Alexa Connect Kit Device SDK

Version 3.1.202002192328
02/19/2020

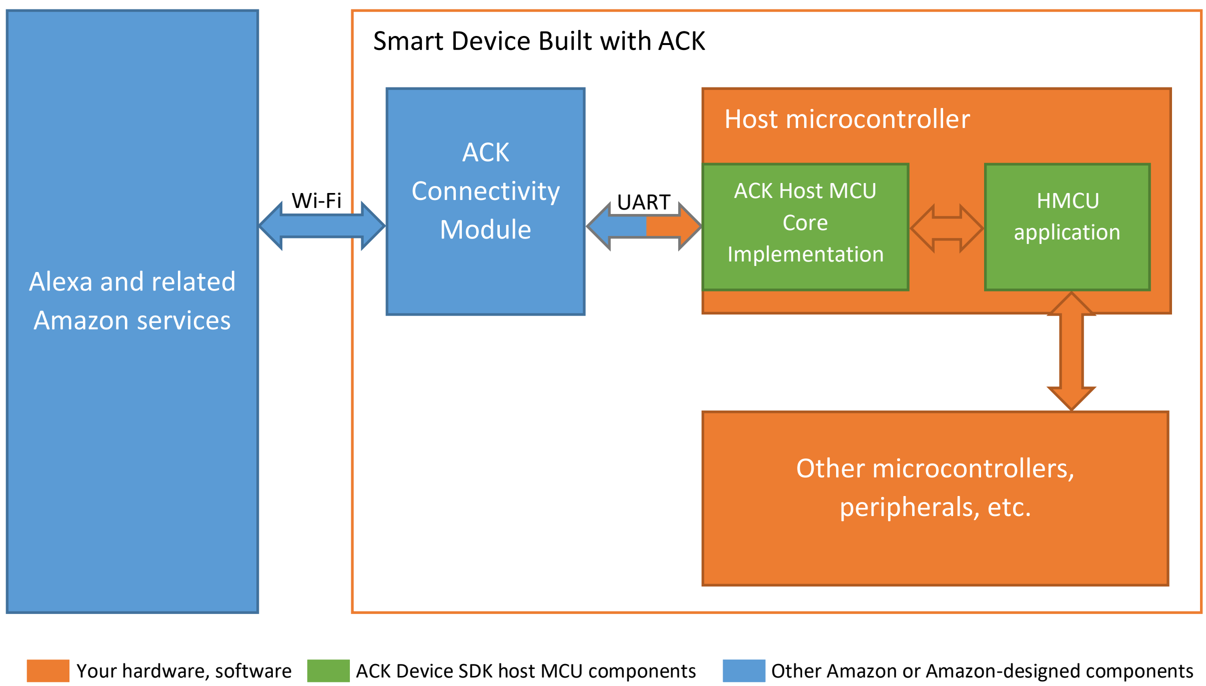
Contents

1. Alexa Connect Kit Device SDK

The Alexa Connect Kit (ACK) Device SDK contains C source code for your device's *Host Microcontroller* ("HMCU"). Components include:

- The *ACK Host MCU Implementation Core* ("ImplCore"), which you integrate into your device's HMCU application.
- A set of *ACK Host MCU sample applications* you can use as starting points for your device's HMCU application, or as a reference.

The overall architecture is shown in this diagram. The orange sections indicate components you implement, using code included in this SDK shown in green.



ImplCore is primarily concerned with four general categories of operation.

- *Lifecycle* management and progression. This refers to various states of registration with and connectivity to the Alexa service. ImplCore delivers Lifecycle state changes to your HMCU application, which reacts by updating an end user-facing LCD display, LEDs, or similar. ImplCore solicits input to the Lifecycle state progression from your HMCU application. Your HMCU application responds to the requests based on the end user's interaction with a keypad, buttons, or similar.
- Dispatching incoming Alexa directives and other commands -- collectively, *Incoming Events* -- to your HMCU application code from the Alexa service. Your HMCU application reacts by changing the state of the hardware and reporting completion of the directive back to ImplCore.
- Sending *Outgoing Events* from your HMCU application code to the Alexa service, to inform Alexa about changes made to device state locally by the end user.
- Sending *Logs and Metrics* to record events that happen on the device. Logs can be used to examine a sequence of events on a particular device. Metrics can be used to get insight on how your device is being used.

For each of these categories, ImplCore operates at a high level of abstraction. Your HMCU application is not directly concerned with low-level details associated with the UART protocol, decoding Incoming Events, encoding Outgoing Events, or managing the ACK Connectivity Module.

ImplCore code runs only at well-defined points in your HMCU application, which you determine. ImplCore neither assumes nor requires, any real-time OS, or asynchronous or background processing model in your HMCU application. ImplCore does assume that your application calls ImplCore regularly and frequently, such as from your application's main loop.

1.1 Terminology

Terms used in this document:

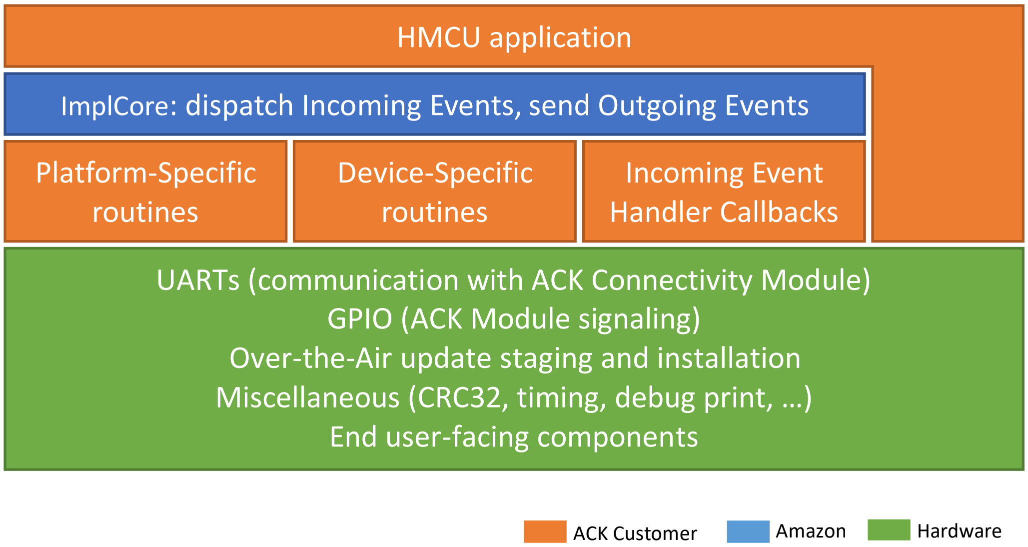
ACK	Alexa Connect Kit
Directive	An Alexa command sent to your device to carry out a user's voice request, or a request made in the Alexa app on a mobile device.
Handler Callback, Incoming Event Handler Callback	A routine you write in your HMCU Application. ImplCore delivers a specific kind of Incoming Event to each routine.
ImplCore	Alexa Connect Kit Device SDK Host MCU Implementation Core. This term refers explicitly to the code in the ACK Device SDK that you don't modify.
HMCU, Host MCU, Host Microcontroller	The microcontroller in your device connected via UART to the ACK Connectivity Module.
Incoming Event	An Alexa directive or ACK command sent to your device to request that it carry out some action.
Outgoing Event	A response you send to an Alexa directive, or a command you issue to Alexa to perform an asynchronous action.

1.2 Component Overview

At a high level, the ACK Device SDK is structured to clearly segregate code you write from ImplCore code you should never need to modify.

- ***ImplCore***, which delivers Incoming Events to *Handler Callbacks* in your HMCU application and allows your HMCU application to send Outgoing Events. You should not need to modify ImplCore.
- ***Platform-Specific*** routines, which bind ImplCore to a particular HMCU and hardware configuration. Generally speaking, you implement platform-specific routines for your device's HMCU. However this task can be eased considerably if you start with one of the HMCU-specific ports included with ImplCore.
- ***Device-Specific*** routines, which bind ImplCore to aspects of your device's end user interaction model that ImplCore needs to understand to manage Lifecycle progression. You implement these routines.
- Besides those components, there are a number of Host MCU sample applications. These act as starting points for your own HMCU applications, and demonstrate how to use the ACK Device SDK.

ImplCore does not interact directly with your hardware. Instead, it delegates hardware-specific access to your Platform-Specific routines, Device-Specific routines, and Handler Callbacks. Your HMCU application of course interacts with your device's hardware in whatever ways make sense for the specifics of its design. ImplCore does not attempt to model or manage your device hardware or how your application interacts with it.



Integrating ImplCore into your HMCU application's main loop involves calling a single ImplCore routine, which dispatches to your Incoming Event Handler Callbacks as needed. You also invoke ImplCore routines to report the outcome of those directives, and to send Outgoing Events that are not directive responses.

Basic programmatic familiarity with Alexa capabilities and directives is assumed in this document. See https://developer.amazon.com/docs/device-apis/message-guide.html for more information.

1.3 Code Organization

Code for the HMCU components of the ACK Device SDK are organized into several directories.

- **At the root:** various readme files and notices related to copyrights and open-source software.
- **applications**: ACK Host MCU sample applications that demonstrate how to use the ACK Device SDK on various kinds of devices. These are the source code only; project structure is under the user\platform directory.
- **core**: Main body of ImplCore. Compile all of these files into your HMCU application. Ensure that core, core\generated, core\generated\v3avs_capabilities, and core\generated\v3avs_types are in your compiler's include path.
- **doc:** additional documentation, including diagrams which illustrate various data flows. You can open the .xml files at https://www.draw.io.
- **external:** open-source software such as Nanopb. Compile all of these files into your HMCU application. Ensure that external\nanopb is in your compiler's include path.
- **include**: C header files you #include in your HMCU application in order to consume ImplCore.
    - ack.h: top-level header file

    - Other ack_*.h: declarations of Incoming Event Handler Callback functions you implement to handle Incoming Events such as Alexa directives, and of functions you call to populate Outbound Events with properties representing the state of your device.
    - details: header files included automatically by other header files. You do not need to explicitly #include these in your code.
    - ack_user_platform.h: declares the Platform-Specific routines you implement to port ImplCore to your own MCU.
    - ack_user_device.h: declares the Device-Specific routines you implement to help ImplCore manage Lifecycle progression.
- **user:** platform-specific code, and project structure for sample applications in the applications directory.

C routines in ImplCore follow a naming scheme to help you identify how to use them.

- Routines starting with "ACK_" are routines your HMCU application calls.
- Routines starting with "ACKUser_" are routines you implement, i.e. Platform- or Device-Specific routines. ImplCore calls these to pass Incoming Events to your HMCU application, and to ask your HMCU application about your device's end user-facing operational state.
- Routines starting with "ACKPlatform_" are routines you implement to port ImplCore to your MCU (or ones you modify, for MCU ports supplied with ImplCore).

Other ImplCore routines -- without one of the above prefixes -- are internal in nature. You do not consume them directly.

1.4 'User' Code vs. Amazon Code

Code in the core and include directories is ACK Device SDK ImplCore which you should not need to modify. Code in the external directory is code sourced from outside of Amazon but consumed by ImplCore. You should not need to modify that code either.

All code you must modify to create your HMCU application is under the user directory. Include files (in the include directory) whose names start with "ack_user_" declare the required routines.

User-supplied component	**Header file declaring required implementation (in include directory)**	**Location of supplied implementations (in user directory)**	**Description**
Platform-Specific routines	ack_user_platform.h	platform\arduino platform\stm32f030-nucleo	~10 functions ImplCore uses to perform platform-specific functions
Device-Specific routines	ack_user_device.h	ack_user_device.c	~7 functions ImplCore uses to gather information about the end user state of your device, to manage Lifecycle progression
Compile-Time Configuration	ack_user_config.h (not in include directory)	applications* (not under user directory)	Compile-time configuration (see section 2 below)

1.5 Concurrency Model

All ImplCore routines are synchronous in nature. No deferred processing of any sort occurs inside ImplCore. Also, no communication occurs over the UART channel between HMCU and ACK Connectivity

Module except inside those synchronous routines. This helps ensure that your HMCU application operates as it did before you integrated ImplCore with it, so things like timing-sensitive operations in your HMCU application are not disturbed.

Note that for a given microcontroller family, reading from the UART may involve interrupt-based or DMA-based operation (see the Platform-Specific section below). However this does not invalidate the simple concurrency model described above because the ACK Connectivity Module never writes to the UART communications channel except in response to requests originating in your HMCU application.

2. Integration with Your HMCU Application

This section discusses how you integrate ImplCore with your HMCU application. If you start with one of the HMCU sample applications included in the ACK Device SDK, basic structure for some of these integration points are already performed.

2.1 Selecting Features and Diagnostics

Alexa Connect Kit features and certain ImplCore configuration options are selected at compile time with appropriate #define's in ack_user_config.h. Every file supplied with ImplCore #include's ack_user_config.h, which you must therefore provide in your HMCU application project.

2.1.1 Debug Printing

ImplCore includes a debug printing feature. Messages are filtered by level as defined in ack_user_config.h. Messages are ultimately passed to the ACKPlatform_DebugPrint Platform-Specific routine for output (see the Outgoing Events section below).

Valid levels range from no debug printing at all (when ACK_DEBUG_PRINT_LEVEL is undefined) to verbose (ACK_DEBUG_PRINT_LEVEL_DEBUG). See ack_debug_print.h for more information.

```
// Debug print behavior. Leave ACK_DEBUG_PRINT_LEVEL undefined for no debug printing.
#define ACK_DEBUG_PRINT_LEVEL ACK_DEBUG_PRINT_LEVEL_INFO
```

Enabling debug printing adds at least 15K of flash consumption due to the messages themselves, as well as the need for the sprintf family of functions from the C runtime library. Debug printing is enabled in every HMCU sample application supplied with ImplCore, but generally you should turn if off or set the log level to ACK_DEBUG_PRINT_LEVEL_ERROR at most for production code.

2.1.2 Asserts

ImplCore uses run-time assertions to check various invariant conditions. An assertion failure causes a debug print message at ACK_DEBUG_PRINT_LEVEL_CRITICAL, after which the code enters an infinite loop which does nothing in its body. You control whether assertions are enabled or disabled, by adding or omitting the following in your ack_user_config.h.

```
// Turn on to enable asserts in Alexa Connect Kit functions.
// Recommend leaving off in production to save space.
#define ACK_ENABLE_ASSERT
```

Disabling asserts causes the checks to be skipped, which saves space but causes undefined behavior if conditions occur that would have failed the checks. You should run with assertions enabled during development, and turn them off for production.

2.1.3 Memory Management

Because ImplCore cannot know ahead of time exactly which features you will require, and due to the nature of the UART communications protocol between your HMCU and the ACK Connectivity Module, ImplCore employs a dynamic memory management scheme. You must declare the size of a buffer that the dynamic memory manager will manage, with a line like the following in your ack_user_config.h.

```
// Size of buffer from which memory blocks are allocated during processing.
#define ACK_MEMORY_POOL_SIZE 1024
```

ImplCore includes a diagnostic feature intended to determine the required buffer size. By default, the diagnostic feature is enabled when asserts are enabled (see the Asserts section above). When enabled, the dynamic memory manager maintains a high-water mark, which is the maximum amount of dynamic memory that has ever been consumed during the processing of an incoming event. You can start ACK_MEMORY_POOL_SIZE at a large value such as 1K, and then observe the value of the sg_maximumHeapletConsumed variable in ack_core_heaplet.c. That value is debug printed periodically as well:

```
[INF: HeapletSetSize:134] Heaplet high-water consumption: 412 bytes
```

You should fully exercise and test your scenarios, and then use the high water mark plus a 10% pad as the actual production value for ACK_MEMORY_POOL_SIZE.

Note: the size you choose must be aligned properly for your HMCU's bitness and memory architecture. For an 8-bit HMCU there may not be any alignment requirement, whereas for a 32-bit HMCU the alignment requirement is typically a sizeof(void*) boundary.

On Arduino Zero, void* is 4 bytes, and so 512 is a valid value for ACK_MEMORY_POOL_SIZE, but 509, 510, and 511 are not valid values.

Some MCUs include specialized memory, such as regions that are not consumed by standard images. ImplCore does not understand how to use such memory for its buffer.

2.1.4 Supported Capabilities

Once you have determined which Alexa capabilities your device will support, you include ImplCore support for each of them by using a corresponding #define in your ack_user_config.h and including a related header file from the include directory.

```
// Capabilities to support.
// For this sample application, we're using only Alexa PowerController.
#define ACK_POWER_CONTROLLER
//#define ACK_PERCENTAGE_CONTROLLER
```

You can support as many capabilities from the table below as your device requires.

Supported capability	#define symbol(s) in ack_user_config.h	Header file to include in addition to ack.h
BrightnessController	ACK_BRIGHTNESS_CONTROLLER	ack_brightness_controller.h
ColorController	ACK_COLOR_CONTROLLER	ack_color_controller.h
ColorTemperatureController	ACK_COLOR_TEMPERATURE_CONTROLLER	ack_color_temperature_controller.h
Cooking and Cooking.ErrorResponse	ACK_COOKING	ack_cooking.h
Cooking.FoodTemperatureController	ACK_COOKING ACK_COOKING_FOOD_TEMPERATURE_CONTROLLER	
Cooking.FoodTemperatureSensor	ACK_COOKING ACK_COOKING_FOOD_TEMPERATURE_SENSOR	
Cooking.PresetController	ACK_COOKING ACK_COOKING_PRESET_CONTROLLER	
Cooking.TemperatureController	ACK_COOKING ACK_COOKING_TEMPERATURE_CONTROLLER	
Cooking.TemperatureSensor	ACK_COOKING ACK_COOKING_TEMPERATURE_SENSOR	
Cooking.TimeController	ACK_COOKING ACK_COOKING_TIME_CONTROLLER	
ModeController	ACK_MODE_CONTROLLER	ack_mode_controller.h
PercentageController	ACK_PERCENTAGE_CONTROLLER	ack_percentage_controller.h
PowerController	ACK_POWER_CONTROLLER	ack_power_controller.h
RangeController	ACK_RANGE_CONTROLLER	ack_range_controller.h
TimeHoldController	ACK_TIME_HOLD_CONTROLLER	ack_time_hold_controller.h
ToggleController	ACK_TOGGLE_CONTROLLER	ack_toggle_controller.h

Including ImplCore support for a capability requires you to implement the corresponding ACKUser_OnXxxDirective routine(s) declared for the capability in the header files shown in the table above. Not supporting a capability omits its code and data from your application.

Each selected capability generally consumes about 300-400 bytes of flash memory, but this varies depending on a capability's complexity.

2.2 Initializing ImplCore

Before using any ImplCore functionality, your HMCU application must initialize it by calling ACK_Initialize. Note: functionality to cleanly terminate ImplCore without rebooting is not provided.

The ACK HMCU sample applications call ACK_Initialize before entering their main loop.

```
// Called once, for one-time initialization.
void setup()
{
    // Initialize ACK Host MCU Implementation Core.
    ACK_Initialize();
}
```

2.3 Main Loop

After initialization, your main loop must periodically call ACK_Process to advance Lifecycle state and dispatch Incoming Events such as Alexa directives to your handler callback routines. Your handler callback routines report the outcome of processing Incoming Events back to ImplCore, which automatically sends directive responses back to Alexa.

Work associated with Lifecycle management and processing Incoming Events is performed *only* inside ACK_Process. ImplCore has no background or asynchronous constructs.

You interleave calls to ACK_Process with whatever other work your HMCU application is doing to run your device.

```
// Called over and over, for main processing.
void loop()
{
    ACK_Process();

    // ...
}
```

ACK_Process is intended to be called on every pass through your HMCU Application’s main loop. The frequency with which you call ACK_Process is a critical factor in how responsive your device appears to end users.

Note: ACK_Process *must* be called periodically regardless of Lifecycle state.

3. Lifecycle Processing and Management

As you repeatedly call ACK_Process from your main loop, the device advances through a series of Lifecycle states related to setup and connectivity from the point of view of the ACK Connectivity Module.

When the Lifecycle state changes, your ACKUser_OnLifecycleStateChange Device-Specific routine is called. Each of these states may also correspond to end user-facing indicators on your device, e.g. a Wi-Fi connected LED, or a flashing light or clock during device setup, etc.

State (ACK_LIFECYCLE_*, in ack.h)	How entered	Next state
UNAVAILABLE The ACK Connectivity Module is not available.	Power up; or ImplCore reboots the ACK Connectivity Module to let it apply a firmware update to itself.	BOOTED once the ACK Connectivity Module finishes booting/rebooting.
BOOTED The ACK Connectivity Module has finished booting/rebooting and is ready to dispatch Incoming Events and to send Outgoing Events.	Automatically after power up, or ACK Connectivity Module completes reboot following application of a firmware update to itself.	FACTORY_RESET_IN_PROGRESS if the end user initiates factory reset using your device's button press, keypress, or similar; NOT_REGISTERED, IN_SETUP_MODE, or NOT_CONNECTED_TO_ALEXA as appropriate otherwise.
FACTORY_RESET_IN_PROGRESS A module factory reset you initiated is in progress.	The end user uses your device's button press, key sequence, or similar to initiate factory reset, and the device is not in use.	BOOTED when factory reset is complete.
NOT_REGISTERED The ACK Connectivity Module is not registered and setup is not in progress.	The device powers up after having failed a previous setup, or the setup process times out.	IN_SETUP_MODE or FACTORY_RESET_IN_PROGRESS if the end user uses your device's button press, keypress, or similar to initiate setup or factory reset.
IN_SETUP_MODE End user setup is active.	The device enters setup mode when powered up for the first time after purchase; or the end user uses your device's button press, keypress, or similar to initiate setup.	NOT_CONNECTED_TO_ALEXA when setup is complete; IN_SETUP_MODE or FACTORY_RESET_IN_PROGRESS if the end user initiates setup or factory reset with your device's button press, keypress, or similar.
CONNECTED_TO_ALEXA The device is set up and the ACK Connectivity Module is connected to the Alexa services.	Setup previously completed successfully and the device has a Wi-Fi connection to the Alexa services.	NOT_CONNECTED_TO_ALEXA if Wi-Fi connectivity drops; IN_SETUP_MODE or FACTORY_RESET_IN_PROGRESS if the end user initiates setup or factory reset with your device's button press, keypress, or similar
NOT_CONNECTED_TO_ALEXA The device is set up but the ACK Connectivity Module is not connected to the Alexa services.	The device is fully set up, but the ACK Connectivity Module cannot connect to the Alexa services over Wi-Fi, or the connection drops.	CONNECTED_TO_ALEXA if Wi-Fi connectivity is established or reestablished; IN_SETUP_MODE or FACTORY_RESET_IN_PROGRESS if the end user initiates setup or factory reset with your device's button press, keypress, or similar.

Lifecycle state is maintained in the ACK_LifecycleState global variable in ImplCore. From ack.h:

```
extern ACKLifecycleState_t ACK_LifecycleState;
```

3.1 Additional Lifecycle State Info

The **ACK_LIFECYCLE_IN_SETUP_MODE** lifecycle state has additional information associated with it. Your HMCU application can use this information for end user-facing display.

```
typedef struct _ACKSetupModeInfo_t
{
    acp_setup_stages SetupStage;

    unsigned IsUserGuidedSetupActive: 1;
    unsigned IsBarcodeScanSetupActive: 1;
```

```
        unsigned IsZeroTouchSetupActive: 1;
    }
    ACKSetupModeInfo_t;
```

SetupState Value	Description
acp_setup_stages_none	Data not available or irrelevant in current Lifecycle state.
acp_setup_stages_discoverable	Setup is waiting for the user to connect (using a registration app on their phone).
acp_setup_stages_setup_in_progress	The end user has connected and setup is executing.
acp_setup_stages_registered	The ACK Connectivity Module is registered to the end user's account.
acp_setup_stages_timeout	Setup has exited but the user is not registered.

Is*SetupActive Field	Description
`IsUserGuidedSetupActive`	User-Guided Setup is active on the ACK Connectivity Module.
`IsBarcodeScanSetupActive`	Bar Code Scan Setup is active on the ACK Connectivity Module.
`IsZeroTouchSetupActive`	Zero-Touch Setup is active on the ACK Connectivity Module.

ImplCore maintains the additional lifecycle state information in the ACK_LifecycleSubStateInfo global variable.

```
    extern ACKLifecycleSubStateInfo_t ACK_LifecycleSubStateInfo;
```

3.2 Device-Specific Routines

Managing Lifecycle progression requires ImplCore to understand certain aspects of your device's end user-related operational state. When ImplCore needs this information, it calls routines with known names, which you must implement as part of your HMCU application. These Device-Specific routines bind ImplCore to your device's interaction model for Lifecycle-related input from and output to the end user.

ACKUser_GetFirmwareVersion is called when ImplCore needs the version of your firmware. This can be any 64-bit value meaningful for your firmware versioning scheme.

ACKUser_IsDeviceInUse is called when ImplCore needs to know whether your device is in use in a way that's meaningful to the end user. If a device is in use, certain lifecycle processing is deferred until the device is not in use. For example, firmware updates are not initiated until this returns false.

ACKUser_DoesUserWantFactoryReset is called when ImplCore needs to know whether the user has requested factory reset via your device-specific factory reset initiation mechanism (the user presses and holds a "reset" button, or enters a certain keystroke on a keypad, etc.).

ACKUser_DoesUserWantUserGuidedSetup is called when ImplCore needs to know whether the user has requested that the device enter setup mode, based on your device-specific User-Guided Setup initiation mechanism (the user presses and holds a "setup" button, or enters a certain keystroke on a keypad, etc.).

ACKUser_DoesUserWantToSubmitLogs is called when ImplCore needs to know whether the user has requested that the device upload logs, based on your device-specific initiation mechanism for submitting logs (the user presses and holds a specific button on the device, or enters a certain keystroke on a keypad, etc.).

ACKUser_OnLifecycleStateChange is called when ImplCore has changed the Lifecycle state. Your application should update user-facing display elements relating to Lifecycle and Connectivity state. For example, lighting a "connected to Alexa" LED, or flashing an LED during factory reset, or flashing a clock when in setup mode. State is maintained in the global variables g_lifecycleState and g_lifecycleSubStateInfo.

ACKUser_EraseUserSettings is called when factory reset is initiated from the service side (as opposed to the user initiating it locally using your device-specific initiation mechanism). Your application should erase persisted user settings.

3.3 Lifecycle Progression

ACK_Process looks for various conditions and advances Lifecycle state or performs various actions as required.

First priority is rebooting the ACK Connectivity Module if a reboot is pending. If not, then any pending factory reset is initiated. If none, then the connectivity state is examined, and finally the next Incoming Event (such as an Alexa directive) – if any – is dispatched to the relevant Incoming Event Handler Callback. Finally, if there was a directive dispatched to your handler callback, ACK_Process sends a response based on the outcome you reported via the appropriate ACK_Complete* routine.

3.3.1 Servicing Reboot Requests

The ACK Connectivity Module requests to be rebooted when it has received a firmware update for itself, or when a factory reset is pending (see next section). A pending reboot is carried out only when the device is not in use.

When ImplCore reboots the ACK Connectivity Module, it changes the lifecycle state to ACK_LIFECYCLE_UNAVAILABLE. When the ACK Connectivity Module finishes rebooting, lifecycle transitions to ACK_LIFECYCLE_BOOTED.

3.3.2 Initiating Factory Reset

ImplCore assumes that a factory reset would be initiated by some end user action such as a reserved keypress on your device’s keypad. Factory reset requests are ignored if the device is in use, if the ACK Connectivity Module is unavailable because it’s rebooting, or if there’s already a factory reset in progress.

After initiating a factory reset, the ACK Connectivity Module will eventually request to be rebooted (see previous section). When it reboots, the factory reset occurs, after which lifecycle will automatically transition to ACK_LIFECYCLE_BOOTED.

3.3.3 Updating Setup and Connectivity State

When the ACK Connectivity Module is booted and not in a factory reset, its setup and connectivity situation becomes relevant.

4. Incoming Events

If an Incoming Event is available when your HMCU application calls ACK_Process, ImplCore calls a specific corresponding Incoming Event handler callback routine. ImplCore hard-codes the names of the routines it calls; you must implement specific routines for the particular Incoming Events appropriate for your device.

Declarations for required handler callback routines are in the header files shown in the table in section 2.1.4 above.

```
// You must implement this routine.
void ACKUser_OnPowerControllerDirective(int32_t correlationId, bool powerOn);
```

See section 2.1 above for more information about how to include or exclude particular ImplCore functionality.

4.1 Alexa Directives

ImplCore supports a broad array of Alexa capabilities. You choose which ones to enable by using compile-time flags (see section 2.1.4 above). If a given Alexa capability is enabled, you must implement one or more matching Handler Callback routines to process the capability's directives. ImplCore calls your handler callbacks *only* from inside ACK_Process.

Supported capability (Alexa.*)	**Routine(s) you must implement to process directives**
BrightnessController	ACKUser_OnBrightnessControllerDirective
ColorController	ACKUser_OnColorControllerDirective
ColorTemperatureController	ACKUser_OnColorTemperatureControllerIncreaseTemperatureDirective ACKUser_OnColorTemperatureControllerDecreaseTemperatureDirective ACKUser_OnColorTemperatureControllerSetTemperatureDirective
Cooking and Cooking.ErrorResponse	ACKUser_OnCookingSetModeDirective
Cooking.FoodTemperatureController	ACKUser_OnCookingFoodTemperatureControllerCookByFoodTemperatureDirective
Cooking.FoodTemperatureSensor	-
Cooking.PresetController	ACKUser_OnCookingPresetControllerCookByPresetDirective
Cooking.TemperatureController	ACKUser_OnCookingTemperatureControllerCookByTemperatureDirective ACKUser_OnCookingTemperatureControllerAdjustCookingTemperatureDirective
Cooking.TemperatureSensor	-
Cooking.TimeController	ACKUser_OnCookingTimeControllerCookByTimeDirective ACKUser_OnCookingTimeControllerAdjustCookTimeDirective
ModeController	ACKUser_OnModeControllerDirective
PercentageController	ACKUser_PercentageControllerDirective
PowerController	ACKUser_OnPowerControllerDirective
RangeController	ACKUser_OnRangeControllerDirective
TimeHoldController	ACKUser_OnTimeHoldControllerHoldDirective ACKUser_OnTimeHoldControllerResumeDirective
ToggleController	ACKUser_OnToggleControllerDirective

For Alexa directives, your handler callback is given a *correlation ID*, which you must pass back when reporting the directive’s outcome back to ImplCore (see section 5.1 below).

```
// You must implement this routine.
void ACKUser_OnPowerControllerDirective(int32_t correlationId, bool powerOn);
```

Section 0 below contains more information about how to handle directives and state report requests.

4.2 State Report Requests

A state report request is an Alexa directive commanding your device to report state for all properties in your device that work with Alexa. You must write a handler callback routine named ACKUser_OnReportStateDirective, which ImplCore calls from ACK_Process when an incoming state report request is received. Generally your handler routine will simply call ACK_CompleteStateReportWithSuccess, but you may also want to send a metric or perform logging.

```
void ACKUser_OnReportStateDirective(int32_t correlationId)
{
    // Sending a metric here is an example of optional processing.
    ACK_SendUsageReportMetric(METRIC_REPORT_STATE_REQUESTED);

    ACK_CompleteStateReportWithSuccess(correlationId);
}
```

5. Outgoing Events

Outgoing Events include responses to Alexa directives, and proactive events which indicate changes in your device’s state unrelated to directives.

You do not directly create and send responses to Alexa directives. Instead, your handler callbacks report the outcome of processing a directive, by calling an ACK_Complete* routine. See section 5.2 below.

To send a proactive Outgoing Event, you call the ACK_Send*Event routine appropriate for the kind of event you want to send. See section 5.3 below.

Some ACK_Complete* and ACK_Send*Event routines require you to specify properties to include in the Outgoing Event. See section 5.1.

Desired Outgoing Event	Response to Incoming Event?	Do you add properties?	Corresponding ImplCore routine you call
Success response to Alexa directive	Yes (Alexa directive)	Yes	ACK_CompleteDirectiveWithSuccess (declared in ack.h)
Error response to Alexa directive	Yes (Alexa directive)	No	ACK_CompleteDirectiveWithSimpleError ACK_CompleteDirectiveWithStartOutOfRangeError ACK_CompleteDirectiveWithTemperatureOutOfRangeError ACK_CompleteDirectiveWithNotSupportedInCurrentModeError (declared in ack.h)
Error response specific to cooking directive Note: For cooking directive errors, you can respond with either a cooking-specific response as shown in this row, or with a general error as shown in the row above	Yes (Alexa cooking directive)	No	ACK_CompleteDirectiveWithCookingError ACK_CompleteDirectiveWithCookingDurationTooLongError (declared in ack_cooking.h)
State report	Yes (Alexa state report request)	Yes	ACK_CompleteStateReportWithSuccess (declared in ack.h)
Change report	No (proactive)	Yes	ACK_SendChangeReportEvent (declared in ack.h)
Dash Replenishment (usage sensor)	No (proactive)	No	ACK_SendDashReplenishmentUsageConsumedEvent (declared in ack_dash_replenishment.h)
Dash Replenishment (level usage sensor)	No (proactive)	No	ACK_SendDashReplenishmentLevelUsageConsumedEvent ACK_SendDashReplenishmentLevelUsageReplacedEvent (declared in ack_dash_replenishment.h)

Note that ACK_Send*Event routines must be called from your main loop. ImplCore is not re-entrant for the purposes of sending an Event such as a change report from inside the processing for directives.

5.1 Properties

Directive response, state report, and change report Outgoing Events send *properties* to the Alexa service. Properties represent the state of the Alexa-controllable parts of your device, such as a power controller being on or off, or an oven temperature.

Although Alexa capabilities and your device properties are related, more than one Alexa capability's directives may operate on a given property. For example, consider a dimmable light bulb which implements the power controller and brightness controller Alexa capabilities; further imagine that the bulb is currently on at any non-0 brightness level, and the user asks Alexa to set the brightness to 0%. In this case, although your device receives only the set-brightness directive it recognizes that setting the brightness to 0% also changes power state from on to off.

5.1.1 Property Ordinals

Within your HMCU application, properties are represented by *property ordinals*. A property ordinal is simply a 0-based index of each property your device ever sends to Alexa. Property ordinals are arbitrary values you determine in your HMCU application; ImplCore does not impose any requirements or constraints on property ordinals, except that they must be between 0 and 31 inclusive. Continuing the dimmable light bulb example, your HMCU application could contain:

```
#define ORDINAL_POWER_STATE_PROPERTY 0
#define ORDINAL_BRIGHTNESS_PROPERTY 1
```

There is no requirement that the power state property be ordinal 0 or that the brightness property be ordinal 1. You can choose any values for ordinals between 0-31 inclusive, in any order.

5.1.2 Representing Properties to ImplCore

Because the precise relationship of directives to affected properties is highly device-specific, the ACK_Complete* and ACK_Send*Event routines for Outgoing Events containing properties require you to designate which properties are included. This is done by using bitfields.

```
void ACK_CompleteDirectiveWithSuccess(
    int32_t correlationId,
    ACKPropertiesBits_t responsePropertiesBits,
    ACKPropertiesBits_t changeReportPropertiesBits);
```

Each bit represents 1 left-shifted by a property ordinal. You can use the ACK_PROPERTY_BIT macro for this. Continuing the dimmable bulb example described above, consider a set-brightness directive that changes the brightness from 30% to 50%. Upon completing HW changes to carry out the directive, your HMCU application would call

```
ACK_CompleteDirectiveWithSuccess(
    correlationId,
    ACK_PROPERTY_BIT(ORDINAL_BRIGHTNESS_PROPERTY),
    ACK_PROPERTY_BIT(ORDINAL_BRIGHTNESS_PROPERTY));
```

Your application may also receive a set-brightness directive that has no effect. In that case your HMCU application would call

```
ACK_CompleteDirectiveWithSuccess(
    correlationId,
    ACK_PROPERTY_BIT(ORDINAL_BRIGHTNESS_PROPERTY),
    0);
```

If you receive a set-brightness directive that changes the power state as well as the brightness, your HMCU application would call

```
        ACK_CompleteDirectiveWithSuccess(
            correlationId,
            ACK_PROPERTY_BIT(ORDINAL_BRIGHTNESS_PROPERTY),
                | ACK_PROPERTY_BIT(ORDINAL_POWER_STATE_PROPERTY)
            ACK_PROPERTY_BIT(ORDINAL_BRIGHTNESS_PROPERTY),
                | ACK_PROPERTY_BIT(ORDINAL_POWER_STATE_PROPERTY));
```

5.1.3 Adding Property Values to Outgoing Events

The previous section explained how you tell ImplCore which properties to include in certain Outgoing Events. This section explains how you actually add property values to the events.

When ImplCore is sending an Outgoing Event that includes properties, it consults a table you must create in your application. The table maps property ordinals to callback routines in your application. Your callback routines call routines in ImplCore to add property values. The table must be named ACKUser_PropertyTable.

```
ACKPropertyTableEntry_t ACKUser_PropertyTable[] =
{
    { ORDINAL_POWER_STATE_PROPERTY, AddPowerStateProperty },
    { ORDINAL_BRIGHTNESS_PROPERTY, AddBrightnessProperty },
    { 0, NULL }
};

// ...


bool AddPowerStateProperty(uint32_t propertyOrdinal, unsigned propertyFlags)
{
    // ...

    ACKError_t error = ACK_AddPowerControllerProperty(/* ... */ g_isPowerOn);

    // ...
}
```

From ack_power_controller.h:

```
// Call this to add a property representing the state of a power controller
// to a response event.
ACKError_t ACK_AddPowerControllerProperty(
    const ACKStateCommon_t* pCommon,
    bool powerOn);
```

5.2 Directive Responses

Your device must respond to each Alexa directive you receive, by calling an ImplCore routine to reporti the outcome of directive processing. ImplCore automatically sends the appropriate corresponding directive response to Alexa.

You report the outcome of processing a directive by calling an ACK_Complete* routine. Typically this is done from inside your handler callbacks. However if your application does not know the outcome immediately and doesn't wish to block waiting for it, you can call an ACK_Complete* routine later. You should do this within 5 seconds of receiving the directive, to avoid Alexa telling the user that something went wrong. The speed with which you send response events directly influences user perception of your device's responsiveness.

When reporting the outcome of directive processing, you provide a *correlation ID*, which Alexa uses to match your response to the directive on which you were operating. The correlation ID is delivered to your handler callback as shown in section 4.1 above.

```
void ACKUser_PercentageControllerDirective(
    int32_t correlationId,
    bool isDelta,
    int32_t value)
{
    ACK_DEBUG_PRINT_I(
        "Percentage controller directive: %s = %d",
        isDelta ? "percentage delta" : "absolute percentage",
        value);

    // ... control something physical attached to the HMCU ...

    // ... create a response event, will be sent later from the main loop
    ACK_CompleteDirectiveWithSuccess(correlationId, /* ... */);

    // NOT ALLOWED HERE.
    ACK_AddPercentageControllerPropertyToEvent(/* ... */);
    ACK_SendChangeReport();
}
```

5.3 Change Reports

Your HMCU application must send an Alexa change report any time the state of hardware reported to Alexa has changed. This includes both local control, e.g. the user turns your device off using the power switch (assuming a device that supports the power controller capability); and when state changes due to an incoming directive, e.g. the user tells Alexa by voice or in the Alexa app to turn your device on or off.

More information about Alexa change reports can be found here. In particular, note the distinction between the property (or properties) which actually changed, vs. other properties (whose state can also be reported in a change report). ACK supports reporting both groups of properties.

5.3.1 In Response to Local Control

If the state of Alexa-controlled hardware in your device changes due to local control or any other reason not associated with a directive Incoming Event, your device must send a change report. This can be done by calling ACK_SendChangeReport.

5.3.2 In Response to a Directive

ImplCore includes functionality to help send a change report as part of responding to directives that actually changed device state.

```
void ACK_CompleteDirectiveWithSuccess(
    int32_t correlationId,
    ACKPropertiesBits_t changedPropertiesBits,
    ACKPropertiesBits_t otherPropertiesBits);
```

If changedPropertiesBits is non-0, ImplCore automatically sends a change report in addition to the directive response.

For example, when the Hello World sample application processes a power controller directive, it causes a change report to be sent as part of the process of sending the directive reply – but only if the power state actually changed. A user can, for example, tell Alexa to turn off a device which is already off. If this happens the Alexa directive has no effect and a change report should not be sent.

The below is from the power controller Handler Callback routine in the Hello World sample application. Before changing the power state of the hardware, the application checks to see whether the directive is *actually* changing the power state.

```
// Power Controller directive callback that turns on a small LED.
void ACKUser_PowerControllerDirective(int32_t correlationId, bool powerOn)
{
    bool changed;

    // ...

    changed = (0 == ACKPlatform_ReadDigitalPin(ACK_HW_PIN_LED_1))
        != (0 == powerOn);

    ACKPlatform_WriteDigitalPin(ACK_HW_PIN_LED_1, powerOn);
```

Then when creating the response, the power state property's bit is given to ACK_CompleteDirectiveWithSuccess if and only if the power state changed.

```
ACK_CompleteDirectiveWithSuccess(
    correlationId,
    changed ? ACK_PROPERTY_BIT(PROPERTY_ORDINAL_POWER_CONTROLLER) : 0,
    ACK_PROPERTY_BIT(PROPERTY_ORDINAL_POWER_CONTROLLER));
```

6. Platform-Specific Routines

Platform-specific code is isolated to a small number of routines whose presence is assumed by ImplCore. You must implement these routines for your Host MCU. The routines are declared in ack_user_platform.h. For reference, an example for Arduino is in ack_arduino_platform.cpp.

ACKPlatform_Initialize initializes your Platform-Specific routines. Your implementation should set up UARTs, GPIO pins, and similar resources needed by the implementations of rest of the ACKPlatform_* functions.

ACKPlatform_TickCount, **ACKPlatform_Delay** provide elapsed milliseconds (can be used for timing) and the ability to delay operation for a period of milliseconds, respectively.

ACKPlatform_Send, **ACKPlatform_Receive**, and **ACKPlatform_DrainRead** provide communications with the ACK Connectivity Module (over a UART).

ACKPlatform_CalculateCrc32 calculates a 32-bit CRC value. This is a platform-specific function because some MCUs have hardware assist for CRC calculation, but note that your implementation must use the same CRC algorithm as is used by the Alexa Connect Kit Connectivity Module. See the implementation in ack_arduino_platform.cpp for more information.

ACKPlatform_DebugPrint writes messages to a platform-specific debug output, typically (but not mandatorily) a UART to which a serial monitor is attached. Your implementation should format the message and send it to the platform-specific output.

ACKPlatform_WriteDigitalPin and **ACKPlatform_ReadDigitalPin** changes and reads the state of a GPIO pin. These are used for signaling the ACK Connectivity Module via its HOST INTERRUPT and RESET pins. They are also used for GPIO in the Host MCU sample applications included in the ACK Device SDK, so that those applications can be coded portably. ImplCore expects your device's hardware to be outside the scope of this mechanism; your HMCU application (including ImplCore Platform-Specific routines, Device-Specific routines, and Handler Callbacks) should work directly with your hardware and not use ACKPlatform_WriteDigitalPin/ACKPlatform_ReadDigitalPin for that purpose.

7. Logging and Metrics

You can send logs and metrics to record events that happen on the device. Logs can be used to examine a sequence of events on a particular device. Metrics can be used to get insight on how your device is being used.

7.1 Logging

In order to enable logging, you must #define ACK_LOGGING in your ack_user_config.h file, and #include ack_logging.h in your code.

When you log a message, it is sent to the ACK Connectivity Module, where it is stored until the user requests that logs be submitted to the service. The ACK Connectivity Module sends logs to the service when your ACKUser_DoesUserWantToSubmitLogs Device-Specific routine returns true. See section 3.2 above.

To log messages, you use these two functions, declared in ack_logging.h.

```
ACKError_t ACK_WriteLog(
    acp_log_level logLevel,
    const char* pComponent,
    const char* pMessage);

ACKError_t ACK_WriteLogFormatted(
    acp_log_level logLevel,
    const char* pComponent,
    const char* pMessageFormat,
    ...);
```

There are three logging levels: acp_log_level_debug, acp_log_level_info, and acp_log_level_error. Using the right level is helpful to diagnose problems in devices. The pComponent parameter is a string meaningful to you representing the component from which a log message originated. The pMessage and pMessageFormat parameters supply the message and the printf-style message format string, respectively.

7.2 Metrics

Metrics are used to record events that happen on the device. There are three types of metrics:

Metric Type	Acp_cmd_record_dem_metric_Type value	Description
USER_PRESENT	acp_cmd_record_dem_Type_USER_PRESENT	Records a metric when a user engages with a device. User engagement is defined by the device maker.
USAGE_REPORT	acp_cmd_record_dem_Type_USAGE_REPORT	Records events that occur on the device as a user interacts with it.
DEVICE_ERROR	acp_cmd_record_dem_Type_DEVICE_ERROR	Records a device error.

Each metric event must have at least one associated data point. A data point has a type, a name, and a value. Names longer than 25 characters are truncated. The type determines what values are allowed for that data point.

Type	Usage	Value Representation
COUNTER	Records a count of events or objects.	32-bit unsigned integer
GAUGE	Records any numerical value.	32-bit floating-point
DISCRETE	Records any single event, with corresponding text. This can be used to record simple occurrence, or an enumerated value meaningful to your device.	string

The ACK_SendMetric function is used to send metrics. This function takes a type and an array of data points. Metrics are sent to the ACK Connectivity Module, where they are stored and later sent in batches.

```
ACKError_t ACK_SendMetric(
```

```
        acp_cmd_record_dem_metric_Type recordMetricsType,
        size_t datapointCount,
        ACKMetricsDatapoint_t* pMetricsDatapoints);
```

To simplify usage, ack_metrics.h also declares additional helper routines to simplify common cases. For example, the ACK_SendDiscreteMetric function can be used to send an event with a single discrete data point.

```
    ACKError_t ACK_SendDiscreteMetric(
        acp_cmd_record_dem_metric_Type metricType,
        const char* pName,
        const char* pText);
```

The following routine can be used to conveniently send an error metric with a single numerical value.

```
    ACKError_t ACK_SendErrorWithValue(const char* pDataName, float value);
```

See ack_metrics.h for more information.

8. Module Diagnostics

ImplCore includes functions to retrieve diagnostic information from the ACK Connectivity Module. There is no specific diagnostic “mode”; your HMCU application can call these routines at any time other than from inside ACKUser_* routines.

```
    // Call this to get device type info from the ACK connectivity module.
    ACKError_t ACK_GetDeviceType(char* pDeviceType, size_t deviceTypeBufferSize);

    // Call this to get provisioning info from the ACK connectivity module.
    ACKError_t ACK_GetProvisioningInfo(
        acp_response_provisioning_provisioning_state* pState);

    // Call this to get the device DSN from the ACK connectivity module.
    ACKError_t ACK_GetHardwareInfo(char* pDsn, size_t dsnBufferSize);

    // Call this to get the ACK connectivity module's firmware version.
    ACKError_t ACK_GetFirmwareVersion(
        uint32_t* pProtocolNumber,
        uint32_t* pBuildNumber,
        char* pIncrementalVersion,
        size_t incrementalVersionBufferSize);
```

To use these functions. You must #define ACK_MODULE_DIAGNOSTICS in your ack_user_config.h file, and #include ack_module_diagnostics.h in your HMCU Application.

For more information, see calls to the above routines made from ack_core_lifecycle.c. Those show the recommended buffer sizes to use with the functions listed above that return strings.

9. Utility Functions

ImplCore includes some utility functions which you may find useful. Code referenced in this section is in the core subdirectory along with the rest of ImplCore.

9.1 Circular Buffer

Your Platform-Specific routines will typically use two UARTs: one for communicating with the ACK Connectivity Module, and one for outputting debug print messages. The two UARTs will frequently be in use simultaneously, for example when a debug print message is being output while ImplCore is trying to receive data from the ACK Connectivity Module. For this reason, a polling implementation for driving the UARTs is not viable; the UARTs must be programmed with a non-blocking strategy such as interrupts or DMA. That in turn requires background-like processing wherein received data is buffered until an application-level read operation retrieves it.

ImplCore includes a reference implementation of a circular buffer for this purpose, in ack_core_circularbuffer.c. Use it via #include ack_circularbuffer.h (from the include directory) in your code.

9.2 CRC Checks

Communications between a Host MCU and the ACK Connectivity Module are protected by cyclic redundancy checks (CRC). A reference implementation of the algorithm that matches the one in the connectivity module is provided in ack_core_crc32.c. Use it via #include ack_crc32.h (from the include directory) in your code.