CELLULAR

Host Common Software Integration

RPMA product series

Application Note

Abstract

This document briefly describes the software interfaces and implementation considerations regarding the RPMA Host Common software component – a library of portable C code which facilitates all interactions between a host application and an u-blox RPMA Wireless module.

www.u-blox.com

UBX-16025680 - R03

Document Information		
Title	**Host Common Software Integration**	
Subtitle	RPMA product series	
Document type	Application Note	
Document number	UBX-16025680	
Revision, date	R03	19-Oct-2017
Disclosure restriction		

This document applies to the following products:

Product name
NANO-S100

Contents

1 Introduction

The document describes the software interfaces and implementation considerations regarding the RPMA Host Common software component – a library of portable C code which facilitates all interactions between a host application and a u-blox RPMA cellular module.

Detailed information on the module software, RPMA network components (e.g., Access Point, Gateway and back-end components) and the RPMA Total Reach air-link are beyond the scope of this document.

The following symbols are used to highlight important information within the document:

☞ An index finger points out key information pertaining to integration and performance.

⚠ **A warning symbol indicates actions that could negatively impact or damage the module.**

1.1 Node versus Module Terminology

For clarification throughout the document there are many references to the term "node", and commonly seen when referencing software code, functions, and labels. This term is equivalent to the term "module". In the past, Ingenu began using the term "node", and now this term has evolved into the term "module" with u-blox.

2 Integration Overview

2.1 Host Application Design

The RPMA network is designed to gather and process information from a number of remote sensors located in buildings, utilities, industrial, homes, or elsewhere. Each sensor node is composed of a u-blox RPMA module, a power source (e.g., a battery), at least one microcontroller, and multiple types of memory and several sensors. The Host Application is the firmware that runs on the end devices' microcontroller and is responsible for the following tasks:

- Collection and analysis of real time data from one or more sensors.
- Storage of real time data.
- Transmission of data to the backend via the u-blox RPMA module.
- Provisioning and debug support.

Figure 1 shows the typical high level view of an end device in a typical RPMA remote sensor application:

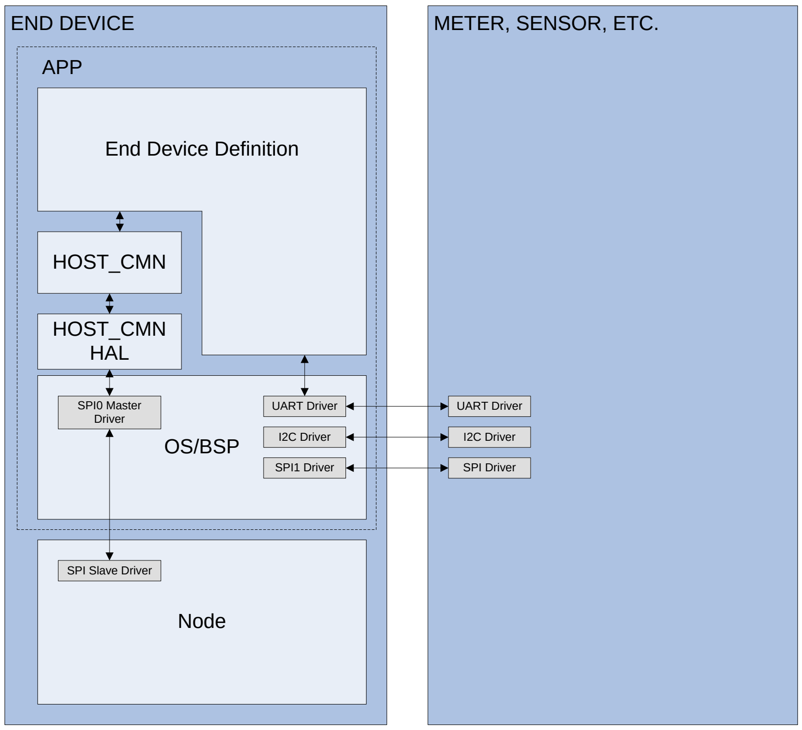

Figure 1: Typical application diagram

Other design considerations for the host application implementation include the following:

- Target microprocessor architecture
- Software build tools/environment
- Choice of Real Time Operating System (RTOS) to execute on target (including the choice to not utilize an embedded RTOS)
- Real-time requirements of the sensor hardware interfaces
- Line powered versus battery powered considerations (i.e., support for sleep modes)
- Memory requirements for program memory and/or data backlogs
- Over-the-Air (OTA) data packet format – Uplink and Downlink
- Asynchronous RPMA Network interactions (e.g., asynchronous alarm events that need to be reported to the backend)

2.2 Host Common software component

The Host Common software is an encapsulated source code component that will be built as part of the host application and provides a set of APIs that is applicable for most host firmware designs. Its primary purpose is to facilitate all exchanges between the Host Application, the u-blox RPMA module, and a PC host. Because of its relatively light-weight implementation (in complexity and in memory requirements), it can operate with or without an RTOS. Optional functionality is also available to the upper layers of the application for logging, firmware management, as well as reliable PC serial transfers.

Figure 2 illustrates how the Host Common component is organized within the host application.

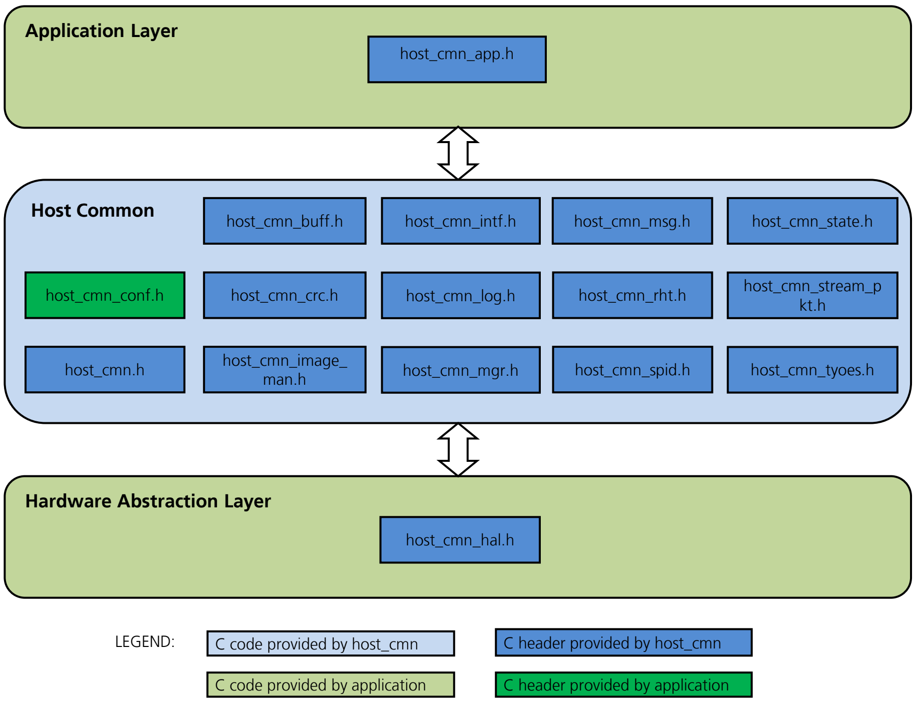

Figure 2: Host Common library organization

As depicted in Figure 2, the target application is responsible for implementing and/or configuring the following upper layer interfaces between the Application Layer and the Host Common software component – these interfaces are specified by the Host Common `host_cmn_app.h` header file and include the following:

- Host Common configuration definitions.
- Host Common application interfaces for RPMA network events (e.g., Pre-UI Notification, Module Reset, Module ACK, etc.).
- Host Common application interfaces for Rx OTA packet handlers.
- Host Common application interfaces for Tx OTA packet status handlers.
- Host Common application interfaces for Module State changes.

The target application is also responsible for implementing the lower layer interfaces between the Host Common software component and the microprocessor peripheral interfaces in the Hardware Abstraction Layer (HAL) – these interfaces are specified by the Host Common `host_cmn_hal.h` header file and include the following:

- Module Serial Peripheral Interface (SPI) Master Driver.
- Module Digital I/O Control Interface Driver.
- Second-based timer (non-blocking).
- Millisecond-based timer (blocking).
- Non-Volatile Memory (NVM) access functions (to support firmware image downloads).
- Host Serial Interface (e.g., UART, SPI, USB, or Ethernet).

Host Common also provides a number of management APIs that be called by either the application or the HAL layer and are prototyped in the `host_cmn_mgr.h` header file – they include the following:

- Host Common run-time initialization.
- Host Common run-time process function.
- Host Common Message Routing function.

This document provides detail on the implementation requirements of each of the listed interfaces.

There are also optional utilities that are also provided by the Host Common software component. They are all conditionally compiled based on the Host Common configuration header file (see section 3.1). These optional utilities are summarized as follows:

- Host-Specific Message Sink for extended set of application-specific messages through the RPMA message framework.
- Host Serial Logging capability.
- Host Application Firmware upgrade component (OTA and local).
- Reliable Host Transfer utility (for host serial data).
- Stream to packet utility (for parsing RPMA message data from host serial interface).
- Binary Diff utility (for larger firmware images).

As with the Host Common RPMA interfaces, this document goes into greater detail on each of aforementioned optional utilities.

2.3 Host Common directory structure

All of the Host Common source code and header files are typically provided to an external integration partner as part of the Reference Application Communication Module (rACM) software distribution. After the rACM source files have been extracted onto a local machine, the top level directory structure will be organized as follows:

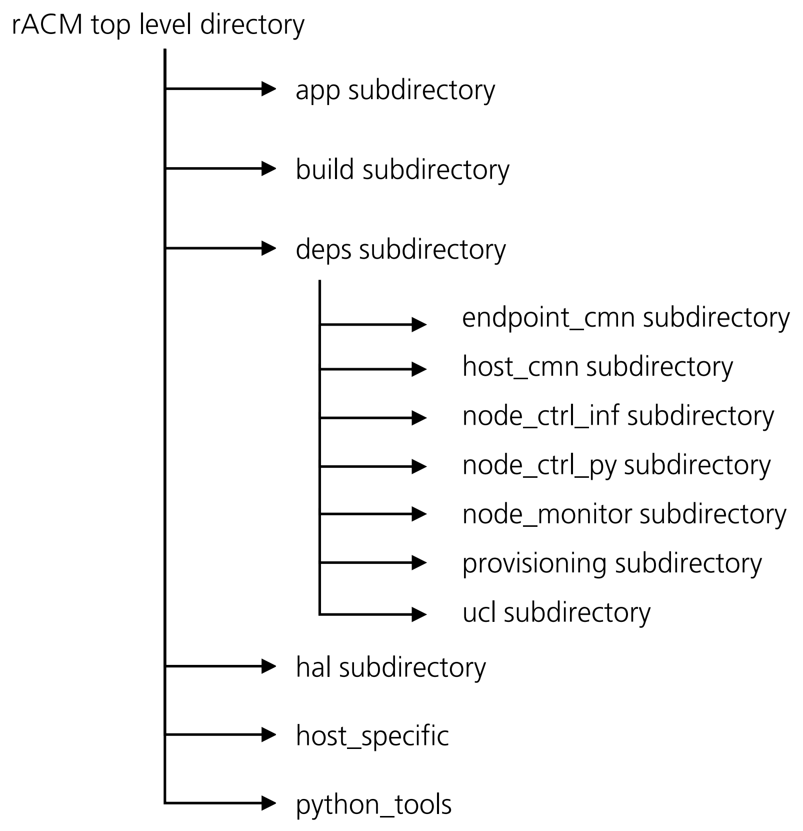

Figure 3: Reference host software directory structure

The Host Common software source, required header files, and supporting Host PC Python scripts are all located in the **deps** directory (deps is short for external dependencies). Additional details of the rACM software source subdirectories are considered beyond the scope of this document.

The external dependency subfolders that make up the Host Common components are summarized as follows:

- **endpoint_cmn** – Header files and Python scripts that are common between host and module targets.
- **host_cmn** – The encapsulated C source code used to implement the Host Common software component.
- **node_ctrl_inf** – Header files that are for the module control interfaces.
- **node_ctrl_py –**PC Host Python scripts used for the module control interfaces.
- **node_monitor –**PC Host Python scripts that implement the RPMA Module Monitor application (used for provisioning and field debugging of the RPMA Module – the details of this application are beyond the scope of this document).
- **Provisioning** – PC Host Python scripts used for provisioning the RPMA Network security key, performing module firmware updates, and configuring the module for a target network.
- **ucl –** PC Host Python scripts that implement the underlying transport layer necessary to communicate to the Host Application through the PC serial drivers.

RPMA integration partners that are developing their own Host Application software architecture and are using the Host Common as a standalone component should simply copy the entire `/deps` directory into their own top level directory. The next section details how to setup source and include paths to ensure the Host Common component will build and link with the rest of the application.

2.4 Setting up the build environment

Integrating the Host Common software component into a custom Host Application requires that the `host_cmn` source files are compiled and linked at the same time as the rest of the application. To do this, the build environment or Makefile must be updated to properly include the host common subdirectory .c and .h files.

The following source files must be included in the build environment:

- `<Project Directory>/deps/host_cmn/src/host_cmn_buff.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_crc.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_image_man.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_intf.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_log.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_mgr.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_msg.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_rht.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_spid.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_state.c`
- `<Project Directory>/deps/host_cmn/src/host_cmn_stream_pkt.c`

The build environment must also be setup for the following include paths:

- `<Project Directory>/deps/endpoint_cmn/src`
- `<Project Directory>/deps/host_cmn/src/`
- `<Project Directory>/deps/node_ctrl_inf`

For the compiling of the host common software components it is not enough the build environment set up with the source files and paths inclusion. The next section describes the remaining integration steps required for the target application to compile and execute on target.

For an example on how to properly set up the build environment to include the host common software components, the rACM /build subdirectory contains a GNU GCC Makefile that can be referenced (i.e., `gnu_host_app.mk`). In addition, if the target application is being developed using the IAR IDE, the rACM build folder contains project files for the ARM Cortex-M4 target that can be leveraged as a template for setting up the custom build environment.

2.5 Integration steps

The following list shows the development steps for integrating the host common software component into a custom host application:

- Configuring the Host Common software component operational parameters (see section 3.1)
- Implementing and unit-test the Host Common Hardware Abstraction Layer (HAL) interfaces (see section 3.2)
- Implementing and unit-test the Host serial interface (see section 3.3)
- Implementing the Host Common Application Layer interfaces (see section 3.4)
- Implementing the run-time initiation of the Host Common software component (see section 3.5)
- Implementing the run-time execution of the Host Common software component (see section 3.6)
- Implementing any application "hooks" for a serial interface that leverages the Host Common stream-to-packet feature (see section 3.7)

At this point, the Host Application and Host Common software component should be able to compile, link, and execute on target. If all of the software components are integrated successfully, the RPMA PC Host Python scripts should be able to process command/response commands to the Host Common component through Host interface. A command that best tests basic functionality of the Host Common software (regardless if the RPMA module is powered on or not) is the `HOST_VERSION` request issued from the PC Host command line as follows:

- `./ctrl.py -d <device> --no_node HOST_VERSION` (or),
- `./ctrl.py -i<ip_addr:port> --no_node HOST_VERSION`

The `-d` option would be used if the Host Interface was implemented using a UART, SPI or USB RS-232 emulation port. The `-i` option would be used if the Host Interface was implemented using an Ethernet socket. Regardless, the response should display the version information as configured in the Host Common configuration header file (see section 3.1).

When the application host powers on the module (through use of the `HOST_CMN_INTF_ PowerOnNode()` API) and sets the Host Operating Mode to PASSTHROUGH (see the next section), then full PC Host->Application Host->Node communications can be exercised by using:

- `./ctrl.py -d <device> GET_VERSION_REQ` (or),
- `./ctrl.py -i<ip_addr:port> GET_VERSION_REQ`

If the HAL components that make up the Host->Node Interface are properly implemented, then the response should display the version information and build stamp of the module firmware. If the basic connectivity can be established through use of the version request, then the full suite of RPMA Python tools can be used to control and configure the RPMA module (e.g., the On-Ramp Wireless Node Monitor application located in the rACM `/python_tools` directory).

Now that the mandatory Host Common components are established (i.e., the communication framework and interface drivers), then the following components can be integrated:

- Implement any application "hooks" with the Application Layer interfaces – this includes the definition and integration of all Over-the-Air messaging. These are largely driven by the high level requirements of the customer Host Application and are considered beyond the scope of this document. However, the u-blox or Ingenu support can help advise on the best implementation that is suited for the RPMA Network.
- Leverage the Host Common Tx SDU Enqueue request for any host-application user payloads that must be transmitted to the backend over the RPMA network (see section 3.7).
- If the application leverages the host serial interface for any application-specific message processing or is required to process module status indicators that are not covered by the application interfaces, then a host-

specific endpoint sink will need to be implemented and initialized with the host common communications framework (see section 3.8).

- If Image Manager functionality is enabled, then the remaining HAL interfaces and application "hooks" for managing firmware updates to on-target memory must be implemented (see section 3.9).
- If the application host needs to support deep sleep functionality and RAM data cannot persist across periods when the application is idle, then persistent memory support of the Host Common software component must be implemented (see section 3.10).
- If logging functionality through the host interface is enabled and used by the Host Application, then log level and functional masks must be defined for the application (see section 3.11). In addition, if the proprietary PACKED format is used (for memory constrained implementations), then the build environment must be modified to support the necessary generation of a dictionary file and dictionary build-time hash stamp.
- Test, provision, and deploy the full-featured application on the RPMA Network.

It needs to be mentioned that this document does not provide a comprehensive overview of all of the Host Common function interfaces – it only expands on the interfaces that are expected in a typical application implementation. Many of the function prototypes found in the `host_cmn` header files are called internally within the Host Common process loop and therefore are not detailed in this documentation.

2.6 Host Common communication framework

The communication framework that is implemented in the Host Common software component is responsible for routing messages to various endpoints on the Host Application as the following figure illustrates:

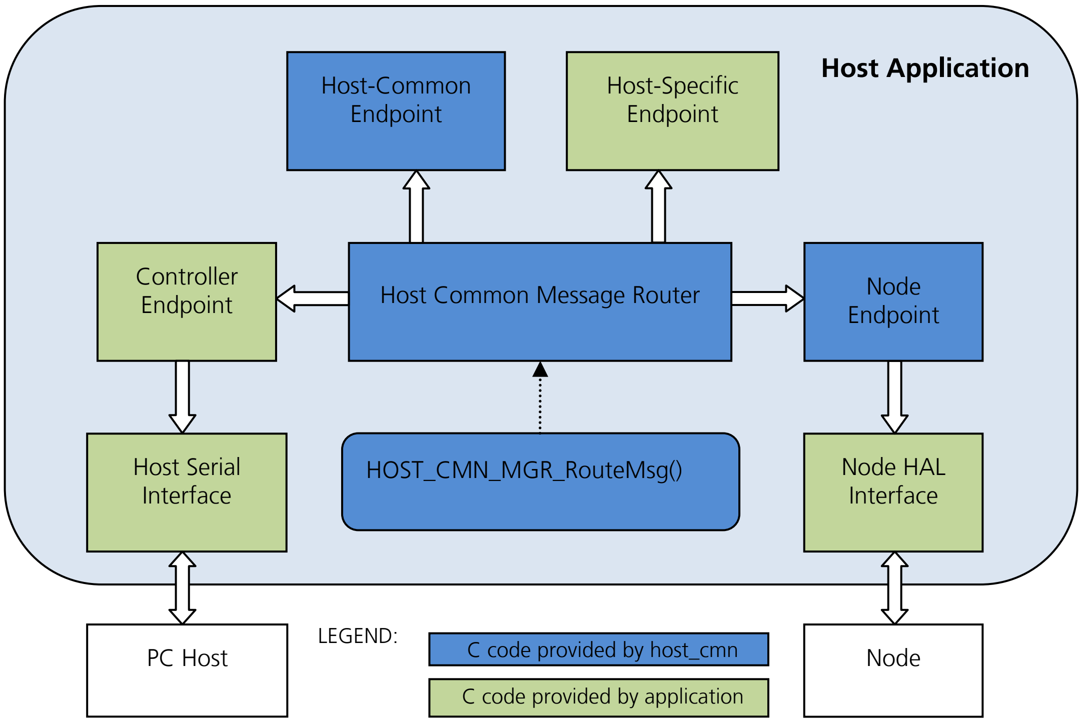

Figure 4: Host Common Communication Framework

The endpoints in this communication framework can be summarized as follows:

- **Controller Endpoint –** Essentially an external platform (i.e., laptop or PC) that is connected to the Host Application through the primary Host Serial Interface (e.g., UART, SPI, USB or Ethernet interface). The Python-based tools and provisioning scripts provided by Ingenu serves as the endpoint interface for message exchanges with the application.

- **Host-Common Endpoint –** Communication endpoint on the Host Application where all Ingenu Host message handlers are implemented (e.g., Host Software Upgrade, Host Version Request, etc.).
- **Host-Specific Endpoint –** Communication endpoint on the Host Application where extended application-specific or optional module message handlers are implemented (see section 3.7 for implementing an optional Host-Specific message sink).
- **Node Endpoint** – The RPMA Module accessed by the dedicated DSPI control interfaces implemented in the Host Common HAL layer.

Message exchanges on the communication framework is gated by a top level state in the Host Common software component called `OPERATING MODE`, which is defined by one of the following:

- **STANDALONE** - No external controller is assumed to be present – module messages are routed only to the internal host application code via the Host Endpoint message sinks. The application can power-down the external host interface for purposes of power-savings.
- **MONITOR** – The internal host application code is assumed to be the master, but an external controller is present and will receive duplicate copies of all messages generated by the module. This mode would typically be used during field debugging on a joined module using the On-Ramp Wireless Node Monitor application.
- **PASSTHROUGH** – The external controller is present and is the master – the internal host application code is not passed module messages, nor can it generate module destined messages. This mode would typically be set to ensure that the application host software does not interfere or disrupt factory ATE and/or provisioning of the device.
- **PROTECTED** – This mode is designed to support application security features where the external controller interface is not trusted. Message exchanges from the external controller interface are routed only to the host-specific message sink in the host endpoint. It is the applications' responsibility to allow authentication and unlocking of the interface to another operating mode.

☞ The Host-Specific message sink must be implemented to support this feature (see section 3.7).

The Host Common operating mode will default to STANDALONE, as this is the mode most applicable to typical fielded devices. The operating mode may be changed in one of two methods:

1. The `HOST_CMN_MSG_TYPE_OPERATING_MODE` message sent by the controller endpoint. or
2. The internal application calling `HOST_CMN_MGR_SetHostCmnOperatingMode()`.

For examples on setting the Host Common operating mode, the rACM software design will actually override the default STANDALONE mode at Power-On reset based on its' run-time configuration options (the `HOST_APP_ENDPT_Init()` function in `host_app_endpt_intf.c`).

3 Host Common interfaces

The following subsections summarize the APIs and data structures used in the integration of the Host Common software component. Section 2.5 lists the order in which the following interfaces should be integrated with the custom host application. A `readme.txt` file is also located in the `deps/host_cmn` subdirectory that provides an explanation of each functional component provided by the Host Common software component.

3.1 Host Common configuration header

The first step of any host common integration effort is to choose the build-time configuration options that the Host Common software component will use. High level requirements of a custom application will drive many of these configuration settings.

The first step is to make a copy of the `host_cmn_conf_sample.h` header file, renaming it to `host_cmn_conf.h` and ensuring that it is located in the build include path. The header file consists of a series of `#defines` that enable/disable and/or configure a number of different options in software.

The following bullet items summarize each mandatory configuration option as well as its' impact on the host application:

- **HOST_CMN_INCLUDE_SW_UPGRADE –** An optional component that enables host common support for firmware upgrades from the external controller endpoint (via the Host Serial Interface) or the RPMA OTA image upgrade process. This define should be commented out if the feature cannot be supported either due to memory constraints or is implemented in an alternative manner in the custom application. See section 3.8 for more details on the Image Manager feature.
- **HOST_CMN_INCLUDE_STREAMPKT** – An optional component that enables/compiles in the stream-to-packet utilities provided by Host Common (see section 3.12). It is highly recommended that the stream-to-packet utility be configured and leveraged to more easily integrate/support the full suite of RPMA Python-based utilities used to configure and diagnose the RPMA module. The only exception to this recommendation is if the physical connectivity to the external PC or control device is natively packetized – i.e. SPI, where chip select may delimit complete messages. Any stream oriented transport layer (serial, TCP, etc.) should include the stream to packet module.
- **HOST_CMN_INCLUDE_RHT –** An optional feature that compensates for lossy Host Serial Interfaces (e.g., IrDA, hub-powered USB serial convertors, etc.). This feature introduces a proprietary reliable delivery layer on top of each message exchange over the serial interface.

  ☞ This requires the **HOST_CMN_INCLUDE_STREAMPKT** option to be configured. The RPMA Python-based utilities must also enable the RHT v.2 protocol wrapper to properly utilize the reliable delivery protocol.
- **HOST_CMN_CONF_HAVE_STDINT –** If c-99 standard integer defines are available in the tool chain environment, this option should be defined to use those types.
- **HOST_CMN_TARGET_1_X –** By default, the Host Common software component is configured to support the RPMA 2.x module interface. This define should be set if using the RPMA 1.x module interfaces. u-blox support should be consulted on how to configure this option as well as which module Firmware version needs to be provisioned to the end device.
- **HOST_CMN_LITTLE_ENDIAN** (or) **HOST_CMN_BIG_ENDIAN** – The Endian-ness of the host CPU/tool chain (the ARM Cortex processor used on the rACM is little Endian).
- **HOST_CMN_APP_ID –** This 32-bit ID is used during the modules' network join/registration with the RPMA network and is used to identify the type of application the module is interfaced with. In an RPMA hosted network, this ID will need to be assigned in coordination with Ingenu project engineering.
- **HOST_CMN_GET_MAJOR_VERSION()**
- **HOST_CMN_GET_MINOR_VERSION()**
- **HOST_CMN_GET_POINT_VERSION() –** These three defines are used to report Host Application firmware version information and can be turned into function calls if need be. The rACM Host Common configuration header file uses the firmware version defines imported from `app/version.h`.

- **HOST_CMN_GET_VERSION_DESCRIPTOR() –** This define is used as a Host Application version string identifier (the maximum length is 8 characters) and can be turned into a function call if need be. The rACM Host Common configuration header file used the firmware string imported from `app/version.h`.
- **HOST_CMN_PRE_UI_ADVANCE_MS** – This option is used to define the time, in milliseconds, prior to the beginning of a scheduled update frame at which the module should notify the application of an impending update interval (via a Pre-Update Notification Indication). This is an important consideration that needs to factor in the CPU time necessary to format uplink user data message and time-sync considerations (see section 3.4 for further details on the Pre-UI notification event). Setting this value to zero will disable the Pre-Update Notification Indication for applications that only report asynchronous UI events (e.g., a remote alarm sensor).
- **HOST_CMN_GET_NODE_TYPE() –** This option is used to define the type of cellular module that the host application is integrated with (i.e., NANO-S1). The rACM module interfaces with the RPMA module, so this option is set as defined by `ORW_MSG_NODE_TYPE_UNODE`.
- **HOST_CMN_FORCE_1_X_SDU_ALIGNMENT –** 1.x module-based applications which do not automatically enforce 8 byte SDU length alignment should uncomment this define option. This will configure the Tx SDU Request API to pad all Tx SDUs to 8-byte alignment (with undefined data from RAM).
- **HOST_CMN_NODE_INACTIVITY_TIMEOUT_SECS** – This is a fail-safe mechanism that will generate an application layer event if the Host Common does not receive any of the following from the module in the allocated time: (1) System state messages, (2) Scan result indicators, or (3) Pre-UI Notification Indication – all of which are normally generated by the module during steady-state operations. In general, this timeout should never be exercised – it is a last ditch effort to recover a failing module through use of the **RESET_N** line. The rACM timeout value is set to a number of seconds in a 5-day period.

The following bullet items summarize each configuration option that is associated with the Image Manager functionality (see section 3.9):

- **HOST_CMN_IM_MAGIC_NUM() –** A 32-bit Image Identifier used between `host_cmn` and the module to accept the application host firmware upgrade. On hosted networks, this ID will typically be assigned by the Ingenu project engineering team.

☞ There are RPMA Python tools which generate an RPMA formatted firmware image that originates from the backend (i.e., it is the raw application binary preceded by an RPMA header file that initiates the firmware download process on the module). The magic number defined in the `host_cmn` configuration file must match the specified magic number, passed in as a command line parameter, in the Python tool utility script.

- **HOST_CMN_IM_INCREMENTAL_VERIFY –** Used when there is no memory mapped random access to a completed reassembled image. The configuration define will be uncommented such that `host_cmn` will do incremental CRC on "chunks" instead of the completed assembled image received from the image manager.
- **HOST_CMN_IM_BASE_PTR() –** Pointer to the image location in memory after a `host_cmn` image manager transfer. It can be turned into a function call if needed. The rACM design used the lower half of Program Flash for use by the Image Manager assembly area.
- **HOST_CMN_IM_MAX_IMAGE_SIZE** – The maximum allocated size of the Image Manager assembly area for use by the `host_cmn` Image Manager.

The following bullet items summarize each configuration option that is associated with the Reliable Host Transfer protocol:

- **HOST_CMN_CONF_RHT_BUFFER_SIZE –** The size in bytes of the outgoing message buffer from the application to the controller. It must be a multiple of 4 bytes.
- **HOST_CMN_CONF_RHT_ACK_TIMEOUT** – The amount of time in milliseconds to wait for an acknowledgement over the serial interface before retransmission.

The following bullet items summarize each configuration option that is associated with the Host Logging utility (see section 3.11):

- **HOST_CMN_LOG_LEVEL** – The output level mask used to filter which log strings are forwarded to the controller endpoint.
- **HOST_CMN_LOG_MASK** – The zone mask used to filter which log strings are forwarded to the controller endpoint.
- **HOST_CMN_LOG_IMPLEMENTATION** – The method of logging (i.e., standard library, raw, fixed or packed) used by the Host Logger.
- **HOST_CMN_LOG_BUILDSTAMP –** The Log build stamp used in each log string – used to ensure that a generated Host Application log dictionary file is synchronized with the Host Application that is executing on target (only applicable to the packed and fixed log implementations).

3.2 Hardware abstraction layer (HAL)

The next step in the integration process is the implementation of the driver interfaces in the Host Common software component required to manage all RPMA message exchanges with the module. Details on this proprietary RPMA message protocol, including control sequences, are handled internally within the Host Common component and are considered beyond the scope of this document. See NANO-S100 series System Integration Manual [2] for more details.

As mentioned in the Integration Overview section, the hardware driver resources required for managing the module are as follows:

- Module Serial Peripheral Interface (SPI) Master driver.
- Digital I/O driver for each of the module control lines (**RESET**, **MRQ**, **SRQ**, **SRDY**, **POWER_ON**, **TOUT**).
- Low-Resolution non-blocking second timer.
- High-Resolution context-blocking millisecond timer.

Furthermore, the custom application must implement all of these drivers according to the interface specification as described in NANO-S100 series System Integration Manual [2] that also summarizes these specifications on a pre-API basis.

The `host_cmn` header file, `deps/src/host_cmn_hal.h`, contains all of the function prototypes that the custom host application must implement, compile, and link with the rest of the application. Failure to implement any one of the function placeholders will prevent the application from building (i.e., undefined symbol during link time).

For an example implementation of the Host Common interfaces implemented on the Kinetis MCU (K20 variant), the rACM software implemented each of these interfaces within a single file in `hal/app_specific/host_cmn_hal_k20`. Obviously, porting the Host Common software component to another target will require drivers that interface correctly with the appropriate hardware interfaces.

The following subsections provide further details on each of these HAL interfaces in the order they are prototyped in `host_cmn_hal.h`:

3.2.1 HOST_CMN_HAL_init()

This function is called whenever the Host Common run-time initialization API is kicked off by the Host application (see section 3.5– it is expected that this occurs when coming out of any software reset). It is expected that the API will configure all of the hardware resources associated with the module HAL interface – this will typically include the following:

- I/O pin mux settings
- Peripheral initialization
- Digital clock enables

The following rules for the DSPI interface initialization apply (see NANO-S100 series System Integration Manual [2] for more details):

- Configure the SPI 3-wire interface with the module as a Master device on a dedicated SPI controller (i.e., no other devices may be present on the SPI bus).
- Transfers over the SPI interface must be configured for full-duplex 8-bit data transfers (i.e., TX data and RX data are both always sent/received on each clock edge).
- The baud rate over the SPI interface must be no greater than 10 Mb/sec.
- The Chip Select (**SPI_CS**) line on the SPI interface is active-low polarity.
- The SPI phase and polarity of the full-duplex **MISO/MOSI** full-duplex transfer must conform to the timing sequence diagram as shown in NANO-S100 series Data Sheet [1]: (i.e., CPOL=0, CPHA=0)

The following rules for the module control interfaces apply (see NANO-S100 series System Integration Manual [2] for more details):

- The Master Request line (**MRQ**) is a digital output, which is initially driven low.
- The Reset line (**RESET_N**) is a tri-stated digital output, which is initially must float.
- The Slave Ready line (**SRDY**) is a digital input, and is polled from the host common component.
- The Slave Request (**SRQ**) is a digital input that MUST be capable of detecting an asynchronous rising edge event (i.e., an edge triggered interrupt). It is the responsibility of the application host to create this mechanism such that a change in the **SRQ** state from low->high results in run-time execution of the Host Common software component (see section 3.6). If the application is a "sleepy" device, the **SRQ** interrupt must be capable of waking up the host processor (otherwise important asynchronous network events will not function).

☞ The module power pin (**POWER_ON**) is not managed explicitly to a default value within this API. The module's power I/O needs special consideration on the target platform and contains its' own dedicated HAL interface to manage. Once set, the power line does not change unless told to by Host Common – this is especially important on sleepy devices where the module power pin must be statically held while the CPU is in a low-power state.

3.2.2 HOST_CMN_HAL_SpiReset()

The Host Common software component calls this function during SPI arbitration or on unexpected error conditions by the Host Common software component.

It is expected that the API will flush/reset/clear any internal resources required by the hardware 4-wire SPI in advance of the initiation of new data transfer.

3.2.3 HOST_CMN_HAL_DelayMicrosecond()

The Host Common component calls this function to enforce delays between various stages of the SPI message exchanges between the Application Host and module. The function must not return for a minimum of the time specified (i.e., it must block the current context). A greater time is acceptable – however, long pauses may result in data overruns. Performing other background processing or blocking to allow other threads to run is allowed as long as no other calls to `host_cmn` are made while the delay function is blocked.

The rACM software, an RTOS-less design, implements this API using a simple NOP loop which converts the specified delay into a number of loop cycles.

3.2.4 HOST_CMN_HAL_ExchangeMessage()

The Host Common component calls this function to transfer a serial byte stream over the 3-wire SPI. There are two important parameters set by `host_cmn` that need to be correctly used by this HAL API to guarantee a successful full-duplex transfer:

- `HostCmnMsgDesc_ttxMsg` – This is essentially an array of descriptors that point to the location of the OUTGOING **MOSI** data for the current SPI transfer. Each txMsg array element (see the definition of `HostCmnMsgDesc_t` in `host_cmn.h`) consists of the following descriptor pair:
- `uint8_t *buff` – A pointer to a contiguous buffer of Tx byte data
- `uint16_t length` – The number of contiguous bytes pointed to by `buff`.

☞ The HAL exchange message API must initiate a full-duplex byte transfer for each txMsgarray table element – The table will terminate when the buff element is set to `NULL`. The SPI chip select must stay asserted for the entire transfer of all table elements.

- `uint8_trxBufPtr` – A pointer to a contiguous buffer in memory (as set by `host_cmn`) for each Rx byte as it is received from the full-duplex SPI transfer.

The function must not return until all full-duplex byte transfers have been completed for the number of bytes specified in the `txMsg` transfer descriptor. Again, whether or not other background processing can be allowed to run will depend on the run-time environment implemented in the host application – however, as is the case with all HAL or APP interface "hooks," no other call to the `host_cmn` component are permitted while the API is running.

The interactions between the DSPI driver and the API parameters are best illustrated with an example implementation from the `hal/app_specific/host_cmn_hal_k20` in the rACM implementation. In this RTOS-less design, each byte exchange is done using a simple hardware polling loop which monitors the Kinetis SPI hardware register bit to indicate the 8-bit transfer is complete. Alternatively, this functionality could be enhanced with the addition of a DMA transfer between the `txMsg` buffer pointers and the hardware using an interrupt to indicate that the entire data exchange is completed – but again, this is all dependent on the run-time environment and SPI hardware that the host application is managing.

3.2.5 HOST_CMN_HAL_InterfaceEnable()

The Host Common component calls this function to manage to Host->Node SPI interfaces during periods where the module is in low-power mode and there is no pending data to exchange between the two processors. The Boolean parameter used by this HAL API indicates the interface enable state (i.e., 0 – inactive, 1 – active).

When the SPI controller is active, the SPI_CS, SPI_MOSI, and SPI_CLK signals are normally going to be driven high by the DSPI hardware block.

When the SPI controller is inactive, the application must ensure that the same SPI_CS, SPI_MOSI, and SPI_CLK signals are driven low.

It may be necessary to switch the port mux settings for these SPI interface between functional and GPIO mode to achieve these line states – but again, that is entirely based on the hardware interfaces supported on the target platform.

3.2.6 HOST_CMN_HAL_SetMrq()

The Host Common component calls this function in advance of a Host-initiated message exchange over the DSPI interface. The Master Request (**MRQ**) is pulled high to pull the module out of any low-powered state and prepare its own resources for the full-duplex exchange. When the Host has completed the transfer of all serial data, **MRQ** will be de-asserted. The Boolean parameter used by this HAL API indicates the line interface state (i.e., 0 – inactive "low", 1 – active "high").

It is anticipated that this functionality will be achieved by setting the appropriate line state in the host processors' General Purpose Output interface register for the pin allocated for use as the **MRQ** interface.

3.2.7 HOST_CMN_HAL_SetReset()

The Host Common component calls this function when bringing the module out of reset either as a result of a Power-On Reset (POR) and the odd exception event. When held in reset, the **RESET_N** pin must be driver "low". Otherwise the **RESET_N** pin must be tri-stated or allowed to float.

It is anticipated that this functionality will be achieved by setting the appropriate line state in the host processors' General Purpose Output interface register for the pin allocated for use as the reset interface. It may be necessary to reconfigure the GPIO is an input if the tri-state line setting is not supported by the targets' GPIO hardware.

3.2.8 HOST_CMN_HAL_IsSrq()

The Host Common component calls this function when it is necessary to check the line state of the modules' Slave Request (**SRQ**) line to indicate that the module has data and/or indicators that it needs to exchange with the host processor.

The HAL API must simply return the Boolean state of the **SRQ** line interface (0 – clear, 1 – set).

It is anticipated that this functionality will be achieved by reading the appropriate line state in the host processors' General Purpose Input interface register for the pin allocated for use as the **SRQ** interface.

3.2.9 HOST_CMN_HAL_IsSrdy()

This function is called by the Host Common component when it is necessary to check the line state of the modules' Slave Ready (**SRDY**) line to indicate that the module is prepared to exchange full-duplex data over the DSPI interface.

The HAL API must simply return the Boolean state of the **SRDY** line interface (0 – clear, 1 – set).

It is anticipated that this functionality will be achieved by reading the appropriate line state in the host processors' General Purpose Input interface register for the pin allocated for use as the SRDY interface.

3.2.10 HOST_CMN_HAL_SetPowerOn()

The Host Common component calls this function to supply power to the module when the Host Application is ready to initiate the RPMA network join process. When enabled, the host application should ensure that the line interface remains at the correct state unless explicitly told otherwise (i.e., a fail-safe recovery attempt for an unresponsive module). The pin (**POWER_ON**) state is set by the Boolean parameter passed in by the HAL API (i.e., 0 – inactive "low", 1 – active "high").

It is anticipated that this functionality will be achieved by setting the appropriate line state in the host processors' General Purpose Output interface register for the pin allocated for use as the **POWER_ON** interface.

☞ On "sleepy" devices, it is important to note that the module power pin (**POWER_ON**) remains statically set during transitions to and from any low-power modes. If not, inadvertent line state changes may result in module resets – the resultant rescans and network join process will result in shortened battery life and adversely affect RPMA Network interactions.

3.2.11 HOST_CMN_HAL_SecondsSinceStartup()

The Host Common software component uses this function to support error/exception detection on the Host->Node interfaces – these are low-resolution timeout events that are implemented with a free-running time counter that is enabled as the Host Processor comes on of Power-On Reset.

The return value for this HAL API is the current "second" count that has elapsed – determining this count value will depend on the timer resources available on the target host platform. If error detection is not a requirement for the host application (or cannot be provided due to hardware constraints), then the HAL API can simply return a value of zero without any adverse effects to Host->Node interactions.

☞ On "sleepy" devices, it is important to account for time that has elapsed while the processor is operating in low-power mode. Even better, the timer resource used for this purpose is still fully functional during deep-sleep – this is a commonly supported feature on low-powered microcontrollers using a low-power 32K crystal oscillator clock source.

The rACM software design utilizes a Kinetis Second-Counter to support this feature – in addition to providing "counts" in relative seconds since POR, it also operates in low-power modes as its clock source is derived from the low-power 32 kHz crystal oscillator.

3.2.12 HOST_CMN_HAL_MilliSecondCounter()

The Host Common software component uses this function to support a non-blocking milli-second counter to support the RHT protocol. It is used for short delays typically no more than 100's of milliseconds – roll-overs are managed by the `host_cmn` so the count is relative and does not require synchronization with any other event on the Host Application.

The return value for the HAL API is simply the microsecond count value based on any free-running counter resource the host application has available. If RHT support has not been configured, then the HAL API can simply return zero.

The rACM software design utilized a free-running 32k counter to support this feature – 32k "counts" are converted to a millisecond value which is then provided as the return value.

3.3 Host serial interface

The Host Application is responsible for providing a dedicated interface such as RS-232 or an Ethernet port socket for passing host message that conform to the module Host messaging specification (see NANO-S100 series datasheet [1] and NANO-S100 series System Integration Manual [2] for more details). This will allow the endpoint device to leverage existing RPMA commands and PC tools that can be used to configure, provision, automation test and debug the RPMA module. In addition, the application can leverage the Host Common message buffers and stream-to-packet utilities (see section 3.12) so that `host_cmn` can do all of the "heavy-lifting" when it comes to parsing the data received from the Host Serial Interface.

The choice of a Host Serial Interface will obviously be influenced by the target platform and its' available peripherals. A Host serial interface can be implemented using the following peripherals:

- UART Interface (RS-232 straight-through or USB to RS-232 convertor)
- SPI Interface
- USB (using a serial emulation connection)
- I^2C
- Ethernet Socket

Regardless of the choice of interface, the Host application will be responsible for creating the HW drivers for both Tx and Rx data streams. In addition bridge functions must be created between the driver interface and `host_cmn` in the following manner:

- Tx serial data transfers (i.e., outgoing from the end device to PC Host) will need to be invoked by the Controller Endpoint function (see section 3.7.1 – the controller endpoint is registered at run-time initialization)
- Rx serial data (i.e., incoming data from the PC Host) will need to be streamed and converted to a completed, intact RPMA message type and routed into `host_cmn` via the `HOST_CMN_MGR_RouteMsg()` API (see section 3.7.3). If the Stream-to-Packet utility is enabled in `host_cmn` and integrated with the Host Serial Interface receive handlers (see section 3.12), then this will be handled internally and routed by the Host Common process loop.

To support Rx stream processing from Host Serial Interface, the Host Common component provides a data structure and APIs that support a circular queue implementation that can be leveraged by the application serial driver. This is not optional if the `host_cmn` Stream-to-Packet utility is being used – a circular buffer where the Rx byte data is piped can be accessed using a Buffer Descriptor that `host_cmn` can provide via `HOST_CMN_MGR_GetStreamBuffer()`.

For an example of a Host Serial Interface implementation that uses both the `host_cmn` circular buffer method and stream-to-packet utilities, the rACM design is an excellent template to follow. The rACM Host Serial Interface is implemented with a dedicated Kinetis UART peripheral and the source code can be found at `hal/app_specific/racm_uart.c`. Since the stream-to-packet utilities are performing all circular buffer management and invoking the Host Common Message Router, the applications' interactions are very simple – i.e., it transfers bytes from the UART Rx FIFO into the Host Circular Buffer and enables the Host Common processing loop.

The following subsections provide additional details on the `host_cmn` circular buffer APIs that can be used by the host serial interface implementation.

☞ Not all of the APIs in `host_cmn_buff.h` are listed – only the ones that application would typically need to use in its Host Serial Interface implementation.

3.3.1 Host Common buffer descriptor

The Host Common Buffer APIs are all designed to manage a circular buffer in memory using a Buffer Description structure which, in turn, manages a serial byte stream. It is set of APIs that is easily adapted for use by the Host Serial Interface for all Rx Host data – as the section overview mentions, it is mandatory to use the host common buffer descriptor in the Host Interface implementation if the Stream-To-Packet utility is to be properly leveraged.

The Host Common Descriptor structure is defined in `host_cmn_buff.h` as follows:

```
/** Buffer Description Structure */
typedef struct
{
   uint32_t  readIndex;
   uint32_t  writeIndex;
   uint8_t * buff;
   uint32_t  buffSize;
} host_cmn_buff_t;
```

If the Host Serial Interface implementation does not leverage Stream-To-Packet utility, then the application code will need to allocate a region of memory for use as the serial circular buffer. It will then need to declare and initialize a Buffer Descriptor to its location in memory.

If the Host Serial Interface implementation does leverage the Stream-To-Packet utility, then host serial implementation will need to get a pointer to the internal `host_cmn` buffer descriptor by using the `HOST_CMN_MGR_GetStreamBuff()` API (this is the method which the rACM software design uses).

3.3.2 HOST_CMN_BUFF_Write()

Writes data to the designated circular buffer. It performs all of the necessary management of the Host Common Buffer Descriptor that is passed in as one of the parameters:

- `host_cmn_buff_t *buffDesc`-Pointer to a Buffer Descriptor that is managing the circular buffer to be written to.
- `uint8_t *input`– Pointer the data to write to the circular buffer.
- `uint32_t size` – Number of data bytes to be added to the circular buffer.

The API will return a Boolean true if the write succeeded or a false if there is insufficient space left in the circular buffer.

3.3.3 HOST_CMN_BUFF_DataAvailable()

Returns the number of unprocessed data bytes left in a circular queue that the Host Common Buffer Descriptor that is passed in as a parameter is pointing to.

3.3.4 HOST_CMN_BUFF_SpaceAvailable()

Returns the number of available data bytes left in a circular queue that the Host Common Buffer Descriptor that is passed in as a parameter is pointing to.

3.3.5 HOST_CMN_BUFF_PrepareRead()

Gets a reference (pointer and length) to data in the buffer. This data will not be removed from the buffer and is protected from overwrite by arriving write data until a corresponding `CommitRead()` call is made.

3.3.6 HOST_CMN_BUFF_CommitRead()

Releases data (i.e., increment the `readIndex`) from the designated circular buffer. It performs all of the necessary management of the Host Common Buffer Descriptor that is passed in as one of the parameters:

- `host_cmn_buff_t *buffDesc` -Pointer to a Buffer Descriptor that is managing the circular buffer to be read from.
- `uint32_tbytesRead` – Number of data bytes to be released from the circular buffer.

3.4 Application layer interfaces

There are several mandatory application "hooks" into the Host Common software component that also must be implemented prior to building the integrated Host Application. This is essentially the APIs the application will need to handle to successfully interact with the RPMA Network.

The custom host application will be responsible for implementing handlers for the following end device features:

- Host Common run-time application indicators.
- Host Common run-time process request.
- Host Common state change event (for managing persistent host common data).
- RPMA Transmit SDU Status event (i.e., status of endpoint)
- RPMA Receive SDU Message event

The `host_cmn` header file, `deps/src/host_cmn_app.h`, contains all of the function prototypes that the custom host application MUST implement, compile, and link with the rest of the application. Failure to implement any one of the function placeholders will prevent the application from building (i.e., undefined symbol during link time).

For an example implementation of the application layer interfaces, the rACM software consolidates all application "hooks" with `host_cmn` in a single file in `app/host_app_endpt_intf.c`.

The following subsections provide further details on each of these application layer interfaces in the order they are prototyped in `host_cmn_app.h`:

3.4.1 HOST_CMN_APP_ManagerEvent()

This is the primary "hook" for how the Host Common reports run-time events that the application may need to manage. The parameter that the `host_cmn` passes into this API is the enumerated value, `HostCmnAppEvent_t`, which is also defined in `host_app_endpot_intf.c` as follows:

- Pre-UI Notification event
- Node State Change event
- Node ACK handshake event (for host-originated command requests)
- Node Reset event
- Message from controller event

The following subsections summarize the implications of each event to the host application and provide a suggestion on how the application should manage them.

3.4.1.1 HOST_CMN_APP_EV_PRE_UI

The Pre-UI Notification event is the most important of the `host_cmn` manager events and requires careful consideration with regards to how the host application responds to this indicator.

A scheduled Uplink Frame for which the module must transmit, regardless if the host has application data to transmit or not, is required at least once per day to maintain the PHY link with RPMA Access Point. However, it is preferable for the application to enqueue any user data for the update frame to alleviate UL network congestion. This is accomplished with the Pre-UI Indicator that is configured and set up during initial `host_cmn` startup sequence. As long as the module is joined to the network, the Pre-UI Notification Indication message is sent prior to each scheduled update frame using the `host_cmn HOST_CMN_PRE_UI_ADVANCE_MS` configuration setting (see section 3.1).

☞ It is important to account for the real-time processing latency when configuring the Pre-UI Advance setting. If the application is using this event to trigger a sensor read whose results will then get reported in the Uplink Frame, hardware settling may be on the order of 10-100 ms. However, if sensor data is already collected in a separate background process and a snapshot of the data can simply be formatted into a user data packet without significant overhead, then latency may be on the order of 1-2 ms.

In most instances, the application will respond to the Pre-UI Notification with a function handler that will form and send user data to the module to be enqueued and transmitted to the RPMA backend (see section 3.8 for the procedure to enqueue a Tx request with Host Common). However, the follow-up steps to the Pre-UI notification are entirely dictated by the requirements of the custom application.

It is possible that the application is not required to send data on a scheduled interval. For example, a simple remote alarm monitoring unit might only be required to send user data for an asynchronous alarm state change. In this case, the Pre-UI notification can be effectively ignored by the host application (or disabled by setting the Pre-UI advance setting to zero).

The rACM design is heavily influenced by the Pre-UI Notification. Sensor data on the rACM is collected at a faster read-interval and saved as undelivered records in NVM. When the Pre-UI notification is received, the application will parse all undelivered sensor data and form and enqueue a Tx SDU request using these chained records. Its state management also resolves contention issues with enqueued Tx SDU requests that may have resulted due to an asynchronous OTA event – a custom host application can also leverage the rACM application implementation if has the same design considerations to manage (i.e., read intervals, UI intervals, and asynchronous events).

3.4.1.2 HOST_CMN_APP_STATE_CHANGE

The state change event is used by Host Common to alert the application of any changes in the node. This is primarily used for managing the endpoint device on whether or not it actually is joined to the RPMA network or not.

However, as with all of the Manager events, there is no additional data to report that what the previous and new states are (`host_cmn` does keep track of this internally). The application must request the new state from Host Common through use of the `HOST_CMN_MGR_ LastNodeState()` API – the return value for this API uses the following enumerated values (as defined in `host_customer_msg.h`):

- **SYS_MGR_STATE_NIL** – The module System Manager has not started.
- **SYS_MGR_STATE_STARTUP** – The module System Manager has been started.
- **SYS_MGR_STATE_IDLE** – Startup sequence complete. Waiting for network enter command.
- **SYS_MGR_STATE_SCANNING** – Scanning for the RPMA network.
- **SYS_MGR_STATE_TRACK** – Scan successful – trying to join RPMA network.
- **SYS_MGR_STATE_JOINED** – Network joined successfully.

The rACM design uses the state change indicator (and follow-up call to get the module state) for setting a high-level Over-The-Air (OTA) tracking state that is saved in persistent memory. rACM events that result in a OTA data to be sent to the backend, leverages this state for determining if the data can be enqueued in a Tx SDU request or whether the event must be saved in NVM and transmitted at a later time when the module rejoins the RPMA network. For all practical purposes, any module state other than `SYS_MGR_STATE_JOINED` means the OTA link to the backend is down.

3.4.1.3 HOST_CMN_APP_EV_ACK

The Host Common generates the Application Event Acknowledgment when it receives the ACK handshake from the node for any host-originated command.

In most host implementations, this event is probably only of interest to the Tx SDU request. However, the ACK handshake is only an acknowledgement that the module has received the request –it should not be interpreted as an acknowledgement that the Tx SDU has been successfully received by the Access point (this is handled in a separate Application layer interface).

The rACM design does not leverage this indicator in its design. However, this indicator can be used to support payload data management, internal buffer management, and/or Tx SDU state management – all of this will be dictated by the custom host application requirements.

3.4.1.4 HOST_CMN_APP_EV_NODE_RESET

The Host Common component will use this indicator to inform the application that it is resetting the module as a result of an error condition (e.g., Inactivity Timeout, SPI transfer failure).

The host application can use this indicator for any error logging or error tracking feature in its design.

3.4.1.5 HOST_CMN_APP_MSG_FROM_CTRL

The Host Common component will use this indicator to inform the application that it has received a controller message from the host serial interface.

The rACM design uses this event in the implementation of a power savings feature involving the Host Serial Interface. An inactivity timeout mechanism controls whether or not the interface should be powered down – any incoming message resets this timer.

Custom host application can leverage this same type of mechanism or ignore it completely based on design requirements affecting power management of the serial interfaces.

3.4.2 HOST_CMN_APP_ActivateProcessingLoop()

The Host Common component uses this application layer interface to request that the host application unblock and/or do an iterative pass through the Host Common process loop (see section 3.6).

A typical scenario for this type of request would be the reception of an RPMA message packet from the Host Serial Interface that is parsed by the steam-to-packet utility (see section 3.12).

Failure to properly invoke the Host Common execution loop will result in delayed or dropped controller requests and/or module events.

The rACM RTOS-less design uses a run flag in its main processing loop for determining whether or not to perform an iterative pass through Host Common process loop. This API simply sets this run flag. Other host application designs may leverage a blocking semaphore or activate a suspended task – again this all depends on the manner that the Host Common process loop is implemented.

3.4.3 HOST_CMN_APP_StateUpdateNotification()

The Host Common software component will use the API to inform the application that one of its internal state variables has changed. On “sleepy” host applications, these state variables must persist across periods of Deep Sleep. Depending on the target microcontroller and memory resources, this may require additional memory management which Host Common relies on the host application to perform. Even on non-sleepy host applications, this type of memory management may still warrant implementation such that the host common component can resume steady-state processing should an exception reset occur.

This type of memory management is best demonstrated using the rACM design – there are two copies of Host Common tracking variables. Host Common will read/write state management variables that are linked in internal SRAM. The rACM host application also links a second copy of the Host Common persistent data in FlexNVM that is updated/synchronized whenever this API is called by `host_cmn`. Because the rACM is a “sleepy” device, the FlexNVM copy is persistent across periods of deep sleep and is used to re-initialize the Host Common Software component as it exits low-power mode.

Further details on the Host Common’s support for deep sleep is found in section 3.10. The `HOST_CMN_APP_StateUpdateNotification()` application interface is an integral part of that feature.

3.4.4 HOST_CMN_APP_TXSDU_Result()

In most instances, the Tx SDU payload will be broken down into smaller Tx PDUs by the modules’ MAC layer. Depending on the RF link, it may take several frames to transmit a single enqueued Tx SDU request originated by the application. Combined with security and a reliable retry mechanism, it could be up to several minutes before the status of the Tx request has been acknowledged by the RPMA network. Keeping the host application powered on during this time or blocking other threads from execution may not be practical.

Host Common uses this API to inform the application of the resultant status of a previously enqueued Tx SDU request with the following parameters:

- `uint16_t tag` – The Tx SDU identifier, as specified in the Tx SDU request (see section 3.8).
- `host_msg_txdu_result_sdustatus_tsdustatus` – The Tx SDU result/status (as defined in `host_customer_msg.h`).

The custom host application can use the Tx SDU result in many different ways – state tracking management, buffer management, book-keeping of undelivered data versus delivered data, or in the implementation of an application transport layer (e.g., fragmentation, reliable delivery, etc.). As is the case with most of the interfaces discussed in this document, the implementation will depend on the application requirements. For all intents and purposes, any `sdustatus` value other than `HOST_MSG_SDU_STATUS_BITS_ACK_SUCCESS` is considered a failed Tx SDU.

The rACM software design uses a rather simple Tx OTA protocol – only one enqueued Tx SDU request is allowed at a time for which the Tx SDU status indicator is used to determine when a Tx SDU request that is in contention with another active request can finally be forwarded through `host_cmn`. Additionally, the `sdustatus` field is used in the bookkeeping of application sensor data to determine when undelivered data can be staged as delivered data in NVM. Further details on the rACM Tx OTA protocol and sensor data management can be found in the rACM HLD (see reference [3]).

3.4.5 HOST_CMN_APP_RXSDU_Message()

For application data models that have user data originating from the backend down to the endpoint, this application layer interface is used by Host Common to notify the application that it has received Rx user data that can be processed.

The parameters passed in through this API are as follows:

- `uint16_t size`- Number of payload bytes in the Rx SDU Message
- `uint8_t *msgPtr` – Pointer to a contiguous block of memory where the Rx SDU Message starts.
- `uint16_t flags` – Control flags for the Rx SDU message (2.x systems only).

The format and the actions taken by the custom host application are entirely dependent on the application requirements.

☞ When the API function returns, the data pointed to by `msgPtr` may be overwritten. It is important that any Rx payload data access be completed before returning. Alternatively, the payload data can be copied to a different buffer in memory to allow processing in a different, lower-priority context.

The rACM design simply invokes a Rx OTA message API in its main processing loop which parses the payload data by Opcode and perform all necessary follow-up actions (e.g., resets the module following reception of an OTA Node Reset command).

3.5 Run-Time initialization

Run-Time initialization of the Host Common software component will need to be done whenever the target processor comes out of reset and the module is (or has been) powered on to initiate the network join process. This step must precede any other call to the Host Common interfaces.

As an example, the rACM software does its run-time initialization of Host Common as part of its reset recovery event management – specifically, calling `HOST_APP_ENDPT_Init()` in `app/host_app_endpt_intf.c`.

Initialization of the Host Common component involves the use of the following API.

3.5.1 HOST_CMN_MGR_Init()

The Host Common Manager Initialization API has several important parameters that the application must understand and set correctly:

- `HOST_CMN_STATE_PreservedState_t *stateLocation` – a pointer to memory where the persistent/preserved Host Common state data is located. Primarily intended for deep sleep operations where data is preserved in some forms of NVM that will vary based on the target microcontroller (further details on Host Common support for Deep Sleep can be found in section 3.10).
- `boolstateSetDefaults` – Boolean flag to indicate to the Host Common component to initialize all `host_cmn` state variables to system reset/boot defaults. When set to false, `host_cmn` will initialize its internal state variables to the data pointer to by `stateLocation`. This is the method in which Host Common and the application data preserve internal `host_cmn` state variables across low-power modes required by a battery-operated application host.

- `HostCmnLogMsgHandler_tlogMsgHandler` – Function pointer to an application-specific interface used for the transport of a formatted string. Set to `NULL` if using the Host Common Logging capabilities through the primary Host Serial Interface (see section 3.11).
- `uint8_t *cmnMsgAssemblyBuff`– A pointer into memory in which Host Common uses to assemble messages. The size of the buffer must accommodate all messages as defined in `host_cmn_msg.h` which this platform uses.

☞ The buffer needs to be unmodified for the duration of any calls from the application host to `HOST_CMN_MSG_ProcessCmd()` and `HOST_CMN_MSG_SendLogToHost()`

- `HostCmnMsgSinkHandler_tcontrollerSink` – Message sink for messages with the controller endpoint destination (i.e., Tx data for the Host Serial Interface – see section 3.7).
- `HostCmnMsgSinkHandler_tappSpecificSink` – Message sink for messages with the Host-Specific endpoint destination. May be set to `NULL` if the application does not have an extended host-specific command set or has a dedicated serial interface for the equivalent feature (see section 3.7 for more details on the message sink).
- `boolenableRHT`- For Host Common configurations that support the Reliable Host Transfer Protocol, this Boolean sets the default RHT state (i.e., 0 – inactive, 1 – active).

The rACM design can be used as an example of using the Host Common Initialization API – the preserved data, message buffer, and message sinks are all declared within the same module (i.e., `host_app_endpt_intf.c`). However, since the rACM design leverages the Host Common Logging utilities (in packed mode), the `logMsgHandler` function point parameter is set to `NULL`.

3.6 Run-Time execution

The internal Host Common design depends on a single threaded execution of `host_cmn` code. All calls to the `host_cmn` interfaces must return before another interface can be called. For instance, in a RTOS-less foreground/ISR design, calls to `host_cmn` interfaces should not occur in ISR context. In an RTOS based system, `host_cmn` functionality should be grouped into a single thread or, alternatively gated by a mutex of some sort to control its access.

The rACM RTOS-less software design invokes the Host Common run-time loop from its main processing loop (`MPL_ProcessingLoop()` within `app/host_app_proc_loop.c`). A global flag variable, `MPL_nodeIntfEventFlags`, is used by the reference application to trigger an iterative pass through `host_cmn` execution loop.

☞ Because the rACM is implemented with a single foreground thread, other processes (e.g., UART stream-to-packet management, the application state engine), are all given a chance to execute in a simple round-robin style implementation. An application with an RTOS pre-emptive scheduler can more easily accommodate the same level of functionality in a more deterministic manner. Critical real-time events on the rACM do exist (e.g., a time-synchronization feature) and are implemented in interrupt context.

The run-time execution loop used by the Host Common component involves the use of the following API:

3.6.1 HOST_CMN_MGR_Process()

The Host Common Manager Process interface must be called by the application to allow `host_cmn` processing for the following conditions:

- At startup, when the `HOST_CMN_MGR_SetNodePower()` API is called.
- Whenever the module **SRQ** line state is high.
- Whenever the HAL function `HOST_CMN_HAL_ActivateProcessLoop()` is invoked (see section 3.4.2).
- Any time return from `Host_CMN_MGR_Process()` indicates a non-idle state.
- If the Host Common Stream-to-Packet utilities are enabled to support the Host Serial Interface, any time byte data is received over the Host Serial Interface (see sections3.3 and 3.12).
- At some background rate, typically at the rate that the application performs other background functions (sensor reads, etc.). This background call rate is not at all critical, and hours between calls is fine. It should be

tied to the expectations for module hard failure recovery (see `HOST_CMN_NODE_INACTIVITY_TIMEOUT_SECS` configuration parameter for timeout description).

Calling `HOST_CMN_MGR_Process()` more often is fine, the only downside is power consumption due to Host CPU activity.

3.7 Host Common communication framework

Integrating with the Host Common Communication Framework (see section 2.6 for an overview) requires the application to implement a few "message sink" functions that are required for `host_cmn` communication endpoints to successfully integrate with the intended target destination. These are implemented as message sink functions that are registered with Host Common as part of the run-time initialization process (see section 3.5.1). There are two destination message sinks that the application is responsible for:

- Controller endpoint message sink (i.e., outgoing data to an external PC host through the Host serial interface).
- Host-Specific endpoint message sink (i.e., application specific messages that are routed as an extended component of the RPMA host command instruction set).

Messages that are originated from the application are passed into the Host Common component using the `HOST_CMN_MGR_ RouteMsg()` for destination determination and forwarding – the internal `host_cmn` message handlers utilizes this same API for messages that come from the upper control interface, from the module, as well as `host_cmn` internal stream-to-packet processing.

The following subsections provide more details on each of these communication framework elements.

3.7.1 Controller Endpoint Message Sink

The Controller Endpoint Message Sink is the bridge between the `host_cmn` communication framework and the external PC host. It is the applications' responsibility to implement this handler so that the `host_cmn` message data is transferred, via the appropriate hardware driver interface, as outgoing data on the Host Serial Interface. The controller endpoint message sink uses the `HostCmnMsgSinkHandler_t` function prototype and passes in a `HostCmnMsgDesc_t` descriptor array as its parameter –i.e. the same descriptor array parameter used by the SPI HAL Exchange Message API detailed in section 3.2.4 and consists of the following descriptor pair:

- `uint8_t *buff` – A pointer to a contiguous buffer of controller message data
- `uint16_t length` – The number of contiguous bytes pointed to by `buff`.

☞ The table will terminate when the `buff` element is set to `NULL`.

As with all `host_cmn` interfaces, the message data that the descriptor data points to must have completed its transfer to the driver function and/or hardware before returning. The exact method of transferring the data will be dependent on the microprocessor hardware resources and run-time environment.

As an example implementation of the Controller Endpoint Message Sink, the rACM software implements this bridge function in the `hostControllerSink ()` within `host_app_endpt_intf.c` (it is one of the initialization parameters used in the rACMs' call to `HOST_CMN_MGR_Init()`). The rACM controller message sink function transfers all of the message bytes from the message descriptor through a Tx UART driver function. After all bytes have been transferred, the function returns.

Alternative implementations might involve the use of DMA transfers and a blocking semaphore to allow other processes a chance to run while the data is transferred, or a circular buffer of data waiting to be transmitted with an interrupt transferring data to a hardware queue.

3.7.2 Host-Specific Endpoint Message Sink

The Host-Specific Endpoint Message Sink is the bridge between the `host_cmn` communication framework and the application itself – normally, this will not involve a hardware serial interface so its implementation will be quite straightforward. As with the Controller Endpoint Message Sink, the function prototype, input parameters, and method of initialization with `host_cmn` are identical.

It is expected that `host_cmn` will have a complete, formatted message ready for processing when this message sink is called – all that is necessary is to parse the RPMA message header for the appropriate Opcode and call the appropriate application handler.

The same context of execution rules apply to the host-specific message sink as all of the `host_cmn` invoked interfaces – i.e., the Host Common component cannot be called from a different context until the API has returned and `host_cmn` has finished its current execution process.

As an example implementation of the Host-Specific Endpoint Message Sink, the rACM software implements this bridge function in the `hostSpecificSink ()` within `host_app_endpt_intf.c` (it is also one of the initialization parameters used in the rACMs' call to `HOST_CMN_MGR_Init()`). The rACM Host-specific sink handler processes the received endpoint data in a command switch-case statement. All of the host-specific Opcodes and message formats that the rACM uses for application specific commands are defined in a `host_specific/host_app_msg.h` header file – these are primarily messages used to configure and provision the ACM host for field deployment via a the UART-based Host Serial Interface. The rACM also includes host-specific Python scripts for processing these commands on the external PC host controller endpoint. A custom host application can also use this as a template in the creation and integration of its own set of command/response handlers using the RPMA message format.

3.7.3 HOST_CMN_MGR_RouteMsg()

Routes all application originated messages to the Host Common communication framework – the destination commonly being the PC host controller or the module. The following parameters need to be provided:

- `HostCmnMsgDesc_tmsg` – The Host Common message descriptor – consisting of an array of message descriptors whereas each array element (see the definition of `HostCmnMsgDesc_t` in `host_cmn.h`) consists of the following descriptor pair:
    - `uint8_t *buff` – A pointer to a contiguous buffer of message byte data
    - `uint16_t length` – The number of contiguous bytes pointed to by `buff`.

☞ The application needs to ensure that the last array element sets the `buff` field to `NULL` to properly terminate the message descriptor element.

- `HostCmnRoutingPoints_t source`– The originating communication endpoint – typically, all messages originating from the application will identify itself as the HOST_SPECIFIC endpoint.

The `host_app_endpt_intf.c` file within the rACM source code can be used as examples of host-specific messages originating from the application – nearly all of the rACM host-specific messages use a command-response format.

☞ Per the run-time execution requirements of the Host Common execution loop (see section 3.6), any call to `HOST_CMN_MGR_RouteMsg()` must also unblock or initiate an iterative pass through `host_cmn` by calling `HOST_CMN_MGR_Process()`.

3.8 Host Tx SDU enqueue requests

All hosts will be required to generate application/user data (referred as a Service Data Unit or SDU) that will need to be transmitted over the RPMA network to the backend for subsequent processing by the end-user. This is an expected follow-up action for the following application events:

- Pre-UI Notification event generated in advance of the scheduled uplink frame.
- An asynchronous alarm event generation by the application.
- In response to an Rx OTA user-command received from the backend.

The high-level design requirements will dictate the format and length of the OTA payload data. However, smaller/efficient payload packets will result in shorter transmit epochs – this translates directly to longer battery life and increased network capacity.

The method in which the Host Application enqueues a Tx SDU request to the module for eventual transmission to the network is via a Host->Module Tx SDU Request message. Host Common provides an API to facilitate this request from the application.

The Tx SDU request must be a minimum of 8-bytes and a maximum of 464 bytes – on 1.x systems, the Tx SDU payload must be a multiple of 8 bytes in length (the configuration parameter, `HOST_CMN_FORCE_1_X_SDU_ALIGNMENT`, will perform this necessary byte padding in the Tx SDU request to conform to this restriction).

In general, the Module MAC layer only supports a single Tx SDU transfer at a time – the application design should take this into consideration in its design using the request API and the Tx SDU response event (see section 3.4.4) to control the flow of transmission packets forwarded to the module.

The rACM software uses an OTA state to help enforce this application restriction, i.e., only a single Tx SDU at a time is in progress at any given time on the endpoint design. A Tx SDU request is considered active from a successful Tx SDU enqueue request with `host_cmn` until the Tx SDU Status event is received (section 3.4.4) or the module is reset (section3.4.1.4) – both are events reported to and handled by the `host_cmn` application layer interfaces. All OTA message formats, Tx and Rx, are defined for the rACM application within `host_specific/host_app_msg.h`.

The Host Common API used for enqueing Tx SDU requests to the module is as follows.

3.8.1 HOST_CMN_MGR_QueueTxSdu()

It is the Host Common API used by the application to send a Tx Service Data Unit (SDU) to the module to be transmitted Over-the-Air to the RPMA backend. The payload buffer used by the application remains unchanged until one of the following:

- A Boolean False returned from this function (e.g., `host_cmn` cannot enqueue the request due typically because the module is not joined or the module already has SDUs in progress, exhausting its internal buffer space).
- A `HOST_CMN_MGR_EV_ACK` event is received (see section 3.4.1.3)
- A `HOST_CMN_MGR_EV_NODE_RESET` event is received (see section 3.4.1.4).

Overwriting the payload buffer before these events occur will result in a corruption of the payload data as it is transferred to the module over the `host_cmn` node endpoint transfer functions.

The parameters used by `HOST_CMN_MGR_QueueTxSDU()` are as follows:

- `uint16_t size` –The payload message length, in bytes

  On 1.x systems where `HOST_CMN_FORCE_1_X_SDU_ALIGNMENT` is NOT set in the Host Common configuration header, then the size must be an 8-byte multiple or the Tx SDU will be rejected.

- `uint16_t host_tag` – A SDU identifier assigned by the application.

  The Tx SDU Status event will use the `host_tag` in its result response.

- `uint16_t flags`- Control flags for the Tx message as defined by `host_msg_sduFlags_t`.

  Unless otherwise specified by the Ingenu project engineering team during OTA payload specifications, the flags parameters should always be set to use the `HOST_MSG_SDU_FLAGS_ACKED` bit field.

- `uint8_t *msgPtr` – Pointer to the message payload buffer – the application will be responsible for defining and maintaining the buffer in memory for use by the `host_cmn` Tx SDU handlers.

Examples of the Tx OTA message generation can be found in the rACM design for the various transmit events in the `app/host_app_proc_loop.h`. In addition to generating the call to `host_cmn`, the rACM uses the `HOST_CMN_SPID_MasterMsgIsComplete()` API to ensure that the Host Common component is not in the process of sending a master-originated SDU payload as an additional failsafe mechanism. This method of detecting a contention issue with the `host_cmn` node endpoint controller can also be used in a custom host application design.

3.9 Image Manager interfaces

The Host Common Image Manager is an optional Host Common feature that is conditionally compiled based on the `HOST_CMN_INCLUDE_SW_UPGRADE` defined in the `host_cmn` configuration file (see section 3.1). In addition, the `host_cmn` configuration file contains configuration defines for an image manager "magic number" as well as memory bound definitions for firmware CRC verification.

The custom application design may choose to perform firmware upgrades through its own application-specific methods – However, it is highly recommended that the design leverages the image broadcast features supported by the RPMA Network for Over-The-Air (OTA) firmware upgrades. The Image Manager feature also leverages the existing RPMA Python tools for image upgrades through the Host Serial Interface. A reason for compiling out this feature is for very resource constrained devices which have no scratch area for image retention.

The module and `host_cmn` components need to manage the RPMA OTA firmware download mechanism. The rACM design does leverage this feature as part of its firmware management feature – the rACM HLD (see reference [3]) can be used as an integration example of this feature (including internal Flash memory layout to support a firmware scratch area).

Before discussing the software interfaces that bridge the gap between the Host Common Image Manager component and the application space, this section also reviews important implementation components that make up RPMA OTA image transfer mechanism which include:

- Detailed description of the Over-The-Air (OTA) image file – in particular, the 256-byte broadcast header.
- Further discussion of the default Host Common Image Manager behavior as it relates to the processing the OTA 256-byte broadcast header.
- A discussion on how the user-space Image Manager callback functions can be used to manage multiple OTA firmware images on the same host target.

3.9.1 Image manager OTA file format

To leverage the RPMA Over-The-Air (OTA) code download feature, all OTA binary images must conform to the following format when provided to the network Element Management System (EMS):

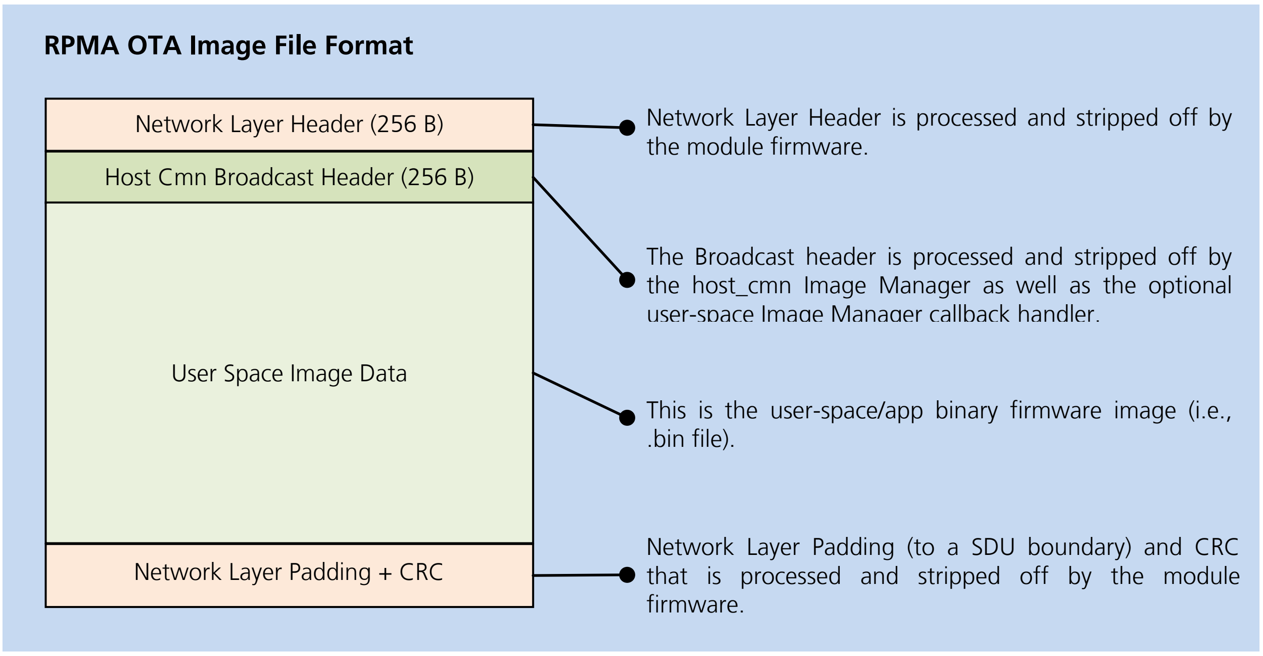
RPMA OTA Image File Format
Network Layer Header (256 B)
Host Cmn Broadcast Header (256 B)
User Space Image Data
Network Layer Padding + CRC
Network Layer Header is processed and stripped off by the module firmware.
The Broadcast header is processed and stripped off by the host_cmn Image Manager as well as the optional user-space Image Manager callback handler.
This is the user-space/app binary firmware image (i.e., .bin file).
Network Layer Padding (to a SDU boundary) and CRC that is processed and stripped off by the module firmware.

Figure 5: On-Ramp Wireless OTA Image File Format

The Network Layer Header, Padding, and CRC are of least interest in the design and integration of the Host Common Image Manager support "hooks". This data is processed exclusively by the RPMA Module firmware – only the Broadcast Header and user image data will get down to the application over the module -> Host DSPI.

The purpose the network layer padding is to make sure the last CDLD image data packet is rounded up to the nearest 256 byte "chunk."

The Host Common Broadcast Header ensures that the application can correctly inform the RPMA Module/Network that the OTA CDLD image is required by the user-space application. As such, the Broadcast Header is organized as follows (multi-byte fields are assumed to use little-endian/network byte ordering):

- Bytes[3:0] – **32-bit Firmware Application ID:** This is intended to match the `HOST_CMN_APP_ID` field defined in `host_cmn_conf.h` for the firmware executing on the endpoint (see NANO-S100 series System Integration Manual [2] for more details).
- Bytes[5:4] – **16-bit OTA Header format:** Host Common and the RACM build tools are hard-coded to use a value of "2" for this field. There are no foreseeable plans to change the header format at this time.
- Bytes[7:6] – **crc16-ccitt**: Calculated over the user-space Image Data (not including this broadcast header). Used during final verification of the image data in the CDLD scratch space before allowing a final image cutover.
- Bytes[11:8] – **32-bit Image Length**: In bytes and calculated over the user-space Image Data (not including this broadcast header).
- Bytes[12] – **Version Check Method enum**: The method in which Host Common will determine whether or not to accept an OTA Broadcast header (unless overridden by an Image Manager user-space callback). They are defined as follows:
  - **`HOST_CMN_IM_CHECK_NONE`** (0): Regardless of the version of the OTA image Data or the image currently executing on target, the `host_cmn` Image Manager will accept the image.
  - **`HOST_CMN_IM_CHECK_UPGRADE_ONLY`** (1): The version of the OTA Image Data must be greater than the image currently executing on target. This is the default image check used by the RACM OTA image generation script.
  - **`HOST_CMN_IM_CHECK_UPGRADE_DOWNGRADE`** (2): The version of the OTA Image Data must not be the same as the image currently executing on target.
  - **`HOST_CMN_IM_CHECK_RANGE`** (3): Upgrade only if the version of the image currently executing on target is within a min/max version range presented in the min/max current version header fields (see below).
- Bytes[15:13] – **OTA Image Version:** One byte for Major, Minor, and Point revision numbers used to identify the OTA Image Data for the various upgrade version check methods (see above).
- Bytes[18:16] – **Minimum Current Version**: Same format as the OTA Image version – the field is only applicable when using the `HOST_CMN_IM_CHECK_RANGE` version check method (see above).
- Bytes[21:19] – **Maximum Current Version:** Same format as the OTA Image version – the field is only applicable when using the `HOST_CMN_IM_CHECK_RANGE` version check method (see above).
- Bytes[23:22] - reserved
- Bytes[255:24] – Reserved for application-specific header data and padding bytes such that the broadcast header is rounded up to a 256-byte image "chunk". The end-user is free to add any additional information in these bytes to further characterize/validate/process an incoming image from the network (e.g., app-specific image type for multi-image support on the same host processor).

This header format is also declared in the `host_cmn` C-source code within `host_cmn_image_man.h` as the `HostCmnIMOtaImageHdr_t` packed struct. In python, the same format is declared in the `host_cmn_gen_ota_image.py` script.

The RACM source code contains a python script in its /build directory that is used to take the binary output produced by the build process (for either a IAR or GNU GCC build process) – which is named `gen_ota_bcast_image.py`. It pulls in two additional python imports to help produce the OTA file image:

- `deps/host_cmn/python/host_cmn_gen_ota_image.py` – Used to generate and prepend the 256-Byte Host Common Broadcast Header (including the calculation of the user-space Image CRC).
- `deps/node_python/gen_ota_image.py` – Used to generate both the 256-byte network header, padding, and network layer CRC.

☞ The `gen_ota_bcast_image.py` is currently hard-coded to use the `RACM APP_ID` (0x18) and `HOST_CMN_IM_CHECK_UPGRADE_ONLY` version check method. At the very least, the `APP_ID` field should be modified to match the user-defined application ID declared by the **HOST_CMN_IM_MAGIC_NUM** macro (see section 3.1).

☞ The `deps/host_cmn/python/host_cmn_gen_ota_image.py` will need to be modified to add any fields that are application specific at byte offsets [255:24].

3.9.2 Default HOST_CMN Broadcast Header Processing

As discussed in the previous section, the Host Common Image Manager will accept a broadcasted network image under the following conditions without any intervention from the user-space application firmware:

- The Application ID in the Host Common Broadcast Header matches the `HOST_CMN_APP_ID` field defined in `host_cmn_conf.h` for the firmware executing on the endpoint (see NANO-S100 series System Integration Manual [2] for more details).
- The version check method defined in the Host Common Broadcast Header as follows:
  - **`HOST_CMN_IM_CHECK_NONE`** (0): Regardless of the version of the OTA image Data or the image currently executing on target, the `host_cmn` Image Manager will accept the image.
  - **`HOST_CMN_IM_CHECK_UPGRADE_ONLY`** (1): The version of the OTA Image Data must be greater than the image currently executing on target. This is the default image check used by the RACM OTA image generation script.
  - **`HOST_CMN_IM_CHECK_UPGRADE_DOWNGRADE`** (2): The version of the OTA Image Data must not be the same as the image currently executing on target.
  - **`HOST_CMN_IM_CHECK_RANGE`** (3): Upgrade only if the version of the image currently executing on target is within a min/max version range presented in the min/max current version header fields (see below).

The Broadcast Header processing may occur multiple times during the typical RPMA OTA Broadcast Image cycle – in particular upon reception of the initial broadcast start indication as well as just prior to receiving the completed image. The broadcast header will also undergo additional verification checks should the module, for any reason, rejoins the RPMA network.

3.9.3 Supplemental Application Broadcast Header Processing

The user-space application has the ability to add additional processing/validation of an OTA Image broadcast header (see previous section) with its own application-specific defined function handler that is registered with the Host Common Image Manager. This method is required if application-specific fields are added to the broadcast header and needs to be taken into consideration or relayed to the user-space application.

User-space Broadcast Header validation/verification is using an additional callback registration by using the `HOST_CMN_IM_Init()` at startup which is used to pass a function pointer to an application-defined secondary header check function – ideally, this API should be used directly after a call to `HOST_CMN_MAN_Init()` (see section 3.5.1).

The format of the callback function that is registered through use of the aforementioned initialization function is specified by the `HostCmnIMOtaHdrCheck_t` definition in `host_cmn_image_man.h` as follows:

```
/*! Callback function type for app specific OTA header management.
 *
 * \paramcmnHdrPtr - pointer to common header
* \paramappSpecificHdrPtr - pointer to start of any app specific header.
* \return bool - true to download image, false to skip.
 */
typedef bool (*HostCmnIMOtaHdrCheck_t)(HostCmnIMOtaImageHdr_t *cmnHdrPtr,
                                       uint8_t *appSpecificHdrPtr);
```

To summarize, the registered user-space function callback will need to process two parameters:

- `cmdHdrPtr`– A pointer in memory where a local copy of the Host Common broadcast header will be found (as described in a previous section).
- `appSpecificHdrPtr` – A pointer in memory where the start of any application specific data in the broadcast header can be found. Specifically, this is a pointer to byte offset[24].

The return value is a Boolean indicator indicating the following:

- FALSE – The image is rejected by the host application and the current active CDLD network broadcast cycle can be ignored.
- TRUE – The image is accepted by the host application – all further network interactions with the current active CDLD network broadcast cycle should be managed by the RPMA module firmware.

As described in the previous section, the Network broadcast header will be processed multiple times during the course of a network Broadcast Image cycle before the actual Module->Host transfer of image data will start (see section 2.10 of reference [4]). The application must be consistent with its processing of this callback.

☞ Once the current OTA image is transferred down to the host application and the host/node reset as a result of a firmware switchover, tracking variables (i.e., the firmware version) should prevent the same image from being re-downloaded (else it might get trapped in a continuous cycled of Module ->Host image transfers and cutover).

3.9.4 Image Manager Call Flow Sequence

The call-flow sequence presented in this section shows all software interactions between the Host Common library and the user-space application and can be used to complement the OTA network Code Download sequence.

It should be mentioned that the call-flow sequence presumes the use of an application callback handler in lieu of the default Host Common Broadcast Header verification – as noted in previous sections, verification checks of the broadcast header are done multiple times before the actual transfer of the image data is started by the Host Common Image Manager.

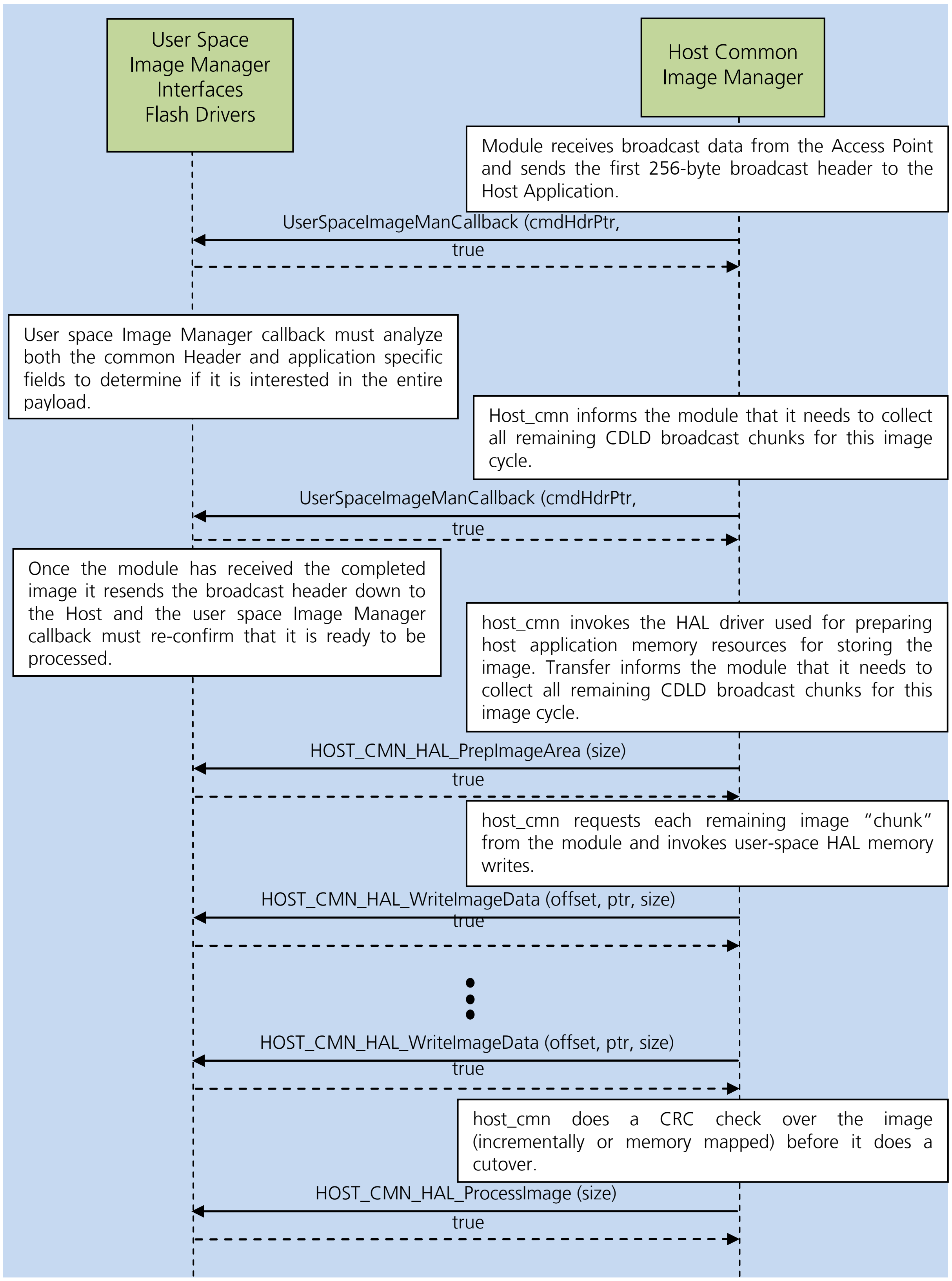
User Space Image Manager Interfaces Flash Drivers
Host Common Image Manager
Module receives broadcast data from the Access Point and sends the first 256-byte broadcast header to the Host Application.
UserSpaceImageManCallback (cmdHdrPtr,
true
User space Image Manager callback must analyze both the common Header and application specific fields to determine if it is interested in the entire payload.
Host_cmn informs the module that it needs to collect all remaining CDLD broadcast chunks for this image cycle.
UserSpaceImageManCallback (cmdHdrPtr,
true
Once the module has received the completed image it resends the broadcast header down to the Host and the user space Image Manager callback must re-confirm that it is ready to be processed.
host_cmn invokes the HAL driver used for preparing host application memory resources for storing the image. Transfer informs the module that it needs to collect all remaining CDLD broadcast chunks for this image cycle.
HOST_CMN_HAL_PrepImageArea (size)
true
host_cmn requests each remaining image "chunk" from the module and invokes user-space HAL memory writes.
HOST_CMN_HAL_WriteImageData (offset, ptr, size)
true
HOST_CMN_HAL_WriteImageData (offset, ptr, size)
true
host_cmn does a CRC check over the image (incrementally or memory mapped) before it does a cutover.
HOST_CMN_HAL_ProcessImage (size)
true

Figure 6: Image Manager Call Flow Sequence

3.9.5 Image Manager HAL Interfaces

If the feature is enabled, in addition to updating the image manager host configuration defines, the following HAL interfaces must also be implemented (see section 3.2 for an overview of the mandatory `host_cmn` HAL interfaces).

3.9.5.1 HOST_CMN_HAL_PrepImageArea()

This is the HAL interface invoked by the `host_cmn` Image Manager when an incoming firmware image has been validated and gives a chance for the application to prepare the image assembly area for updates. The `host_cmn` component will pass in a single parameter:

- `uint32_tlen`– The total length, in bytes, of the image to upload into memory.

If the reassembly area in memory is too small for the completed image, then the application must return false to abort the image transfer. It is the application responsibility to ensure that enough memory is available for the binary image.

For robustness, the uploaded image area is presumed to be located somewhere in Non-Volatile Memory (NVM). For Flash NVM, this API provides an opportunity for the application to erase/clear the target sectors.

Alternatively, the reassembly area may be located in RAM – in this case, no further preparation steps will be required and the HAL API may simply return without additional steps.

As an example, the rACM design uses the second half of Program Flash as an image reassembly area – its' implementation of this HAL API ensured that all the sectors located in this region are erased by invoking the appropriate Flash drivers as a preliminary step in the firmware upgrade process.

3.9.5.2 HOST_CMN_HAL_WriteImageData()

The `host_cmn` Image Manager receives the Image Data from either the Host Serial Interface or the module in "chunks" – as each chunk is received, it invokes this API so that the application can perform the appropriate steps to write the data into the allocated Image Manager assembly area. This HAL API uses the following parameters:

- `uint32_t offset` – The offset, in bytes, from the image base to start writing the current "chunk" of image data.
- `uint8_t buff` – The pointer to a location in memory where the image manager data needs to be copied from.
- `uint16_tlen` – The length, in bytes, of the image data to write.

The actions taken by this API should be fairly straightforward. Depending on the type of memory allocated by the host application for use as the Image Manager assembly area, this may consist of a simple copy or a call to a flash programming sequence driver.

The rACM driver implements this HAL API with a call to its Kinetis-based internal flash drivers.

3.9.5.3 HOST_CMN_HAL_ProcessImage()

The `host_cmn` Image Manager calls this API when it has completed the transfer of all image data to the assembly area and has validated the data with a CRC check. The application may choose to do its own image validation checks at this point.

The details on how the application switches from the current firmware to uploaded firmware image is beyond the scope of this document.

As an example, the rACM design uses this API to signal an event to the main processing loop to perform all of the necessary steps to terminate steady-state processing. After this is done, it branches to a special function running from internal RAM where it copies the uploaded image to the Execute-in Place region of program flash. When the copy is completed (and validated), the rACM performs a software-initiated host reset to start running the new firmware.

3.9.6 Managing Multiple Images for same Application ID

On the rACM, the Host Image Manager and HAL supporting drivers are coded to support a single firmware image that uses a scratch area in internal flash on the Kinetis MCU and then swapped to the execute-in-place (XIP) bank of internal flash.

Supporting multiple images on the same host application that leverage the RPMA Host Common library can be fairly straightforward through use of the Host Common Header Check override function (see previous section) with a supporting firmware tracking variable in the Image Manager HAL drivers. The following simple code examples help illustrate how such an approach would work:

```
/* Static tracking variable for keeping track of application image type */
#define APP_IMAGE_TYPE_HOST 0            /* Image stored in internal flash on host */
#define APP_IMAGE_TYPE_SENSOR 1          /* Image forwarded to sensor firmware over application
UART */
#define APP_IMAGE_TYPE_UNKNOWN 2 /* Uninitialized */
static uint8 appImageType = APP_IMAGE_TYPE_UNKNOWN;

/*! User-space callback function to process/validate the CDLD broadcast header */
booluserSpaceOtaBroadcastHeaderVerify(OtaImageHdr_t *cmnHdrPtr, uint8_t *appSpecificHdrPtr)
{
   /* Is CDLD image intended for host application and greater than the version currently
running? */
if (cmdHdrPtr->cdldMagicNumber != HOST_CMN_APP_ID)
return (false); /* Image is for another application on the ORW network – do not upgrade */
if (imageUpgradeOnlyCheck(cmdHdrPtr->imageVersion) == false)
return (false); /* Image is current – do not upgrade */

   /* Update Application image type tracking variable stored in the first byte of the app
specific portion of the header */
appImageType = *appSpecificHdrPtr;
return (true); /* Allow image upgrade */
}

/*! CDLD Prep Image Area HAL (see NANO-S100 series Data Sheet [1]) */
boolHOST_CMN_HAL_PrepImageArea(uint32_t length)
{
bool status;
   /* Switch on example application image type */
switch(appImageType)
   {
case APP_IMAGE_TYPE_HOST:
         /* Init/erase and example flash part for image scratch area (example only) */
status = eraseFlashSections(CDLD_SCRATCH_AREA_START, length);
break;
case APP_IMAGE_TYPE_SENSOR:
         /* No image prep needed for external sensor (example only) */
status = false;
break;
case APP_IMAGE_TYPE_UNKNOWN:
default:
         /* This is reached in error (example only) */
status = false;
break
   }
return (status);
}
```

The same switch logic can be expanded to the other two HAL APIs required to support the Host Common Image Manager HAL APIs: `HOST_CMN_HAL_WriteImageData()` and `HOST_CMN_HAL_ProcessImage()`.

3.9.7 Host Common Image Serial Upgrade

In addition to OTA code downloads using the RPMA network, Host Common also provides support for host firmware upgrades through the Host Interface serial port which re-uses a subset of the Image Manager Interfaces. In fact, the image upgrade through the serial interface should be used to unit test the User-Space Image Manager HAL Interfaces – this allows the end-user to more efficiently test the image upgrade process without involvement of the RPMA head-end resources.

To perform an upgrade over the Host Serial Interface, the `host_cmn_sw_upgrade.py` python script should be used – the argument to this script should be the generated raw user-space binary image (without the network and `host_cmn` broadcast header).

There are two important distinctions about the image serial upgrade over the Host Interface:

- The `host_cmn` broadcast header is not checked as part of this serial upgrade – any registered Image Manager callback will not be invoked as a result.
- The serial upgrade is unconditional – i.e., there are no restrictions and/or qualifiers in validating whether or not the host will accept the image with the one exception that the image size cannot exceed the allocated scratch space.

3.10 Deep Sleep Persistent Memory

Battery powered devices should enter sleep states in which SRAM contents (or, in some cases, all but a small portion of SRAM contents) are not retained. An exit from these sleep states behaves much like a power-on or other processor reset event, with the caveat that a small amount of persistent state is retained (common methods are in a voltage islanded SRAM portion, a flash section, an external SPI or other memory device, etc.). This type of power save state is referred to in `host_cmn` as “deep sleep.”

The `host_cmn` component supports this type of operation. The application must have some algorithm for determining whether a "boot" event is a valid wakeup from deep sleep or whether it is a hard start of some sort, in which case there is no valid stored state. This must be communicated to the `host_cmn` component via the “stateSetDefaults” flag to `HOST_CMN_MGR_Init()` (see section 3.4) (or the "setDefaults" parameter to `HOST_CMN_STATE_Init()` if the management module is not in use).

The state which `host_cmn` expects to be preserved across deep sleeps is encapsulated in `host_cmn_state.h/.c`. At initialization time, the application provides a pointer to a `HOST_CMN_STATE_PreservedState_t` structure – if there is random access memory directly accessible which will not lose state, no further action is necessary. If not, the application must save the state from the shadow copy passed to the init function to persistent store before entering deep sleep, and restore the previously saved state (if any) before calling `HOST_CMN_STATE_Init()`.

The application can use either of two methods to trigger storing the state persistently. If the application design requires an update to persistent memory every time a change is made, then the function `HOST_CMN_APP_StateUpdateNotification()` (see section 3.4.3) will need to commit any updated data to NVM at this time. Alternatively, the state can be stored just before entering a state where SRAM retention will be lost.

Some routines in the `host_cmn` code do not tolerate loss of SRAM state. If `HOST_CMN_MGR_Process()` returns an idle status, then deep sleep is allowed - otherwise SRAM state must be retained.

Deep sleep should also be prevented as long as there is pending/partial message data received from the Host Serial Interface. If the Host Serial Interface implementation on the application is leveraging the `host_cmn` buffer utilities (see section 3.3), SRAM state should also be retained (assuming, of course, Rx stream data is assembled in SRAM buffers).

A high **SRQ** signal from the module must always exit the deep sleep state and begin calls to `HOST_CMN_MGR_Process()`. It is up to the application to decide when the upper control interface (i.e. UART to PC) is idle and can be unresponsive due to deep sleep.

The rACM software design is a "sleepy" device and is integrated with `host_cmn` to support deep sleep functionality. The Reset Management, Application Endpoint Interfaces, and Main Processing Loops are all implemented per the guidelines listed in this section. It is recommended that the rACM be used as a template for the implementation of a custom host application that also needs to support any deep sleep requirement.

3.11 Host common logging utilities

One of these methods is used to log over a serial, IP, or other interface on the Host Application:

- Application-specific logging method over a dedicated interface.
- Leveraging the existing logging framework in Host Common which utilizes the primary Host Interface.

For target platforms that are restricted to using a single interface, the application design will have no choice but to leverage the Host Common logging utilities. Also, mixing non-standard `printf` outputs over the same interface that is being used to convey RPMA formatted messages will interfere with the Python transport layers on the PC host. Ultimately, the application requirements and available communications peripherals will drive the logging implementation for the endpoint device.

However, if the Host Common Logging Utilities are leveraged – they are enabled and configured at build time using the Host Common configuration header (see section 3.1). One of logging methods supported by `host_cmn` should be selected:

- **HOST_CMN_LOG_IMPLEMENTATION_STDLIB –** Host Logging implementation that assumes string format via the application tools standard library (e.g., `vfprintf()`, `spnprintf()`, etc.). The code that generates these types of log strings can be large due to the ASCII character strings in code-space as well as the overhead by including the aforementioned standard library code.
- **HOST_CMN_LOG_IMPLEMENTATION_RAW** –Host logging implementation that assumes no formatting – i.e., the format string is printed without interpretation. Code size can still be affected by the code-space used to store ASCII character strings for each log statement. Further, any arguments or variable data must be formatted manually by the caller.
- **HOST_CMN_LOG_IMPLEMENTATION_FIXED** – Host logging implementation that assumes 3 32-bit parameters per log statement. This method is primarily used for Host Applications that previously used the RPMA UNIL library logging strings. This method stores ASCII filenames (though not string debug statements) in the code space, which can still negatively impact code size. This method can be combined with the host Python tools for PACKED below to translate filename, line number, and 3 fixed parameters into a formatted string. If this is not done, only filename, line number, and unformatted integer arguments will be displayed on message unpack on the external PC.
- **HOST_CMN_LOG_IMPLEMENTATION_PACKED** – A host logging implementation that uses proprietary tools to pack formatted log arguments into an efficient look-up table so that code-space usage is minimized. However, this method requires out-of-band delivery of a matching unpack information (aka log dictionary file) used by the On-Ramp Wireless host tools for unpacking.

☞ The rACM implementation uses the PACKED option for serial logging. Two Python tools located in the build directory, `gen_hash_stamp.py` and `gen_log_dictionary.py` are used by the build tools to build a log dictionary file (`host_app.logdict`) used by the RPMA Wireless host logging tools. It is recommended that any custom application development that is targeted for small memory footprints borrow/leverage this method of logging.

Regardless of the logging method, the Host Common utilities have configurable support for the following levels of logging for a particular selected log level, that level and all others at higher priority will be compiled in:

- **HOST_CMN_LOG_LEVEL_NONE** – Set to disable (and even compile out) the Host Common Logging feature.
- **HOST_CMN_LOG_LEVEL_TRACE** – Trace level logging.
- **HOST_CMN_LOG_LEVEL_INFO** – Information level logging.
- **HOST_CMN_LOG_LEVEL_WARN** – Warning level logging.
- **HOST_CMN_LOG_LEVEL_ERR** – Error level logging.
- **HOST_CMN_LOG_LEVEL_ALL** – All levels of logging enabled.

The last set of Host Logging configuration parameters are used to limit logging to a selected zone within the application (i.e., functional components). This is implemented via a bitmask with the currently defines zones used by `host_cmn`:

- **HOST_CMN_LOG_SPI_ZONE** – `host_cmn` logging of the Host->Module SPI transfer protocol (bit#0).
- **HOST_CMN_LOG_MSG_ZONE** – `host_cmn` logging of message routine within the communications framework (bit#1).
- **HOST_CMN_LOG_IMGMAN_ZONE** – `host_cmn` logging of image manager events (bit#2).
- **HOST_CMN_LOG_MGR_ZONE –** `host_cmn` logging of the host process manager (bit#3).

Additional zones will need to be defined and enabled on a per-application basis – the rACM software has created a `HOST_CMN_LOG_APP_ZONE` that encompasses all application-specific log strings.

When configured, the application can invoke the Host Common Logger through use of the following function macros defined in `host_cmn_log.h`.

3.11.1 HOST_CMN_LOG_TRACE()

The application uses this macro function API to log `HOST_CMN_LOG_LEVEL_TRACE` log strings using the configured method of logging. The parameters for this API that need to be passed in are as follows:

- `ZONE` – The bitmask `HOST_CMN` zone definition to associate the log statement with.
- `FMT` –The `printf()` style format string.
- `VA_ARGS` – varargs arguments supplied for the format string.

☞ When using the `HOST_CMN_LOG_IMPLEMENTATION_FIXED`, only a maximum of 3 varargs are allowed.

3.11.2 HOST_CMN_LOG_INFO()

The application uses this macro function API to log `CMN_LOG_LEVEL_INFO` log strings using the configured method of logging. The parameters for this are identical to the Trace-Level Macro (see section 3.11.1).

3.11.3 HOST_CMN_LOG_WARN()

The application uses this macro function API to log `CMN_LOG_LEVEL_WARN` log strings using the configured method of logging. The parameters for this are identical to the Trace-Level Macro (see section 3.11.1).

3.11.4 HOST_CMN_LOG_ERR()

The application uses this macro function API to log `CMN_LOG_LEVEL_ERR` log strings using the configured method of logging. The parameters for this are identical to the Trace-Level Macro (see section3.11.1).

3.11.5 HOST_CMN_LOG_ALWAYS()

The application uses this macro function API to log `CMN_LOG_LEVEL_ALWAYS` log strings using the configured method of logging. The parameters for this are identical to the Trace-Level Macro (see section 3.11.1).

3.12 Stream-to-Packet utilities

If the Host Serial Interface (see section 3.3) has been implemented on the Host Application to configure and leverage the Host Common Stream-to-Packet utilities, then the `host_cmn` component is responsible for managing a circular buffer where the receive stream bytes are sent by the peripheral drivers. The Host Common run-time process loop will perform all of the stream parsing necessary to convert the incoming character stream to an RPMA format message (see rACM Software High Level Design document [4]). When the message has been routed through the internal communication framework to the destination endpoint, `host_cmn` will free the parsed data bytes from the circular buffer. With this type of implementation, the application only needs to perform the following actions in its Host Interface driver:

- Get a pointer to the `host_cmn` Stream-To-Packet Buffer Descriptor struct (section 3.3.1) though the use of the `HOST_CMN_MGR_GetStreamBuff()` API.
- Write each Rx byte or continuous buffer of bytes, as it receives them, to the circular buffer pointed to by Stream-To-Packet Buffer descriptor through the use of the `HOST_CMN_BUFF_Write()` API (section 3.3.2).

Alternatively, receive data can be passed into the `host_cmn` communication framework by invoking the `HOST_CMN_MGR_ProvideControllerStreamData()` API, providing at most `HOST_CMN_MGR_STREAM_MAX_CHARS` bytes per write. In either case, writes to the stream to packet system must be followed by a call to the `HOST_CMN_MGR_Process()` interface to consume the data.

If the Reliable Host Transfer protocol is enabled via the `HOST_CMN_INCLUDE_RHT` configuration option, the `host_cmn` component will perform all message integrity checks and retry protocol base on the incoming stream data. No other actions are required by the application Host Serial Interface drivers or application hooks other than enable/disable this feature during run-time initialization (see section 3.5).

For completeness, the following subsections describe the Stream-To-Packet utility APIs that the application will need to interface with:

3.12.1 HOST_CMN_MGR_GetStreamBuff()

The application can use this API to get a reference to the circular buffer used for incoming stream data. It returns a pointer to a buffer descriptor, `host_cmn_buf_t`, that is managed internally within `host_cmn`.

☞ Any time data is inserted into the `host_cmn` circular buffer used for streaming, `HOST_CMN_MGR_Process()` must be called by the application.

3.12.2 HOST_CMN_HAL_ProvideControllerStreamData()

The application can use this API to take a set of bytes received from the Host Serial Interface, not yet formed into a complete message, and pass in to the internal `host_cmn` circular queue. These parameters are required:

- `uint8_t *buff`– Pointer to the receive data.
- `uint16_t len` – Number of bytes of receive data.

☞ Unlike other `host_cmn` interfaces, this routine may be called from other execution contexts, for instance out of a serial interrupt service routine.

Appendix

A List of acronyms

Abbreviation / Term	Explanation / Definition
ACK	Acknowledgement
AP	Access Point. The RPMA network component geographically deployed over a territory.
API	Application Programming Interfaces
BSP	Board Support Package
CLK	Clock
dNode	A third generation, small form factor, wireless network module developed by On-Ramp Wireless that works in combination with various devices and sensors and communicates data to an Access Point. Also referred to as Node.
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
HLD	High Level Design
LPT	Low Power Timer
LSB	Least Significant Byte
MCU	Microcontroller Unit
MISO	Master-In, Slave-Out
MOSI	Master-Out, Slave In
MRQ	Master Request
MSB	Most Significant Byte
Node	RPMA module.
NVM	Non-volatile Memory
On-Ramp Wireless	Old name for the RPMA, the Ingenu proprietary wireless communication technology and network.
ORW	On-Ramp Wireless
OTA	Over-the-Air
PDU	Protocol Data Unit
PHY	Physical Layer
POR	Power-On Reset
RAM	Random Access Memory
RPMA	Random Phase Multiple access
rACM	Reference Application Communication Module
RTOS	Real Time Operating System
RX	Receive
SDU	Service Data Unit
SPI	Synchronous Peripheral Interface
SRDY	Slave Ready
SRQ	Slave Request
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
TOUT	Timing OUT
TX	Transmit
UI	Uplink Interval
UNIL	Universal Node Interface Library

Related documents

[1] u-blox NANO-S100 series Data Sheet, Docu No UBX-16025707

[2] u-blox NANO-S100 series System Integration Manual, Docu No UBX-16026400

[3] Ingenu rACM example software and code

[4] Ingenu rACM Developer Guide Docu. No 010-0105-00

Revision history

Revision	Date	Name	Comments
R01	10-Oct-2015	rsus	Initial release
R02	10-Mar-2017	clee	Early Production Information Added clarification between the term node and module
R03	19-Oct-2017	clee	"Disclosure restriction" replaces "Document status" on page 2 and document footer.

Contact

For complete contact information visit us at www.u-blox.com

u-blox Offices

North, Central and South America

u-blox America, Inc.

Phone:	+1 703 483 3180
E-mail:	info_us@u-blox.com

Regional Office West Coast:

Phone:	+1 408 573 3640
E-mail:	info_us@u-blox.com

Technical Support:

Phone:	+1 703 483 3185
E-mail:	support_us@u-blox.com

Headquarters
Europe, Middle East, Africa

u-blox AG

Phone:	+41 44 722 74 44
E-mail:	info@u-blox.com
Support:	support@u-blox.com

Documentation Feedback

E-mail:	docsupport@u-blox.com

Asia, Australia, Pacific

u-blox Singapore Pte. Ltd.

Phone:	+65 6734 3811
E-mail:	info_ap@u-blox.com
Support:	support_ap@u-blox.com

Regional Office Australia:

Phone:	+61 2 8448 2016
E-mail:	info_anz@u-blox.com
Support:	support_ap@u-blox.com

Regional Office China (Beijing):

Phone:	+86 10 68 133 545
E-mail:	info_cn@u-blox.com
Support:	support_cn@u-blox.com

Regional Office China (Chongqing):

Phone:	+86 23 6815 1588
E-mail:	info_cn@u-blox.com
Support:	support_cn@u-blox.com

Regional Office China (Shanghai):

Phone:	+86 21 6090 4832
E-mail:	info_cn@u-blox.com
Support:	support_cn@u-blox.com

Regional Office China (Shenzhen):

Phone:	+86 755 8627 1083
E-mail:	info_cn@u-blox.com
Support:	support_cn@u-blox.com

Regional Office India:

Phone:	+91 80 4050 9200
E-mail:	info_in@u-blox.com
Support:	support_in@u-blox.com

Regional Office Japan (Osaka):

Phone:	+81 6 6941 3660
E-mail:	info_jp@u-blox.com
Support:	support_jp@u-blox.com

Regional Office Japan (Tokyo):

Phone:	+81 3 5775 3850
E-mail:	info_jp@u-blox.com
Support:	support_jp@u-blox.com

Regional Office Korea:

Phone:	+82 2 542 0861
E-mail:	info_kr@u-blox.com
Support:	support_kr@u-blox.com

Regional Office Taiwan:

Phone:	+886 2 2657 1090
E-mail:	info_tw@u-blox.com
Support:	support_tw@u-blox.com