sets or to retrieve trim information, and so forth.

The `Configuration` method should contain code to configure a sensor based on the requested subscription settings.

If the sensor has Active and Standby states, the `StartData` function should set the sensor to its active state and enable interrupts, if using interrupt notifications. Alternatively, if the Bus Manager is used for polling, the function tells the BM to start servicing the callback.

The `EndData` function can put the sensor back to its Standby mode and stop the BM callbacks or disable the sensor's external interrupt.

The `Shutdown` function can set the sensor to its lowest configurable power state.

The `Calibrate` function is unused by ISF and can be safely left unimplemented.

The `Convert` function should take in a native data sample from the sensor and convert it to any supported standard sensor type.

3 Using ISF features in your code

3.1 Bus Manager

The Bus Manager (BM) service provided by the ISF Core is available for application-level timing of periodic or oneshot timing events down to the microsecond level. This section shows both scenarios and explains the steps for using the BM in the application.

The BM callback functions registered by the user are executed at the BM task priority level sequentially, thus the real-time requirements of the callback function should be considered when using this feature.

A periodic callback can be registered to the BM as follows:

```
#include "isf_bm.h"

bm_callback_token_t    token;
uint32                 period;

token = bm_register_periodic_callback( period, (bm_callback_t *)callback ,
(void *)&userData);
```

where:

- **`token`** is the identifier of the registered callback
- **`period`** is the time between callbacks in microseconds
- **`callback`** is a function as follows:
  `typedef void (bm_callback_t)(void *)`
- **`userData`** is a pointer to any state data required by the callback (NULL is fine).

This call sets up the service, but does not start the callback. That is done as follows:

```
bm_start(false, token);
```

Once the `bm_start()` is called, the BM schedules the period to elapse. When it expires, the callback is executed in the order that it was registered.

One example of what can be done in a callback is to flash an LED using KSDK GPIO driver:

```
void callback( uint32* pLEDState )
{

   // Toggle the LED
   GPIO_DRV_WritePinOutput(LED_BLUE, *pLEDState);
   *pLEDState += 1;
}
```

When it is time to terminate the callback, the user can execute the following:

```
bm_stop(token);
callbackRet = bm_unregister_callback(token);
```

In order to do a *oneshot*, the callback must execute the above code itself. For example:

```
void callback( bm_callback_token_t *pToken )
{

   // Do something once

   // terminate the callback
   bm_stop(token);
   callbackRet = bm_unregister_callback(token);
}
```

3.2 Event Handler

The Event Handler (EH) service, provided by the ISF Core, provides the capability to execute registered callback routines when events occur. Callback execution occurs on the EventHandler's task allowing the application to proceed with its other processing and letting the OS handle the task preemption according to each task's configured priority.

The Bus Manager currently executes its callback in the Bus Manager task itself. This is acceptable when the registered callback is very short, non-blocking and deterministic. However, if a long-running task is registered with the Bus Manager, its execution could cause unwanted consequences to the timely execution of other registered callbacks. In this case, an approach might be to simply set an event in the BM callback and register the long running call with the EventHandler instead. This allows the BM to process the periodic timer callback very quickly. It just sets the event flag and returns. The event flag causes the EH to come out of its event wait and performs the callback in its task instead of the BM's. It is important to note that the same problem exists with a long-running callback in the EH causing other EH callbacks to wait until it is finished. The EH still must be used with care, but it does allow several non-interfering application functions to share a single thread rather than each taking its own.

4 Communicating with the PC

4.1 Using streams

In ISF v2.0, a more convenient streaming protocol called *CI Streaming* was introduced.

This is a brief discussion of CI streaming, including examples. The full details of the CI streaming protocol including programming APIs can be found in Section 4 of the ISF v2.2 Software Reference Manual.

An ISF embedded application project by default contains three host-addressable applications.

Table 2**. ISF_KSDK_EmbApp Project Application IDs**

AppID	Name	Description
0	Device Info	ISF Built-in: Returns Device Information
1	ISF Command/Response (C/R) App	ISF Built-in: Manages Mailbox register configuration
2	ISF Embedded App	The main application

APPID 0 – Device Info
The Device Info application responds to a single, non-standard command.

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

The application returns the Device Info record. The contents of the Device Info record are described by the `device_info_t` type in *isf.h*.

APPID 1 – The Mailbox App
The mailbox application implements a deprecated Streaming Protocol (previously known as Quick-Read). Use the Streaming Protocol discussed in Section 4.1.1.1, CI Streaming examples, to stream data from the device.

APPID 2 – ISF_KSDK_EmbApp
An embedded application, by default, contains two sets of data that are accessible by the host: the configuration data and the sensor data. The configuration data is readable/writable and is used to set the available features of the application. The host can access the configuration data by using the ISF Command Interpreter (CI) `CI_CMD_READ_CONFIG` and `CI_CMD_WRITE_CONFIG` commands.

The sensor output data contains the timestamp when the data sample was taken along with the sensor data for each sensor subscription. The host can access the information by using the CI `CI_CMD_READ_APP_DATA` command. Note: if the host has set the application to streaming mode, the application sends back the sensor data asynchronously at the prescribed sample rate.

See Table 3 for the application's data layout and description.

Table 3. **ISF EmbApp (AppID 2) – typical sensor output data layout**

Byte Offset	Name	Description
0 1 2 3	ISFEmbApp_APP_TS_BITS_31_24 ISFEmbApp_APP_TS_BITS_23_16 ISFEmbApp_APP_TS_BITS_15_8 ISFEmbApp_APP_TS_BITS_7_0	TimeStamp tickCount[0] (MSB) TimeStamp tickCount[1] TimeStamp tickCount[2] TimeStamp tickCount[3] (LSB) 32-bit timestamp value in microseconds. This is the recorded time when the sensor data sample is taken. The time value comes from a continuously running timer counter that started at system power up.
4 5 6 7	ISFEmbApp_APP_X_ACCEL_BITS_31_24 ISFEmbApp_APP_X_ACCEL_BITS_23_16 ISFEmbApp_APP_X_ACCEL_BITS_15_8 ISFEmbApp_APP_X_ACCEL_BITS_7_0	X data[0] (MSB) X data[1] X data[2] X data[3] (LSB)
8 9 10 11	ISFEmbApp_APP_Y_ACCEL_BITS_31_24 ISFEmbApp_APP_Y_ACCEL_BITS_23_16 ISFEmbApp_APP_Y_ACCEL_BITS_15_8 ISFEmbApp_APP_Y_ACCEL_BITS_7_0	Y data[0] (MSB) Y data[1] Y data[2] Y data[3] (LSB)
12 13 14 15	ISFEmbApp_APP_Z_ACCEL_BITS_31_24 ISFEmbApp_APP_Z_ACCEL_BITS_23_16 ISFEmbApp_APP_Z_ACCEL_BITS_15_8 ISFEmbApp_APP_Z_ACCEL_BITS_7_0	Z data[0] (MSB) Z data[1] Z data[2] Z data[3] (LSB)

Notes:

- The sensor output data fields for any *ISF_KSDK_EmbApp* are documented in *App1_types.h* in the `App1SensorData_t` type definition.
- In general, the layout consists of the sensor output data for each subscription laid out in subscription order. This data is usually a 4-byte timestamp plus four bytes of either fixed or floating point data for each axis supported by the sensor, when the subscription is configured to use fixed point or floating point.
- The layout could also be a native sensor output structure as defined in the corresponding Sensor Adapter Include file as defined in the corresponding standard types file in the *Generated Code/SensorGeneric* project directory

4.1.1 Reference set of host command sequences

4.1.1.1 Example `ISF_KSDK_EmbApp` app host commands

Note that the commands shown here represent only the payload portion of the packet sent over the serial port. When using the Kinetis Interface Tool (KIT) included in the ISF installer, the 0x7E or 7E packet start and end bytes are added automatically and do not have to be entered explicitly.

For example, to reset AppID 2, entering `01 02 06 00 00` in KIT's **Command Interface** tab **Bytes to Write** field results in `7E 01 02 06 00 00 7E` being sent to the MCU.

Reset application (CI_CMD_RESET_APP)

Send: `01 02 06 00 00`

Write application configuration (CI_CMD_WRITE_CONFIG)

State	Value
Unsubscribed	`01 02 02 00 01 00`
Subscribed	`01 02 02 00 01 01`
Oneshot	`01 02 02 00 01 02`
Streaming	`01 02 02 00 01 03`

CI Streaming example

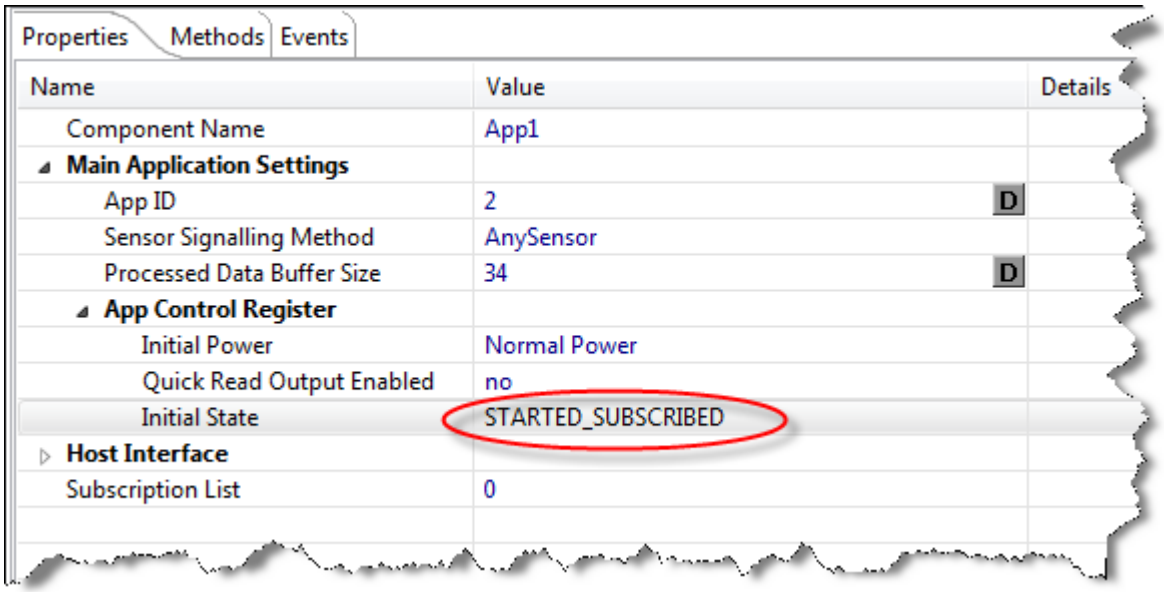
In Streaming Mode, the application sends data back to the host whenever new sensor data is available. This example assumes that an Embedded App has been configured with three sensors. Subscription 1 is for accelerometer data, subscription 2 is for gyroscope data and subscription 3 is for pressure sensor data. The subscriptions are all returning engineering fixed data which means that sample data consists of a 4-byte timestamp followed by four bytes per axis of sensor data.

Suppose the following sample rates are configured in the Embedded App:

Sensor type	Frequency
Accelerometer	100 Hz
Gyroscope	200 Hz
Pressure sensor	1 Hz

In Processor Expert, setting Embedded App Control Register Initial State to **STARTED_SUBSCRIBED** is convenient as well. See Figure 1.

Figure 1. PEx properties GUI setting Embedded App Control Register state



After generating code, building and executing the application, the following example command sequence can be used:

1. Reset application.
   a. Reset the Command/Response application.

   Send: `01 01 06 00 00`

   Receive: `01 01 80 00 00`

	Value	Description
Send	01	Protocol ID = 1
	01	AppID = 1
	06	Command = CI_CMD_RESET_APP
	00	Offset = 0
	00	Length = 0
Receive	01	Protocol ID = 1
	01	AppID = 1
	80	Command Complete with status 0
	00	NULL
	00	NULL

b. Reset the Embedded application (AppID 2)

Send: `01 02 06 00 00`

Receive: `01 02 80 00 00`

	Value	Description
Send	01	Protocol ID = 1
	02	AppID = 2
	06	Command = CI_CMD_RESET_APP
	00	Length Requested = 0
	00	Offset = 0
Receive	01	Protocol ID = 1
	02	AppID = 2
	80	Command Complete with status 0
	00	NULL
	00	NULL

2. Define three streams for sensor data.

a. Set up Stream ID A1 to return accelerometer data.

Send: `02 03 A1 01 01 01 00 10 00 00`

Receive: `02 80 03 00 00`

	Value	Description
Send	02	Protocol ID = 2
	03	Create Stream command (3)
	A1	Stream ID = 1
	01	Only one stream element in this stream
	01	Trigger bitmask = 1. This message is sent whenever Element 1 is updated.
	01	First element is from Dataset ID 1 (Subscription 1/accelerometer data)
	00 10	Length = 16
	00 00	Offset = 0
Receive	02	Protocol ID = 2
	80	Command Complete with status 0
	03	Create Stream command (3)
	00	Length MSB
	00	Length LSB

b. Set up Stream ID A2 to return gyroscope data.

Send: `02 03 A2 01 01 02 00 10 00 00`

Receive: `02 80 03 00 00`

	Value	Description
Send	02	Protocol ID = 2
	03	Create Stream command (3)
	A2	Stream ID = A2
	01	Only one stream element in this stream
	01	Trigger bitmask = 1. This message is sent whenever Element 1 is updated.
	02	First element is from Dataset ID 2 (Subscription 2/gyroscope data)
	00 10	Length = 16
	00 00	Offset = 0
Receive	02	Protocol ID = 2
	80	Command Complete with status = 0
	03	Create stream command (3)
	00	Length MSB
	00	Length LSB

c. Set up Stream ID A3 to return pressure data.

Send: `02 03 A3 01 01 03 00 08 00 00`

Receive: `02 80 03 00 00`

	Value	Description
Send	02	Protocol ID = 2
	03	Create Stream command (3)
	A3	Stream ID = A3
	01	Only one stream element in this stream
	01	Trigger bitmask = 1. This message is sent whenever Element 1 is updated.
	03	First element is from Dataset ID 3 (Subscription 3/pressure data)
	00 08	Length = 8. Note: Length is only 8 bytes because there is only one axis for pressure/altitude.
	00 00	Offset = 0
Receive	02	Protocol ID = 2
	80	Command Complete with status 0
	03	Create stream command (3)
	00	Length MSB
	00	Length LSB

3. Enable CI Streaming.

Send: `02 01`

Receive: `02 80 01 00 00`

	Value	Description
Send	02	Protocol ID = 2
	01	CI_CMD_STREAM_ENABLE_DATA_UPDATE command
Receive	02	Protocol ID = 2
	80	Command Complete with status 0
	01	CI_CMD_STREAM_ENABLE_DATA_UPDATE command
	00	Length MSB
	00	Length LSB

4. Stop streaming.

Send: `02 02`

Receive: `02 80 02 00 00`

	Value	Description
Send	02	Protocol ID = 2
	02	CI_CMD_STREAM_DISABLE_DATA_UPDATE command
Receive	02	Protocol ID = 2
	80	Command Complete with status 0
	02	CI_CMD_STREAM_DISABLE_DATA_UPDATE command
	00	Length MSB
	00	Length LSB

5. Delete a stream (Stream ID 2 – gyroscope data).

Send: `02 04 A2`

Receive: `02 80 04 00 00`

	Value	Description
Send	02	Protocol ID = 2
	04	Delete Stream command (4)
	A2	Stream ID to delete = A2
Receive	02	Protocol ID = 2
	80	Command Complete with status = 0
	04	Delete Stream command (4)
	00	Length MSB
	00	Length LSB

6. Delete all streams and reset streaming.

Send: `02 00`

Receive: `02 80 00 00 00`

	Value	Description
Send	02	Protocol ID = 2
	00	Stream Reset command (0)
Receive	02	Protocol ID = 2
	80	Command Complete with status 0
	00	Stream Reset command (0)
	00	Length MSB
	00	Length LSB

7. Create a new stream (Stream ID 1) with all three sensors in one stream triggered on Gyro data update.

Send: `02 03 01 03 02 01 00 10 00 00 02 00 10 00 00 03 00 08 00 00`

Receive: `02 80 03 00 00`

	Value	Description	
Send	02	Protocol ID = 2	
	03	Create Stream command (3)	
	01	Stream ID = 1	
	03	3 stream elements in this stream	
	02	Triggered on element 2 (bitmask)	
		Element list: (dataset, 16-bit length, 16-bit offset)	
	01	Stream element 1	Dataset 1
	00 10		Length = 16
	00 00		Offset = 0
	02	Stream element 2	Dataset 2
	00 10		Length = 16
	00 00		Offset = 0
	03	Stream element 3	Dataset 3
	00 08		Length = 8
	00 00		Offset = 0
Receive	02	Protocol ID = 2	
	80	Command Complete with status 0	
	03	Create Stream command (3)	
	00	Length MSB	
	00	Length LSB	

8. Start streaming.

Send: `02 01`

Receive: `02 80 01 00 00`

	Value	Field Name
Send	02	Protocol ID = 2
	01	Start Streaming command (1)
Receive	02	Protocol ID = 2
	80	Command Complete with status 0
	01	Start streaming command (1)
	00	Length MSB
	00	Length LSB

9. Delete a stream (Stream ID 1 – data from all three sensors).

Send: `02 04 01`

Receive: `02 80 04 00 00`

	Value	Field Name
Send	02	Protocol ID = 2
	04	Delete Stream command (4)
	01	Stream ID to delete = 1
Receive	02	Protocol ID = 2
	80	Command Complete with status 0
	04	Delete Stream command (4)
	00	Length MSB
	00	Length LSB

10. Create a new stream with all three sensors, but this time trigger off of pressure updates (1 Hz).

Send: `02 03 01 03 04 01 00 10 00 00 02 00 10 00 00 03 00 08 00 00`

Receive: `02 80 03 00 00`

	Value	Field Name	
Send	02	Protocol ID = 2	
	03	Create Stream command (3)	
	01	Stream ID = 1	
	03	3 stream elements in this stream	
	04	Trigger mask is 04 which is 0b000 0100 indicating that element 3 is the trigger.	
		Element list: (dataset, 16-bit length, 16-bit offset)	
	01	Stream element 1	Dataset 1
	00 10		Length = 16
	00 00		Offset = 0
	02	Stream element 2	Dataset 2
	00 10		Length = 16
	00 00		Offset = 0
	03	Stream element 3	Dataset 3
	00 08		Length = 8
	00 00		Offset = 0
Receive	02	Protocol ID = 2	
	80	Command Complete with status = 0	
	03	Create Stream command (3)	
	00	Length MSB	
	00	Length LSB	

A set of common Streaming commands is provided with the KIT. To use this file, on the **Command Interface** tab, use the **Command List: Open File** button and choose *Command Interpreter Streaming Commands.txt.*

4.1.2 Calling StreamUpdate() in App1_ProcessData()

The ISF embedded app components (*ISF_KSDK_EmbApp* and *ISF_KSDK_BasicApp*) are both very effective at getting sensor data out to the host for graphing. But what happens when a developer wants to perform some calculations using that sensor data and produce new and different outputs?

The Embedded App component creates a function `App1_ProcessData()` function in the file *App1_Functions.c* file in the *Sources* directory.

The `App1_ProcessData()` function is called in the Embedded App's sensor data loop and allows developers to implement custom sensor data processing logic whenever new sensor data arrives.

After implementing the custom sensor code in ProcessData, it is desirable to make the results available to the remote host via the ISF Command Interpreter's streaming interface.

To do this, update a streaming dataset with a pointer to the computed data.

It is important to choose a Stream ID that does not conflict with any of the sensor subscription datasets. Each sensor subscription makes its data available in a dataset with a Dataset ID that matches its subscription number. For example, subscription 1 publishes data in Dataset 1. Subscription 2 publishes data in Dataset 2. Additional information explaining Stream IDs and Dataset IDs can be found in the Streaming protocol section of the ISF Software Reference Manual.

Suppose we wish to compute the vector magnitude of the accelerometer data from subscription 1. Assume there are no other sensor subscriptions in the Embedded App. This allows us to publish the computed vector magnitude in Dataset 2.

We could, therefore, code an `App1_ProcessData()` function similar to:

```
void App1_ProcessData(void* pProcessedDataBuffer, int32_t signalledEvents)
{
    // Cast the void * pointer to the specific embedded application data type.
    // This new pointer should be used to access sensor data and deliver results
    // to insure type safety.
    App1SensorData_t *pProcessedData = (App1SensorData_t *)pProcessedDataBuffer;
    /******Write your code here*******/
    float vmag;
    float x,y,z;

    x = pProcessedData->rawAccelerometerData_Sub0[0].accel[0];
    y = pProcessedData->rawAccelerometerData_Sub0[0].accel[1];
    z = pProcessedData->rawAccelerometerData_Sub0[0].accel[2];

    vmag = (float)(sqrt(x*x + y*y + z*z));

    isf_ci_stream_update_data(2, sizeof(vmag), 0, (uint8 *)&vmag);
}
```

If more than one sensor subscription exists and the `Sensor Signaling` method is set to `AnySensor`, then `App1_ProcessData()` is invoked whenever new data is available for any sensor. In order to execute different code based on which sensor data has arrived, a check may be placed in `App1_ProcessData()` as shown below:

```
void App1_ProcessData(void* pProcessedDataBuffer, int32_t signalledEvents)
{
    // Cast the void * pointer to the specific embedded application data type.
    // This new pointer should be used to access sensor data and deliver results
    // to insure type safety.
    App1SensorData_t *pProcessedData = (App1SensorData_t *)pProcessedDataBuffer;
    /******Write your code here*******/
    if (signalledEvents & App1_Accelerometer0_DATA_READY_EVENT) // Accelerometer event.
    {
        float vmag;
        float x,y,z;

        x = pProcessedData->rawAccelerometerData_Sub0[0].accel[0];
        y = pProcessedData->rawAccelerometerData_Sub0[0].accel[1];
        z = pProcessedData->rawAccelerometerData_Sub0[0].accel[2];

        vmag = (float)(sqrt(x*x + y*y + z*z));

        isf_ci_stream_update_data(2, sizeof(vmag), 0, (uint8 *)&vmag);
    }

    if (signalledEvents & App1_Gyrometer1_DATA_READY_EVENT) // Gyrometer event.
    {
        float vmag;
        float x,y,z;

        x = pProcessedData->rawGyrometerData_Sub1[0].angularVelocity[0];
        y = pProcessedData->rawGyrometerData_Sub1[0].angularVelocity[1];
        z = pProcessedData->rawGyrometerData_Sub1[0].angularVelocity[2];

        vmag = (float)(sqrt(x*x + y*y + z*z));

        // Only send a rotational vector magnitude update
        // when larger than some threshold.
        if ( vmag > 50.0 ) {
               isf_ci_stream_update_data(3, sizeof(vmag), 0, (uint8 *)&vmag);
        }
    }
}
```

4.1.3 Working with datasets and multiple streams

By default, both Embedded App components (*ISF_KSDK_EmbApp* and *ISF_KSDK_BasicApp*) maintain a one-to-one mapping of subscriptions to datasets. That is, the data from Subscription 1 is updated as Dataset 1, the data from Subscription 2 is updated as Dataset 2. But this is purely a convention used by these applications. In principle, an application can update as many different datasets as desired. It is also typical for the remote host to create a stream per dataset, but this also is only by convention. A stream can actually contain data from multiple datasets. For example, continuing with the code in Section 4.1.2, Dataset 3 gets updated when the vector magnitude of the gyroscope sample exceeds some threshold. Suppose the remote host wants to see the gyroscope data, accelerometer data and the vector magnitude, but only when the magnitude exceeds the threshold. To do so, the remote host defines a stream containing the accelerometer data from Dataset 1, the gyroscope data from Dataset 2 and the vector magnitude from Dataset 3, and sets the stream trigger to

send the stream only when the vector magnitude (Dataset 3) is updated. For demonstration purposes, have this data arrive in Stream 13. The accelerometer and gyroscope data are automatically updated into Datasets 1 and 2, respectively, by the Embedded App.

The stream command sent from the remote host would therefore be:

```
7E 02 03 0D 03 04 01 00 10 00 00 02 00 10 00 00 03 00 08 00 00 7E
```

Bit	Description	
7E	Start of Packet	
02	Streaming protocol ID = 2 (which is the Streaming Protocol)	
03	Create Stream command = 03	
0D	Creating stream = 13	
03	The stream contains three elements	
04	The Trigger mask (triggers on Element 3 0x04 = 0b00000100)	
01	Stream Element 1	Take data from Dataset 1 (accelerometer)
00		Number of bytes from Dataset 1 MSB
10		Number of bytes from Dataset 1 LSB
00		Offset within Dataset 1 MSB
00		Offset within Dataset 1 LSB
02	Stream Element 2	Take data from Dataset 2 (gyroscope)
00		Number of bytes from Dataset 2 MSB
10		Number of bytes from Dataset 2 LSB
00		Offset within Dataset 2 MSB
00		Offset within Dataset 2 LSB
03	Stream Element 3	Take data from Dataset 3
00		Number of bytes from Dataset 3 MSB
04		Number of bytes from Dataset 3 LSB
00		Offset within Dataset 3 MSB
00		Offset within Dataset 3 LSB
7E	End of Packet	

4.1.4 Starting streams automatically from the application

It is sometimes desirable to have an embedded application initialize and start streaming data immediately with no intervention by the remote host. This is accomplished by creating the embedded application with its initial state set to *STARTED_SUBSCRIBED* in Processor Expert. That starts the sensor sampling as soon as the app runs without needing the remote host to command the app to the *STARTED_SUBSCRIBED* state. All that remains then is to have the application automatically declare one or more streams and initiate streaming. To do this, some streaming code can be placed in the `App1_Initialization()` function, also in *App1_Functions.c*:

```
void App1_Initialization()
{
    /******Write your code here*******/
    uint8               triggerMask = 1;
    ci_stream_element_t element1;

    // Define a stream that sends the register data

    element1.datasetID = 3;
    element1.length    = sizeof(float); // vmag is a float
    element1.offset    = 0;

    // Create stream 5 which contains vmag from dataset 3 (accel)
    st = isf_ci_stream_create(5, 1, &triggerMask, &element1);

    element1.datasetID = 4;
    // Create stream 6 which contains vmag from dataset 4 (gyro)
    st = isf_ci_stream_create(6, 1, &triggerMask, &element1);

    //Now start streaming automatically
    isf_ci_stream_set_stream_enable();
}
```

4.2 Creating custom commands

It is usually possible to implement all remote host interface function by adding fields to the embedded applications configuration structure and then writing and setting those fields via the `CI_CMD_WRITE_CONFIG` command. However, it is sometimes convenient to define a new command for the application. Defining a new command is simply a matter of adding a new case to the switch statement in the application `App1_ci_app_callback()` function. This is done in Processor Expert by incrementing the **User Defined Host Commands** counter in the Host Interface section of the *ISF_KSDK_EmbApp* component configuration.

By default, the first callback generated is named `HCICB1_Callback()` and skeleton code is generated in the *Events.c* file in the *Sources* directory. Code within sections of this file identified by `/******Write your code here*******/` are not overwritten when Processor Expert regenerates code.

The Command Interpreter invokes the `HCICB1_Callback()` with two parameters:

```
void HCICB1_Callback(void* pHostPacket, void* pAppPacket)
```

The pHostPacket contains the information sent to the app from the remote host. The pAppPacket allows the embedded application to return information to the remote host.

The pHostPacket is a pointer to a structure defined as:

```
typedef struct
{
    uint8      appId;    /*! Application ID */
    uint8      cmd;      /*! Host command */
    uint16     offset;   /*! Offset into the application's data */
    uint8      byte_cnt; /*! Number of bytes to read/write */
  } ci_host_cmd_packet_t;
```

And the pAppPacket points to a structure of type:

```
typedef struct
{
  ci_rw_enum rw;         /*! Data transfer direction wrt to the app (CI_RW_READ, CI_RW_WRITE)
  uint8      bytes_xfer; /*! Number of bytes actually read from or written to the host */
  uint8      bytes_left; /*! Number of bytes left to read from or write to the host */
} ci_app_resp_packet_t;
```

To retrieve the incoming information from the remote host, the isf_ci_app_read() function is used:

```
bytesRead = isf_ci_app_read(pHostPacket->appId, pHostPacket->byte_cnt, &readBuffer);
```

To send data back to the remote host, the isf_ci_app_write() function is used:

```
bytesXfer = isf_ci_app_write(pHostPacket->appId, numBytes, &dataBuffer);
```

In either case, the pAppPacket structure must always be updated to reflect the action taken by setting the pAppPacket->rw value, and the pAppPacket->bytes_xfer value. In the case of reading data from the remote host using the isf_ci_app_read() function, the pAppPacket->rw value must be set to CI_RW_READ, and the pAppPacket->bytes_xfer must be set to the number of bytes read. Conveniently, this value is returned by isf_ci_app_read() making the idiomatic code:

```
pAppPacket->rw = CI_RW_READ;
pAppPacket->bytes_xfer = isf_ci_app_read(pHostPacket->appId,
pHostPacket->byte_cnt, &readBuffer);
```

Or for writing data to the remote host:

```
pAppPacket->rw = CI_RW_WRITE;
pAppPacket->bytes_xfer = isf_ci_app_write(pHostPacket->appId, numBytes,
&dataBuffer);
```

As an example, suppose we are running the App1_ProcessData() function but wish to make the detection threshold for the vmag stream update settable by the remote host. This can be accomplished in two ways.

First, the threshold value could be added to the app configuration data structure and use the standard CI_CMD_WRITE_CONFIG command.

The current `App1AppSettings_t` is defined in *App1_types.h* as:

```
typedef struct {
    uint8                       control;                    /* Current application state */
    isf_SubscriptionSettings_t AccelerometerSettings_Sub0; /* Sensor subscription data */
    isf_SubscriptionSettings_t GyrometerSettings_Sub1;     /* Sensor subscription data */
} App1AppSettings_t;
```

Add the extra configuration value to the bottom:

```
typedef struct {
    uint8                       control;                    /* Current application state */
    isf_SubscriptionSettings_t AccelerometerSettings_Sub0; /* Sensor subscription data */
    isf_SubscriptionSettings_t GyrometerSettings_Sub1;     /* Sensor subscription data */
    float                       magReportThreshold;

} App1AppSettings_t;
```

And update our `App1_ProcessData()` function:

```
extern App1AppInstance_t App1AppInstance;

void App1_ProcessData(void* pProcessedDataBuffer, int32_t signalledEvents)
{
    // Cast the void * pointer to the specific embedded application data type.
    // This new pointer should be used to access sensor data and deliver results
    //  to insure type safety.
    App1SensorData_t *pProcessedData = (App1SensorData_t *)pProcessedDataBuffer;
    /******Write your code here*******/
    if (signalledEvents & App1_Accelerometer0_DATA_READY_EVENT) // Accelerometer event.
    {
        float vmag;
        float x,y,z;

        x = pProcessedData->rawAccelerometerData_Sub0[0].accel[0];
        y = pProcessedData->rawAccelerometerData_Sub0[0].accel[1];
        z = pProcessedData->rawAccelerometerData_Sub0[0].accel[2];

        vmag = (float)(sqrt(x*x + y*y + z*z));

        isf_ci_stream_update_data(2, sizeof(vmag), 0, &vmag);
    }

    if (signalledEvents & App1_Gyrometer1_DATA_READY_EVENT) // Gyrometer event.
    {
        float vmag;
        float x,y,z;

        x = pProcessedData->rawGyrometerData_Sub1[0].angularVelocity[0];
        y = pProcessedData->rawGyrometerData_Sub1[0].angularVelocity[1];
        z = pProcessedData->rawGyrometerData_Sub1[0].angularVelocity[2];

        vmag = (float)(sqrt(x*x + y*y + z*z));

        // Only send a rotational vector magnitude update
        // when larger than some threshold.
        if ( vmag > App1AppInstance.settings.magReportThreshold ) {
            isf_ci_stream_update_data(3, sizeof(vmag), 0, &vmag);
        }
    }
}
```

The remote host can issue a `CI_CMD_WRITE_CONFIG` command to offset 28 with four bytes of a floating point value in the endianness of the embedded microprocessor to set the `magReportThreshold` value.

The offset of 28 was determined by calculating the offset into the `App1AppSettings_t` structure of the start of the **magReportThreshold** field.

One point to note is that to add the **magReportThreshold** field to the `App1AppSettings_t` structure, it was necessary to edit the *App1_Types.h* file. This file will be overwritten the next time Processor Expert generates code, thus removing the addition. This can be worked around and ways to do this are discussed in Section 6.

Alternatively, a new command could be specifically created to set the threshold as discussed above.

The implemented callback function might look like:

```
extern float magReportThreshold;

typedef union {
    long  lval;
    float fval;
} floatBytes_t;

void HCICB1_Callback(void* pHostPacket, void* pAppPacket)
{
/******Write your code here*******/
    uint8_t             cmdBuf[10];
    isf_status_t        st;
    floatBytes_t        floatBytes;

    /* get real types back instead of the void * pointers */
    ci_host_cmd_packet_t *pHP = (ci_host_cmd_packet_t*)pHostPacket;
    ci_app_resp_packet_t *pAP = (ci_app_resp_packet_t*)pAppPacket;

    pAP->rw = CI_RW_READ;
    pAP->bytes_xfer =
        (uint8)isf_ci_app_read(
            pHP->appId, (uint32)pHP->byte_cnt, cmdBuf
        );
    floatBytes.lval = (cmdBuf[0]<<24)|(cmdBuf[1]<<16)|(cmdBuf[2]<<8)|(cmdBuf[3]);

    magReportThreshold = floatBytes.fval;
}
```

The order in which the bytes of the float are sent can now be made according to preference, as long as they are put back together properly. One could even decide to send an ASCII string containing the value and parse that back into a float inside the `HCICB1_Callback()` function.

And of course the `magReportThreshold` variable can now be declared as a global in *App1_Functions.c:*

```
float magReportThreshold = 0.0;
```

The test in `App1_ProcessData()` can now be modified to simply:

```
if ( vmag > magReportThreshold ) {
    isf_ci_stream_update_data(3, sizeof(vmag), 0, (uint8 *)&vmag);
}
```

5 Main application flow

5.1 Embedded app

This section discusses the basic operational flow of the *ISF_KSDK_EmbApp* component.

The *ISF_KSDK_EmbApp* component generates its main code to the *App1.c*, *App1.h*, and *App1_Types.h* files in the *Generated_Code* folder.

Each *ISF_KSDK_EmbApp* instance executes in a task. The OS task is configured to run `App1_Task()` which invokes the `isf_lib_init()` function, waits for all ISF components to complete initialization and then invokes `App1_MainTask()`.

The file *App1_Functions.c* is provided to allow the developer to execute custom initialization logic. The `App1_MainTask()` function calls the `App1_Initialization()` routine within it.

Once initialization is complete, the sensor subscriptions specified in the *Processor Expert Sensor Subscription* section of the embedded application are set up.

The generated subscription settings code is executed, the event data structures are created and initialized, the software FIFOs used to contain the subscription sample data are initialized and then each sensor is initialized.

After sensor initialization is complete, the application moves to the subscription state specified in the Processor Expert component configuration. By default, this is **STOPPED_UNSUBSCRIBED**, but may be modified to **STARTED_SUBSCRIBED**, or **STOPPED_SUBSCRIBED**.

It then enters a *forever* loop waiting for new sensor samples to arrive using an OS event synchronization mechanism. Execution continues depending on whether the event wait was configured to wait for **ANY_SENSOR** or **ALL_SENSORS** in Processor Expert.

If the application was configured to go into a **STOPPED_UNSUBSCRIBED** or **STOPPED_SUBSCRIBED** state, then no sensor data is received until a command from the remote host is received to set the application state to **STARTED_SUBSCRIBED**.

For example, the command:

```
7E 01 02 02 00 01 52 7E
```

sets the application to *SUBSCRIBED_STARTED*, and is decoded as follows

Value	Description
7E	Start of Packet
01	Command/Response protocol
02	AppID 2– Embedded App
02	CI_CMD_WRITE_CONFIG
00	Write offset zero in the Config App's config buffer
01	Number of bytes to write
52	The byte to write 0x52 sets the subscription state to STARTED_SUBSCRIBED
7E	End of packet

When the remote host sends this command, it is received by the Command Interpreter (CI). The CI examines the destination AppID and invokes the corresponding registered CI Callback. The EmbApp's callback is `App1_ci_app_callback()` and is found in *App1.c*. The registered callback list is found in *ISFCore1.c* in the `ci_callback[]` array. The array entries are ordered by AppID.

The `App1_ci_app_callback()` function can then take different actions based on the command sent. The *ISF_KSDK_EmbApp* implementation uses a large switch statement, based on the command.

It is suggested you browse through the `App1_ci_app_callback()` function to observe how it operates. Try generating code with no User-Defined Host Commands and then with one or more User-Defined Host Commands to see how the generated code changes.

Once the flow of sensor data has started, the event wait call returns and execution flow enters the main sensor loop logic.

Inside the main sensor loop, the `signalledEvents` word is checked to determine which sensors have new data available. For each of these sensors, the data is retrieved from its registered FIFO and its corresponding streaming dataset is updated. This triggers the Command Interpreter to update and potentially send a stream update packet to the remote host, for any stream containing data from that dataset.

After these sensor subscription streams have all been updated, `App1_ProcessData()` is invoked where user-specific sensor data processing is performed.

After `App1_ProcessData()` returns, the loop is complete and it cycles back to the event, *wait* at the top of the loop, to await the arrival of new data.

5.2 Basic app

The *ISF_KSDK_BasicApp* component provides a much simpler, and powerful, application than does the *ISF_KSDK_EmbApp* component.

The *ISF_KSDK_BasicApp* component organizes all its sensor subscription information in a large array indexed by subscription number. This allows the app to iterate through the array to perform actions on all subscriptions.

It generally operates like the *ISF_KSDK_EmbApp* component described in Section 5.1, as it:

1. Synchronizes on the ISF thread initializations
2. Creates and initializes the event used for sensor subscription notifications
3. Invokes the `BasicApp1_Initialization()` routine, where developers can place their own custom initialization logic
4. It loops through app configured sensor subscriptions calling:
   a. `isf_fifo_init()`
   b. `init_sensor()`
   c. `configure_sensor()`

At this point, where the *ISF_KSDK_EmbApp* calls `App1_GotoState()`, the *ISF_KSDK_BasicApp* simply starts the sensors by calling `start_sensor()` for all subscriptions and then enters the main sensor loop. Processing of sensor data coming out of the event wait is similar to the *ISF_KSDK_EmbApp* processing. For each sensor with new data available, the *ISF_KSDK_BasicApp* updates the corresponding stream dataset. Instead of invoking the user-defined `App1_ProcessData()` once, at the bottom of the loop, the *ISF_KSDK_BasicApp* invokes the `BasicApp1_OnAnySensor_Data_Ready()` function once for each sensor with new data available.

Parameters are passed to the function. The function is able to determine which sensor caused the invocation and which other sensors also have data currently available.

Unlike the *ISF_KSDK_EmbApp* with its subscription state machine and fully implemented Command-Interpreter callback, the *ISF_KSDK_BasicApp* provides a simpler Command Interpreter callback routine in the *BasicApp1_Functions.c* file. This file resides in the *Sources* directory and can thus be modified without fear of being overwritten during subsequent code generation operations. New commands, using switch cases, can be added directly by the developer without the need to configure additional commands in Processor Expert.

5.3 Integrating ISF into your own application

Besides the two application components, it is also possible to use ISF functionality from existing MQX or FreeRTOS applications. Instead of including either of the embedded application components in your project, add the ISF calls directly into the code of the existing project. You must ensure that `isf_lib_init(0)` is executed somewhere at the beginning of the application. For example, in the application's init routine, ensure that the ISF service tasks are properly created with task priorities that work with the rest of the application's tasks. At that point, it is okay to begin making sensor calls. The typical order might be:

1. Create an event to use for sensor data notification (`OSA_EventCreate()`)
2. Initialize a FIFO (`isf_fifo_init()`)
3. Initialize a sensor (`init_sensor()`)
4. Configure the sensor (`configure_sensor()`)
5. Start the sensor sampling (`start_sensor()`)
6. Wait for sensor data (`OSA_EventWait()`)
7. Work with sensor data
8. If applicable, pause the sensors whenever sensor data is not needed (`stop_sensor()`)
9. Shut down the sensor when the application is finished executing (`shutdown_sensor()`).

As a best practice, take the code from the *BasicApp* component and cut and paste as necessary into your own application.

6 Working with Processor Expert

6.1 Moving away from provided apps

A savvy developer can treat the Processor Expert generated code as simply a starting point for further development. After initial PEx code generation, the generated code can be copied out of the *Generated_Code* directory and into the *Sources* directory. Then, the files in the *Generated_Code* directory can be disabled or excluded from the build. This allows the code to be modified in any way the developer would like, without concerns for the code being overwritten if the code is regenerated, such as if a pin mux setting needs to be changed.

For example, if the Dataset ID being used for a particular sensor subscription needed to be changed in *Generated_Code/App1.c*, a developer could copy *App1.c* from *Generated_Code* to the *Sources* directory and then modify the code as needed. The Dataset ID character being changed is shown in the following code excerpts.

Change

```
isf_ci_stream_update_data(1,sizeof(App1AppInstance.data.rawAccelerometerData
_Sub0[0]), 0, (uint8 *)pData);
```

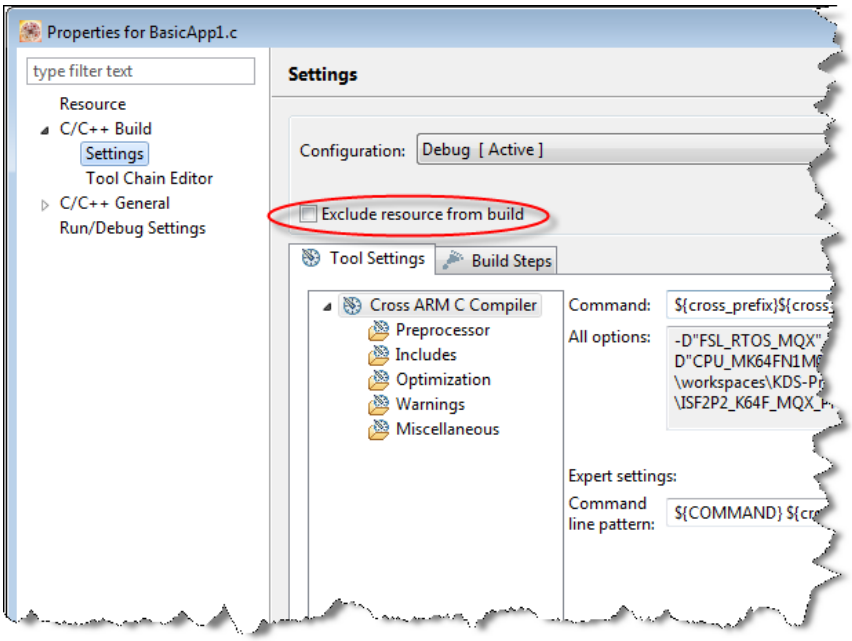
To

```
isf_ci_stream_update_data(6,sizeof(App1AppInstance.data.rawAccelerometerData
_Sub0[0]), 0, (uint8 *)pData);
```

Note: Characters in the previous code examples are boxed and highlighted for emphasis.

Files in the *Generated_Code* directory can be excluded from the build, to ensure KDS does not try to compile the same file twice. This can be done by selecting the file in KDS, right clicking for **Properties**, and checking the **Exclude resource from build** box in the **Build Settings**. See Figure 2.

Figure 2. *KDS BasicApp1.c* – Exclude Resource from Build checkbox



6.2 Using FreeRTOS

ISF v2.2 supports both MQX RTOS and FreeRTOS through the *fsl_os_abstraction* PEx component provided with the Kinetis SDK. The selection of RTOS is done through the **OS property** drop-down menu. Selection of FreeRTOS creates the *free_rtos* component through Component Development Environment (CDE) inheritance. This component must be configured to work correctly with ISF. In the Configuration parameters (*FreeRTOSConfig.h*), two **Heap memory configuration** values must be changed.

1. Set the default **Memory scheme** to **Alloc/Free**. (The default value is **Malloc**, which is the C stdlib implementation.)
2. Increase the **Total heap size** to 12 KB. The default Total heap size is 8 KB but FreeRTOS allocates all the task stacks on the heap, as a result, this needs to be increased to 12 KB.

It is important to note that FreeRTOS uses the upper 8 bits of all Events as internal flags. Therefore, the upper 8 bits are not available for user applications, and should be avoided.

6.3 Creating an ISF v2.2 project from scratch

ISF v2.2 provides example projects for a variety of NXP Freedom (FRDM) boards. However, it may become necessary for a user to create an ISF v2.2 project for a Kinetis MCU on a FRDM platform that does not have an example project. This section outlines the steps involved to get started with this process.

1. In KDS 3.0, go to **File->New->Kinetis Project**. Give the project a name, then click **Next**.

2. Select from the boards supported for the desired Kinetis MCU. Note: If the board is not supported by KDS 3.0, then this process becomes much more complicated. Select a board, click **Next**. There should be an option to select the KSDK 1.x version for the board, Select the **Processor Expert** option, then click **Finished.**

3. After the project is created, bring in the *ISF_KSDK_Core* in the normal way.

4. As Comm Channels (I^2C, SPI, UART) are added, consult the schematics for the target board to configure the underlying KSDK PEx components correctly.

When you have a working project, please consider posting to the ISF Community page for others to benefit from your efforts.

7 References

Resource	Description	Link
FRDM-STBC-AGM02[1]	Tool Summary Page	nxp.com/FRDM-STBC-AGM02
FXAS21002C	Product Summary Page	nxp.com/FXAS21002C
FXLN83xxQ	Product Summary Page	nxp.com/FXLN83XXQ
FXLS8952C[1]	Product Summary Page	nxp.com/FXLS8952C
FXOS8700C	Product Summary Page	nxp.com/FXOS8700CQ
MPL3115A2	Product Summary Page	nxp.com/MPL3115A2
MPXV5004DP	Product Summary Page	nxp.com/MPXV5004DP
ISF v2.1 website	Tool Summary Page	nxp.com/ISF-2.2-KINETIS
ISF v2.2 installer	Software	nxp.com/ISF-2.2-KINETIS
ISF v2.2 training videos	Training	nxp.com/ISF-2.2-KINETIS (Training tab)
ISF v2.2 Release Notes	Documentation	nxp.com/ISF-2.2-KINETIS
ISF v2.2 Software Reference Manual	Documentation	nxp.com/ISF-2.2-KINETIS
ISF v2.2 website	Tool Summary Page	nxp.com/ISF-2.2-KINETIS
Kinetis SDK website	Tool Summary Page	nxp.com/KSDK
Processor Expert website	Tool Summary Page	nxp.com/PROCESSOREXPERT

1. These products are not available at the time of the ISF v2.2 release.

8 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_UG
Revision 1.0, 02/2016