Kinetis Bootloader 1.1.0 Reference Manual

Rev 0, 12/2014

freescale™

Contents

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

Chapter 1
Introduction

1.1 Introduction....7

1.2 Terminology....7

1.3 Block diagram....8

1.4 Features supported....8

1.5 Components supported....9

Chapter 2
Functional description

2.1 Introduction....11

2.2 Memory map....11

2.3 The Kinetis Bootloader Configuration Area (BCA)....11

2.4 Start-up process....13

2.5 Clock configuration....14

2.6 Bootloader entry point....15

Chapter 3
Kinetis bootloader protocol

3.1 Introduction....17

3.2 Command with no data phase....17

3.3 Command with incoming data phase....18

3.4 Command with outgoing data phase....19

Chapter 4
Bootloader packet types

4.1 Introduction....23

4.2 Ping packet....23

4.3 Ping response packet....24

4.4 Framing packet....25

4.5 CRC16 algorithm....27

4.6 Command packet 28
4.7 Response packet 29

Chapter 5
Kinetis bootloader command API

5.1 Introduction 33
5.2 GetProperty command 33
5.3 SetProperty command 35
5.4 FlashEraseAll command 37
5.5 FlashEraseRegion command 38
5.6 FlashEraseAllUnsecure command 39
5.7 ReadMemory command 40
5.8 WriteMemory command 42
5.9 FillMemory command 44
5.10 FlashSecurityDisable command 46
5.11 ReceiveSBFile command 47
5.12 Execute command 47
5.13 Call command 48
5.14 Reset command 48
5.15 FlashProgramOnce command 49
5.16 FlashReadOnce command 51
5.17 FlashReadResource command 52

Chapter 6
Supported peripherals

6.1 Introduction 55
6.2 I2C Peripheral 55
6.3 SPI Peripheral 57
6.4 UART Peripheral 59
6.5 USB Peripheral 61
6.5.1 Clock configuration 62

6.5.2 Device descriptor 62
6.5.3 Endpoints 64
6.5.4 HID reports 64

Chapter 7
Peripheral interfaces

7.1 Introduction 67
7.2 Abstract control interface 68
7.3 Abstract byte interface 69
7.4 Abstract packet interface 70
7.5 Framing packetizer 70
7.6 USB HID packetizer 70
7.7 Command/data processor 71

Chapter 8
Memory interface

8.1 Abstract interface 73
8.2 Flash driver interface 74
8.3 Low level flash driver 74

Chapter 9
Kinetis bootloader porting

9.1 Introduction 77
9.2 Choosing a starting point 77
9.3 Preliminary porting tasks 77
9.3.1 Download device header files 78
9.3.2 Copy the closest match 78
9.3.3 Provide device startup file (vector table) 78
9.3.4 Clean up the IAR project 79
9.3.5 Bootloader peripherals 80
9.4 Primary porting tasks 82
9.4.1 Header file modification 82

9.4.2 Bootloader peripherals..... 83
9.4.2.1 Supported peripherals..... 83
9.4.2.2 Peripheral initialization..... 83
9.4.2.3 Clock initialization..... 84
9.4.3 Bootloader configuration..... 84
9.4.4 Bootloader memory map configuration..... 85

Chapter 10
Creating a custom flash-resident bootloader

10.1 Introduction..... 87
10.2 Where to start..... 87
10.3 Flash-resident bootloader source tree..... 88
10.4 Modifying source files..... 90
10.5 Example..... 90
10.6 Modifying the peripherals_<mcu>.c file..... 91
10.7 Removing unused files from the project..... 91

Chapter 11
Revision history

Chapter 1
Introduction

1.1 Introduction

The Kinetis bootloader is a configurable flash programming utility that operates over a serial connection on Kinetis MCUs. It enables quick and easy programming of Kinetis MCUs through the entire product life cycle, including application development, final product manufacturing, and beyond. The bootloader is delivered in two ways. The Kinetis bootloader is provided as full source code that is highly configurable. The bootloader is also preprogrammed by Freescale into ROM or flash on select Kinetis devices. Host-side command line and GUI tools are available to communicate with the bootloader. Users can utilize host tools to upload/download application code via the bootloader.

1.2 Terminology

target

The device running the bootloader firmware (aka the ROM).

host

The device sending commands to the target for execution.

source

The initiator of a communications sequence. For example, the sender of a command or data packet.

destination

Receiver of a command or data packet.

incoming

From host to target.

outgoing

From target to host.

1.3 Block diagram

This block diagram describes the overall structure of the Kinetis bootloader.

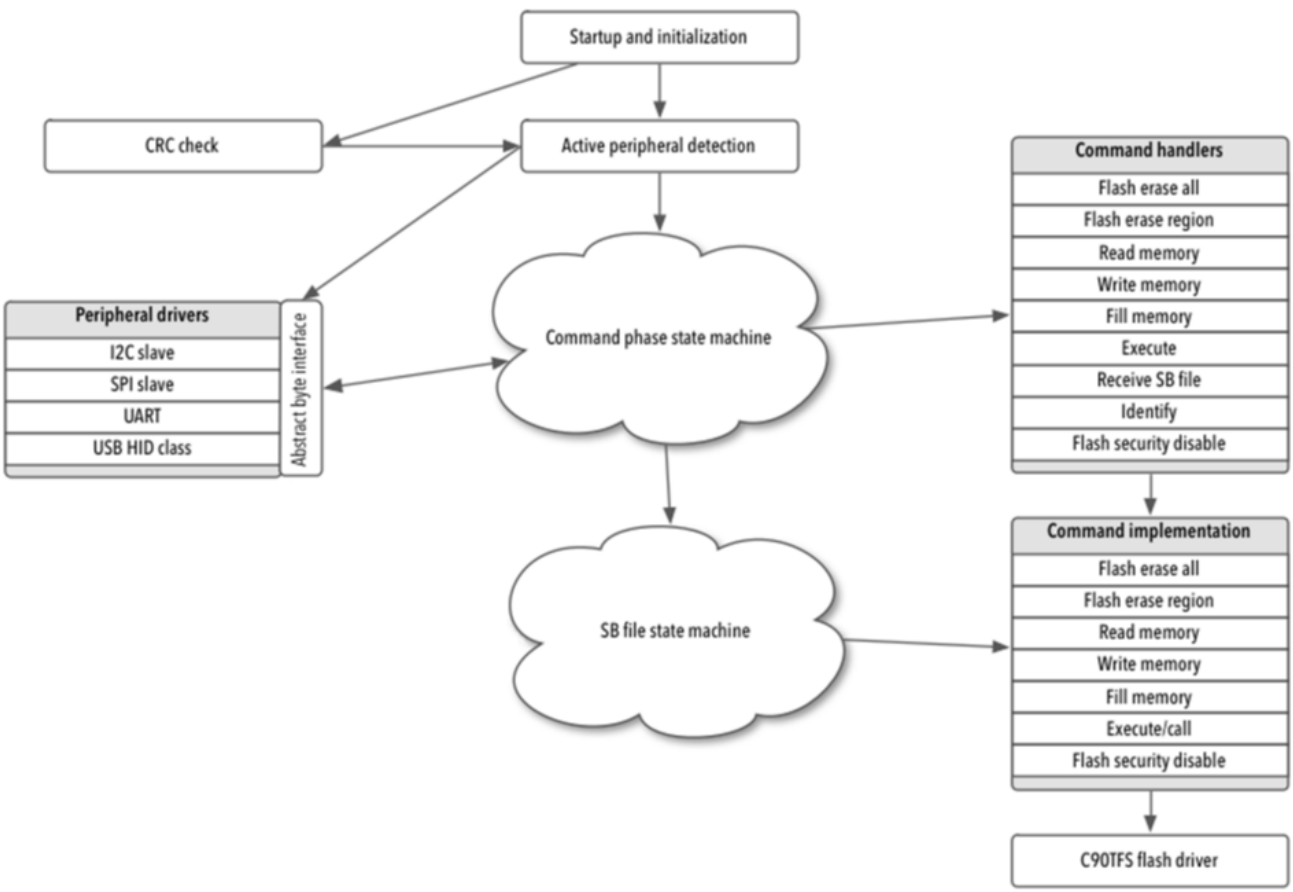

Figure 1-1. Block diagram

1.4 Features supported

Here are some of the features supported by the Kinetis bootloader:

- Supports UART, I2C, SPI and USB peripheral interfaces.
- Automatic detection of the active peripheral.
- Ability to disable any peripheral.
- UART peripheral implements autobaud.
- Common packet-based protocol for all peripherals.
- Packet error detection and retransmit.
- Flash-resident configuration options.
- Fully supports flash security, including ability to mass erase or unlock security via the backdoor key.
- Protection of RAM used by the bootloader while it is running.
- Provides command to read properties of the device, such as Flash and RAM size.
- Multiple options for executing the bootloader either at system start-up or under application control at runtime.

1.5 Components supported

Components for the bootloader firmware:

- Startup code (clocking, pinmux, etc.)
- Command phase state machine
- Command handlers
  - GenericResponse
  - FlashEraseAll
  - FlashEraseRegion
  - ReadMemory
  - ReadMemoryResponse
  - WriteMemory
  - FillMemory
  - FlashSecurityDisable
  - GetProperty
  - GetPropertyResponse
  - ReceiveSBFile
  - Execute
  - Call
  - Reset
  - SetProperty
  - FlashEraseAllUnsecure
  - FlashProgramOnce
  - FlashReadOnce
  - FlashReadOnceResponse

    - FlashReadResource
    - FlashReadResourceResponse

- SB file state machine
- Packet interface
    - Framing packetizer
    - Command/data packet processor

- Command implementation
    - Flash erase all
    - Flash erase region
    - Read memory
    - Write memory
    - Fill memory
    - Flash security disable
    - Get property
    - Recieve SB file
    - Execute
    - Call
    - Reset
    - Set property
    - Flash program once
    - Flash read once
    - Flash read resource

- Memory interface
    - Abstract interface
    - Flash Driver Interface
    - Low level flash driver

- Peripheral drivers
    - I2C slave
    - SPI slave
    - UART
        - Auto-baud detector
    - USB device HID class
        - USB controller driver
        - USB framework
        - USB HID class

- CRC check engine
    - CRC algorithm

Chapter 2
Functional description

2.1 Introduction

The following subsections describe the Kinetis bootloader functionality.

2.2 Memory map

See the Kinetis bootloader chapter of the reference manual of your particular SoC for the ROM and RAM memory map used by the bootloader.

2.3 The Kinetis Bootloader Configuration Area (BCA)

The Kinetis bootloader reads data from the Bootloader Configuration Area (BCA) to configure various features of the bootloader. The BCA resides in flash memory at offset 0x3C0 from the beginning of the user application, and provides all of the parameters needed to configure the Kinetis bootloader operation. For uninitialized flash, the Kinetis bootloader uses a predefined default configuration. A host application can use the Kinetis bootloader to program the BCA for use during subsequent initializations of the bootloader.

Table 2-1. Configuration Fields for the Kinetis bootloader

Offset	Size (bytes)	Configuration Field	Description
0x00 - 0x03	4	tag	Magic number to verify bootloader configuration is valid. Must be set to 'kcfg'.
0x04 - 0x07	4	crcStartAddress	Start address for application image CRC check. To generate the CRC, refer to the CRC chapter.

Table continues on the next page...

Table 2-1. Configuration Fields for the Kinetis bootloader (continued)

Offset	Size (bytes)	Configuration Field	Description
0x08 - 0x0B	4	crcByteCount	Byte count for application image CRC check.
0x0C - 0x0F	4	crcExpectedValue	Expected CRC value for application CRC check.
0x10	1	enabledPeripherals	Bitfield of peripherals to enable. bit 0 LPUART bit 1 I2C bit 2 SPI bit 4 USB
0x11	1	i2cSlaveAddress	If not 0xFF, used as the 7-bit I2C slave address.
0x12 - 0x13	2	peripheralDetectionTimeout	If not 0xFF, used as the timeout in milliseconds for active peripheral detection.
0x14 - 0x15	2	usbVid	Sets the USB Vendor ID reported by the device during enumeration.
0x16- 0x17	2	usbPid	Sets the USB Product ID reported by the device during enumeration.
0x18 - 0x1B	4	usbStringsPointer	Sets the USB Strings reported by the device during enumeration.
0x1C	1	clockFlags	See Table 2-3, clockFlags Configuration Field
0x1D	1	clockDivider	Divider to use for core and bus clocks when in high speed mode.
0x1E	1	-	Reserved.
0x1F	1	-	Reserved.
0x20	4	-	Reserved.
0x24	4	-	Reserved.

The first configuration field 'tag' is a tag value or magic number. The tag value must be set to 'kcfg' for the bootloader configuration data to be recognized as valid. If tag-field verification fails, the Kinetis bootloader acts as if the configuration data is not present. The tag value is treated as a character string, so bytes 0-3 must be set as shown in the table.

Table 2-2. tag Configuration Field

Offset	tag Byte Value
0	'k' (0x6B)
1	'c' (0x63)
2	'f' (0x66)
3	'g' (0x67)

The flags in the clockFlags configuration field are enabled if the corresponding bit is cleared (0).

Table 2-3. clockFlags Configuration Field

Bit	Flag	Description
0	HighSpeed	Enable high speed mode (i.e., 48 MHz).
1 - 7	-	Reserved.

2.4 Start-up process

These conditions force the hardware to start the Kinetis bootloader:

- The BOOTSRC_SEL field of FOPT register is set to either 0b11 or 0b10. This forces the ROM to run out of reset.
- The BOOTCFG0 pin is asserted. The pin must be configured as BOOTCFG0 by setting the BOOTPIN_OPT bit of FOPT to 0.
- A user applications running on flash or RAM calls into the Kinetis bootloader entry point address in ROM, to start Kinetis bootloader execution.

The BOOTSRC_SEL bits (FOPT register, FOPT [7:6]) determine the boot source. The FOPT register is located in the flash configuration field at address 0x40D in the flash memory array. For a complete list of options, see the Boot options section in the Reset and Boot chapter of the reference manual for your specific SoC. If BOOTSRC_SEL is set to 0b11 or 0b10, the device boots to ROM out of reset. Flash memory defaults to all 1s when erased, so a blank chip automatically boots to ROM.

The BOOTCFG0 pin is shared with the NMI pin, with NMI being the default usage. Regardless of whether the NMI pin is enabled or not, the NMI functionality is disabled if the ROM is executed out of reset, for as long as the ROM is running.

When the ROM is executed out of reset, vector fetches from the CPU are redirected to the ROM's vector table in ROM memory at offset 0x0. This ensures that any exceptions are handled by the ROM.

After the Kinetis bootloader has started, the following procedure starts bootloader operations:

1. The RCM_MR [FORCEROM] bits are set so the device reboots back into the ROM if/when the device is reset.
2. Initializes the bootloader's .data and .bss sections.
3. Reads bootloader configuration data from flash at offset 0x3C0. The configuration data is only used if the tag field is set to the expected 'kcfg' value. If the tag is

incorrect, then the configuration values are set to default, as if the data was all 0xFF bytes.

4. Clocks are configured.
5. Enabled peripherals are initialized.
6. The bootloader waits for communication to begin on a peripheral.
   - If detection times out, the bootloader jumps to the user application in flash.
   - If communication is detected, all inactive peripherals are shut down, and the command phase is entered.

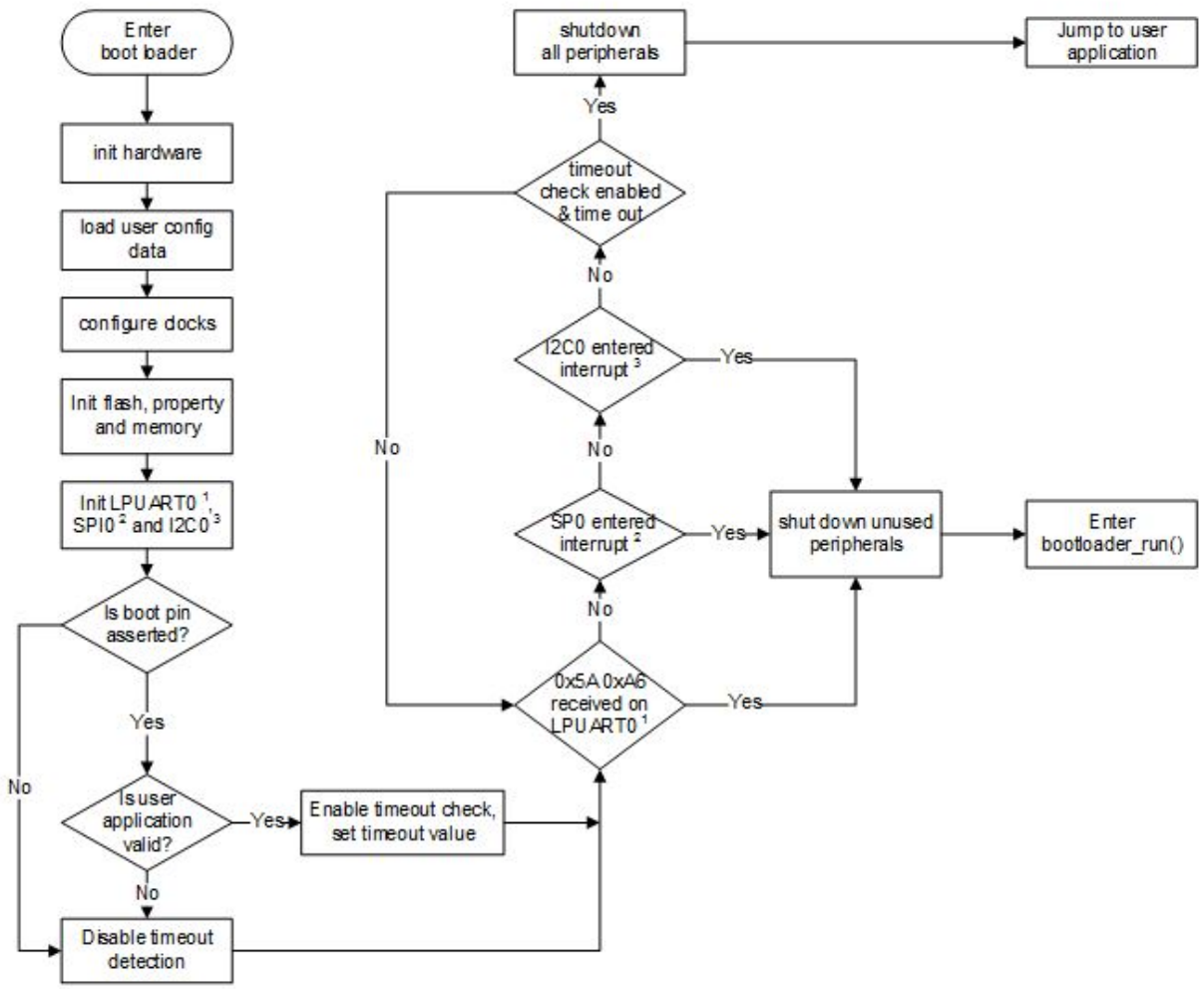

Figure 2-1. Kinetis bootloader start-up flowchart

2.5 Clock configuration

By default, the bootloader does not modify clocks. Kinetis bootloader in ROM uses the clock configuration of the chip out of reset, unless the clock configuration bits in the FOPT register are cleared, or if a USB peripheral is enabled.

- Alternate clock configurations are supported by setting fields in the bootloader configuration data.
- If the HighSpeed flag of the clockFlags configuration value is cleared or if a USB peripheral is enabled, the bootloader enables the internal 48 MHz reference clock. Higher speed clocks are used when available.
- In high-speed mode, the core and bus clock frequencies are determined by the clockDivider configuration value.
- The core clock divider is set directly from clockDivider, unless a USB peripheral is enabled. If a USB peripheral is enabled and clockDivider is greater than 2, clockDivider is reduced to 2 in order to keep the CPU clock above 20 MHz.
- The bus clock divider is set to 1, unless the resulting bus clock frequency would be greater than the maximum supported value. In this case, the bus clock divider is increased until the bus clock frequency is at or below the maximum.
- Note that the maximum baud rate of serial peripherals is related to the core and bus clock frequencies. To achieve the desired baud rates, high-speed mode should be enabled in BCA.

2.6 Bootloader entry point

The Kinetis bootloader provides a function (runBootloader) that a user application can call, to run the bootloader.

To get the address of the entry point, the user application reads the word containing the pointer to the bootloader API tree at offset 0x1C of the bootloader's vector table. The vector table is placed at the base of the bootloader's address range, which is 0x1C00_0000 for the ROM. Thus, the API tree pointer is at address 0x1C00_001C.

The bootloader API tree is a structure that contains pointers to other structures, which have the function and data addresses for the bootloader. The bootloader entry point is always the first word of the API tree.

The prototype of the entry point is:

```
void run_bootloader(void * arg);
```

The arg parameter is currently unused, and intended for future expansion. For example, passing options to the bootloader. To ensure future compatibility, a value of NULL should be passed for arg.

Example code to get the entry pointer address from the ROM and start the bootloader:

```
// Variables

uint32_t runBootloaderAddress;

void (*runBootloader)(void * arg);


// Read the function address from the ROM API tree.

runBootloaderAddress = **(uint32_t **)(0x1c00001c);

runBootloader = (void (*)(void * arg))runBootloaderAddress;


// Start the bootloader.

runBootloader(NULL);
```

NOTE

The user application must be executing at Supervisor (Privileged) level when calling the bootloader entry point.

Chapter 3
Kinetis bootloader protocol

3.1 Introduction

This section explains the general protocol for the packet transfers between the host and the Kinetis bootloader. The description includes the transfer of packets for different transactions, such as commands with no data phase and commands with incoming or outgoing data phase. The next section describes various packet types used in a transaction.

Each command sent from the host is replied to with a response command.

Commands may include an optional data phase.

- If the data phase is incoming (from the host to Kinetis bootloader), it is part of the original command.
- If the data phase is outgoing (from Kinetis bootloader to host), it is part of the response command.

3.2 Command with no data phase

NOTE

In these diagrams, the Ack sent in response to a Command or Data packet can arrive at any time before, during, or after the Command/Data packet has processed.

Command with no data phase

The protocol for a command with no data phase contains:

- Command packet (from host)
- Generic response command packet (to host)

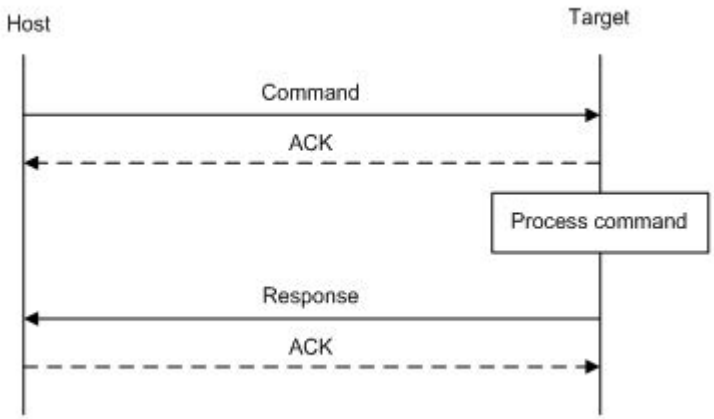

Figure 3-1. Command with no data phase

3.3 Command with incoming data phase

The protocol for a command with incoming data phase contains:

- Command packet (from host)(kCommandFlag_HasDataPhase set)
- Generic response command packet (to host)
- Incoming data packets (from host)
- Generic response command packet (to host)

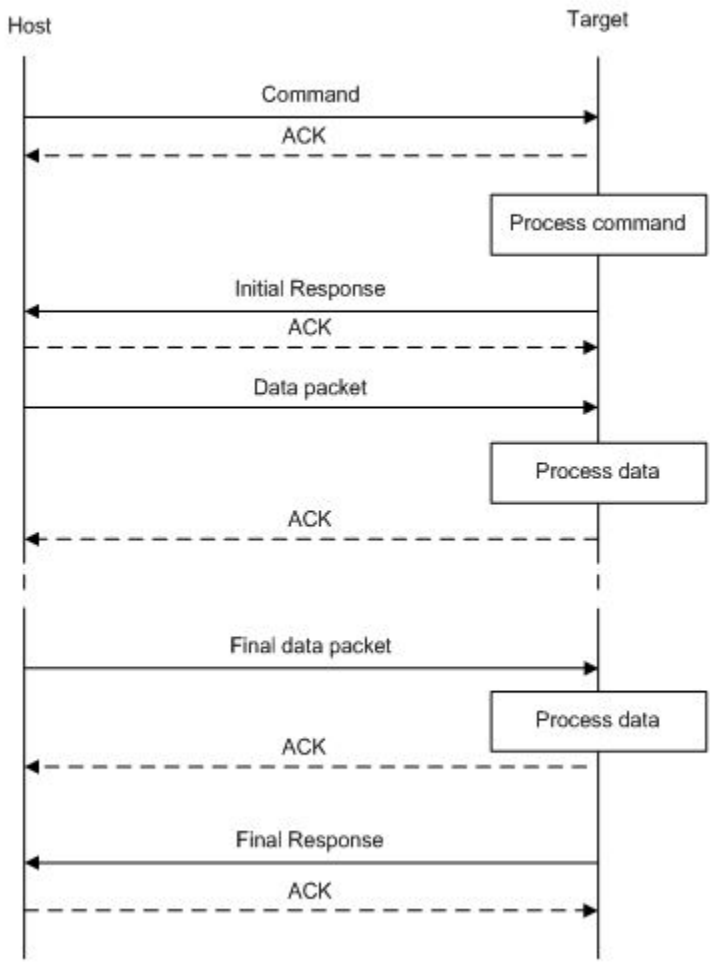

Figure 3-2. Command with incoming data phase

Notes

- The host may not send any further packets while it is waiting for the response to a command.
- The data phase is aborted if the Generic Response packet prior to the start of the data phase does not have a status of kStatus_Success.
- Data phases may be aborted by the receiving side by sending the final Generic Response early with a status of kStatus_AbortDataPhase. The host may abort the data phase early by sending a zero-length data packet.
- The final Generic Response packet sent after the data phase includes the status for the entire operation.

3.4 Command with outgoing data phase

The protocol for a command with an outgoing data phase contains:

- Command packet (from host)
- ReadMemory Response command packet (to host)(kCommandFlag_HasDataPhase set)
- Outgoing data packets (to host)
- Generic response command packet (to host)

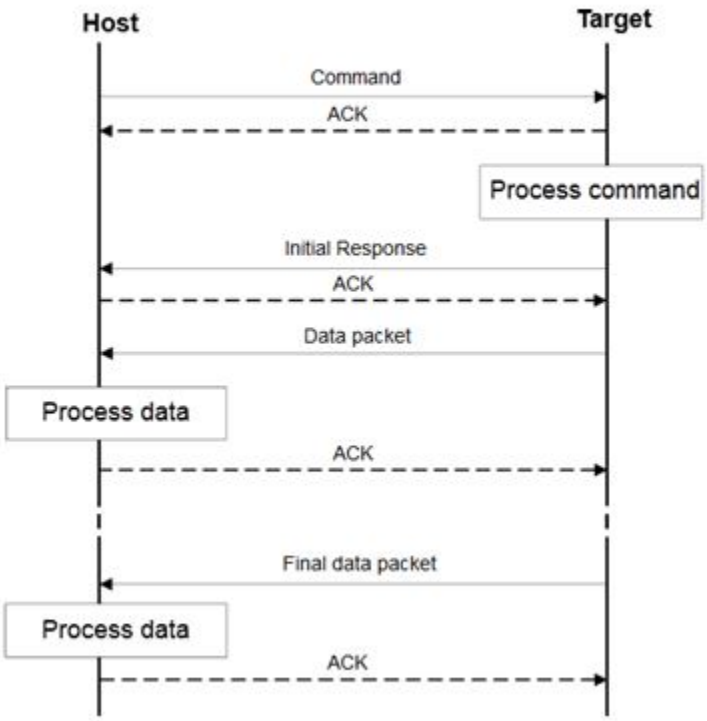

Figure 3-3. Command with outgoing data phase

Note

- The data phase is considered part of the response command for the outgoing data phase sequence.
- The host may not send any further packets while the host is waiting for the response to a command.
- The data phase is aborted if the ReadMemory Response command packet, prior to the start of the data phase, does not contain the kCommandFlag_HasDataPhase flag.

- Data phases may be aborted by the host sending the final Generic Response early with a status of kStatus_AbortDataPhase. The sending side may abort the data phase early by sending a zero-length data packet.
- The final Generic Response packet sent after the data phase includes the status for the entire operation.

Chapter 4
Bootloader packet types

4.1 Introduction

The Kinetis bootloader device works in slave mode. All data communication is initiated by a host, which is either a PC or an embedded host. The Kinetis bootloader device is the target, which receives a command or data packet. All data communication between host and target is packetized.

NOTE

The term "target" refers to the "Kinetis bootloader device".

There are 6 types of packets used:

- Ping packet
- Ping Response packet
- Framing packet
- Command packet
- Data packet
- Response packet

All fields in the packets are in little-endian byte order.

4.2 Ping packet

The Ping packet is the first packet sent from a host to the target to establish a connection on selected peripheral in order to run autobaud. The Ping packet can be sent from host to target at any time that the target is expecting a command packet. If the selected peripheral is UART, a ping packet must be sent before any other communications. For other serial peripherals it is optional, but is recommended in order to determine the serial protocol version.

In response to a Ping packet, the target sends a Ping Response packet, discussed in later sections.

Table 4-1. Ping Packet Format

Byte #	Value	Name
0	0x5A	start byte
1	0xA6	ping

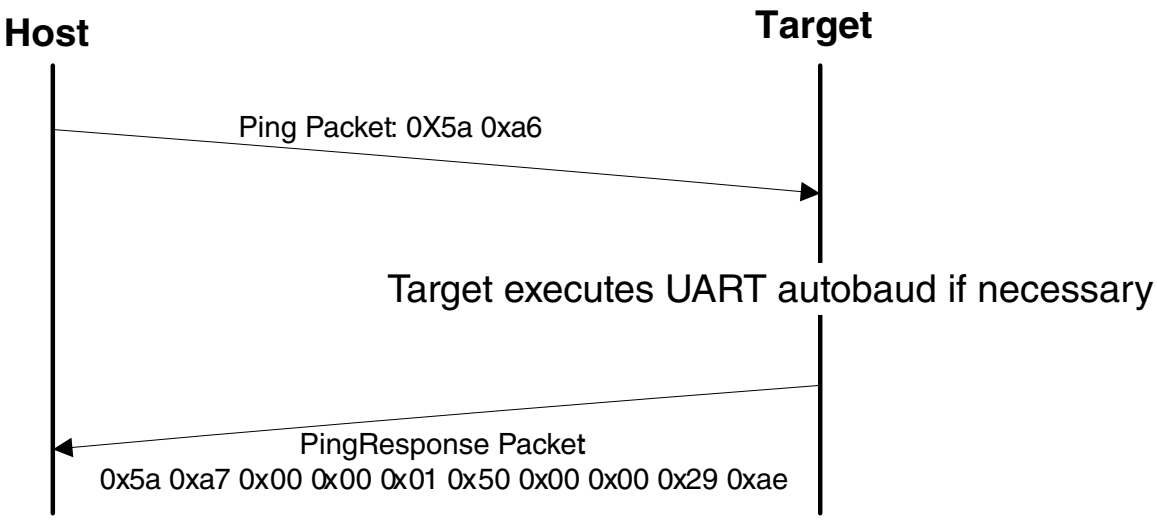

Figure 4-1. Ping Packet Protocol Sequence

4.3 Ping response packet

The target sends a Ping Response packet back to the host after receiving a Ping packet. If communication is over a UART peripheral, the target uses the incoming Ping packet to determine the baud rate before replying with the Ping Response packet. Once the Ping Response packet is received by the host, the connection is established, and the host starts sending commands to the target.

Table 4-2. Ping Response packet format

Byte #	Value	Parameter
0	0x5A	start byte
1	0xA7	Ping response code
2		Protocol bugfix
3		Protocol minor
4		Protocol major
5		Protocol name = 'P' (0x50)
6		Options low
7		Options high

Table continues on the next page...

Table 4-2. Ping Response packet format (continued)

Byte #	Value	Parameter
8		CRC16 low
9		CRC16 high

The Ping Response packet can be sent from host to target any time the target expects a command packet. For the UART peripheral, it must be sent by host when a connection is first established, in order to run autobaud. For other serial peripherals it is optional, but recommended to determine the serial protocol version. The version number is in the same format at the bootloader version number returned by the GetProperty command.

4.4 Framing packet

The framing packet is used for flow control and error detection for the communications links that do not have such features built-in. The framing packet structure sits between the link layer and command layer. It wraps command and data packets as well.

Every framing packet containing data sent in one direction results in a synchronizing response framing packet in the opposite direction.

The framing packet described in this section is used for serial peripherals including the UART, I2C, and SPI. The USB HID peripheral does not use framing packets. Instead, the packetization inherent in the USB protocol itself is used.

Table 4-3. Framing Packet Format

Byte #	Value	Parameter	
0	0x5A	start byte	
1		packetType	
2		length_low	Length is a 16-bit field that specifies the entire command or data packet size in bytes.
3		length_high	
4		crc16_low	This is a 16-bit field. The CRC16 value covers entire framing packet, including the start byte and command or data packets, but does not include the CRC bytes. See the CRC16 algorithm after this table.
5		crc16_high	
6 . . .n		Command or Data packet payload	

CRC16 algorithm:

```
uint16_t crc16_update(const uint8_t * src, uint32_t lengthInBytes)

{
```

```
    uint32_t crc = 0;
    uint32_t j;
    for (j=0; j < lengthInBytes; ++j)
    {
        uint32_t i;
        uint32_t byte = src[j];
        crc ^= byte << 8;
        for (i = 0; i < 8; ++i)
        {
            uint32_t temp = crc << 1;
            if (crc & 0x8000)
            {
                temp ^= 0x1021;
            }
            crc = temp;
        }
    }
    return crc;
}
```

A special framing packet that contains only a start byte and a packet type is used for synchronization between the host and target.

Table 4-4. Special Framing Packet Format

Byte #	Value	Parameter
0	0x5A	start byte
1	0xA*n*	packetType

The Packet Type field specifies the type of the packet from one of the defined types (below):

Table 4-5. packetType Field

packetType	Name	Description
0xA1	kFramingPacketType_Ack	The previous packet was received successfully; the sending of more packets is allowed.
0xA2	kFramingPacketType_Nak	The previous packet was corrupted and must be re-sent.

Table continues on the next page...

Table 4-5. packetType Field (continued)

packetType	Name	Description
0xA3	kFramingPacketType_AckAbort	Data phase is being aborted.
0xA4	kFramingPacketType_Command	The framing packet contains a command packet payload.
0xA5	kFramingPacketType_Data	The framing packet contains a data packet payload.
0xA6	kFramingPacketType_Ping	Sent to verify the other side is alive. Also used for UART autobaud.
0xA7	kFramingPacketType_PingResponse	A response to Ping; contains the framing protocol version number and options.

4.5 CRC16 algorithm

This section provides the CRC16 algorithm.

```
uint16_t crc16_update(const uint8_t * src, uint32_t lengthInBytes)
{
    uint32_t crc = 0;
    uint32_t j;
    for (j=0; j < lengthInBytes; ++j)
    {
        uint32_t i;
        uint32_t byte = src[j];
        crc ^= byte << 8;
        for (i = 0; i < 8; ++i)
        {
            uint32_t temp = crc << 1;
            if (crc & 0x8000)
            {
                temp ^= 0x1021;
            }
            crc = temp;
        }
    }
    return crc;
```

}

4.6 Command packet

The command packet carries a 32-bit command header and a list of 32-bit parameters.

Table 4-6. Command Packet Format

Command Packet Format (32 bytes)										
Command Header (4 bytes)				28 bytes for Parameters (Max 7 parameters)						
Tag	Flags	Rsvd	Param Count	Param1 (32-bit)	Param2 (32-bit)	Param3 (32-bit)	Param4 (32-bit)	Param5 (32-bit)	Param6 (32-bit)	Param7 (32-bit)
byte 0	byte 1	byte 2	byte 3							

Table 4-7. Command Header Format

Byte #	Command Header Field	
0	Command or Response tag	The command header is 4 bytes long, with these fields.
1	Flags	
2	Reserved. Should be 0x00.	
3	ParameterCount	

The header is followed by 32-bit parameters up to the value of the ParameterCount field specified in the header. Because a command packet is 32 bytes long, only 7 parameters can fit into the command packet.

Command packets are also used by the target to send responses back to the host. As mentioned earlier, command packets and data packets are embedded into framing packets for all of the transfers.

Table 4-8. Command Tags

Command Tag	Name	
0x01	FlashEraseAll	The command tag specifies one of the
0x01	FlashEraseAll	The command tag specifies one of the commands supported by the Kinetis bootloader. The valid command tags for the Kinetis bootloader are listed here.
0x02	FlashEraseRegion	
0x03	ReadMemory	
0x04	WriteMemory	
0x05	FillMemory	
0x06	FlashSecurityDisable Reserved	
0x07	GetProperty	
0x08	ReceiveSbFile	
0x09	Execute	

Table continues on the next page...

Table 4-8. Command Tags (continued)

Command Tag	Name	
0x10	FlashReadResource	
0x11	Reserved	
0x0A	Call	
0x0B	Reset	
0x0C	SetProperty	
0x0D	FlashEraseAllUnsecure	
0x0D	Reserved	
0x0E	FlashProgramOnce	
0x0F	FlashReadOnce	

Table 4-9. Response Tags

Response Tag	Name	
0xA0	GenericResponse	The response tag specifies one of the responses
0xA0	GenericResponse	The response tag specifies one of the responses the Kinetis bootloader (target) returns to the host. The valid response tags are listed here.
0xA7	GetPropertyResponse (used for sending responses to GetProperty command only)	
0xA3	ReadMemoryResponse (used for sending responses to ReadMemory command only)	
0xAF	FlashReadOnceResponse (used for sending responses to FlashReadOnce command only)	
0xB0	FlashReadResourceResponse (used for sending responses to FlashReadResource command only)	

Flags: Each command packet contains a Flag byte. Only bit 0 of the flag byte is used. If bit 0 of the flag byte is set to 1, then data packets follow in the command sequence. The number of bytes that are transferred in the data phase is determined by a command-specific parameter in the parameters array.

ParameterCount: The number of parameters included in the command packet.

Parameters: The parameters are word-length (32 bits). With the default maximum packet size of 32 bytes, a command packet can contain up to 7 parameters.

4.7 Response packet

The responses are carried using the same command packet format wrapped with framing packet data. Types of responses include:

- GenericResponse

- GetPropertyResponse
- ReadMemoryResponse
- FlashReadOnceResponse
- FlashReadResourceResponse

GenericResponse: After the Kinetis bootloader has processed a command, the bootloader sends a generic response with status and command tag information to the host. The generic response is the last packet in the command protocol sequence. The generic response packet contains the framing packet data and the command packet data (with generic response tag = 0xA0) and a list of parameters (defined in the next section). The parameter count field in the header is always set to 2, for status code and command tag parameters.

Table 4-10. GenericResponse Parameters

Byte #	Parameter	Descripton
0 - 3	Status code	The Status codes are errors encountered during the execution of a command by the target. If a command succeeds, then a kStatus_Success code is returned. Table 1, Kinetis BootloaderFlashloader Status Error Codes, lists the status codes returned to the host by the Kinetis bootloader.
4 - 7	Command tag	The Command tag parameter identifies the response to the command sent by the host.

GetPropertyResponse: The GetPropertyResponse packet is sent by the target in response to the host query that uses the GetProperty command. The GetPropertyResponse packet contains the framing packet data and the command packet data, with the command/response tag set to a GetPropertyResponse tag value (0xA7).

The parameter count field in the header is set to greater than 1, to always include the status code and one or many property values.

Table 4-11. GetPropertyResponse Parameters

Byte #	Value	Parameter
0 - 3		Status code
4 - 7		Property value
. . .		. . .
		Can be up to maximum 6 property values, limited to the size of the 32-bit command packet and property type.

ReadMemoryResponse: The ReadMemoryResponse packet is sent by the target in response to the host sending a ReadMemory command. The ReadMemoryResponse packet contains the framing packet data and the command packet data, with the command/response tag set to a ReadMemoryResponse tag value (0xA3), the flags field set to kCommandFlag_HasDataPhase (1).

The parameter count set to 2 for the status code and the data byte count parameters shown below.

Table 4-12. ReadMemoryResponse Parameters

Byte #	Parameter	Descripton
0 - 3	Status code	The status of the associated Read Memory command.
4 - 7	Data byte count	The number of bytes sent in the data phase.

FlashReadOnceResponse: The FlashReadOnceResponse packet is sent by the target in response to the host sending a FlashReadOnce command. The FlashReadOnceResponse packet contains the framing packet data and the command packet data, with the command/response tag set to a FlashReadOnceResponse tag value (0xAF), and the flags field set to 0. The parameter count is set to 2 plus *the number of words* requested to be read in the FlashReadOnceCommand.

Table 4-13. FlashReadOnceResponse Parameters

Byte #	Value	Parameter
0 – 3		Status Code
4 – 7		Byte count to read
…		…
		Can be up to 20 bytes of requested read data.

The FlashReadResourceResponse packet is sent by the target in response to the host sending a FlashReadResource command. The FlashReadResourceResponse packet contains the framing packet data and command packet data, with the command/response tag set to a FlashReadResourceResponse tag value (0xB0), and the flags field set to kCommandFlag_HasDataPhase (1).

Table 4-14. FlashReadResourceResponse Parameters

Byte #	Value	Parameter
0 – 3		Status Code
4 – 7		Data byte count

Chapter 5
Kinetis bootloader command API

5.1 Introduction

All Kinetis bootloader command APIs follows the command packet format wrapped by the framing packet as explained in previous sections.

For a list of commands supported by Kinetis bootloader refer to Section 5.

For a list of status codes returned by Kinetis bootloader refer to Appendix A.

5.2 GetProperty command

The GetProperty command is used to query the bootloader about various properties and settings. Each supported property has a unique 32-bit tag associated with it. The tag occupies the first parameter of the command packet. The target returns a GetPropertyResponse packet with the property values for the property identified with the tag in the GetProperty command.

Properties are the defined units of data that can be accessed with the GetProperty or SetProperty commands. Properties may be read-only or read-write. All read-write properties are 32-bit integers, so they can easily be carried in a command parameter.

For a list of properties and their associated 32-bit property tags supported by Kinetis bootloader, refer to Appendix B.

The 32-bit property tag is the only parameter required for GetProperty command.

Table 5-1. Parameters for GetProperty Command

Byte #	Command
0 - 3	Property tag

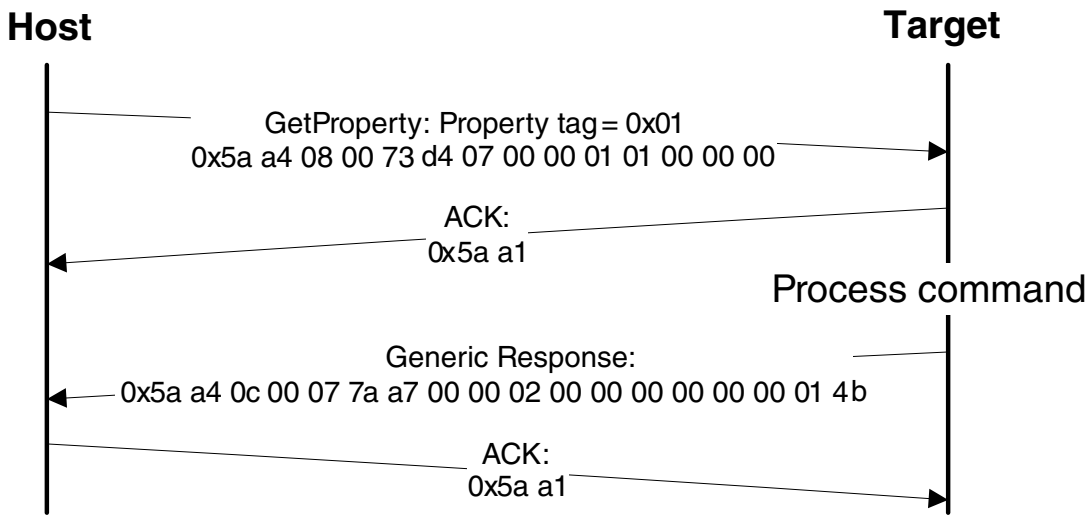

Figure 5-1. Protocol Sequence for GetProperty Command

Table 5-2. GetProperty Command Packet Format (Example)

GetProperty	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x08 0x00
	crc16	0x73 0xD4
Command packet	commandTag	0x07 – GetProperty
	flags	0x00
	reserved	0x00
	parameterCount	0x01
	propertyTag	0x00000001 - CurrentVersion

The GetProperty command has no data phase.

Response: In response to a GetProperty command, the target sends a GetPropertyResponse packet with the response tag set to 0xA7. The parameter count indicates the number of parameters sent for the property values, with the first parameter showing status code 0, followed by the property value(s). The next table shows an example of a GetPropertyResponse packet.

Table 5-3. GetProperty Response Packet Format (Example)

GetPropertyResponse	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x0c 0x00 (12 bytes)

Table continues on the next page...

Table 5-3. GetProperty Response Packet Format (Example) (continued)

GetPropertyResponse	Parameter	Value
	crc16	0x07 0x7a
Command packet	responseTag	0xA7
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	status	0x00000000
	propertyValue	0x0000014b - CurrentVersion

5.3 SetProperty command

The SetProperty command is used to change or alter the values of the properties or options of the bootloader. The command accepts the same property tags used with the GetProperty command. However, only some properties are writable--see Appendix B. If an attempt to write a read-only property is made, an error is returned indicating the property is read-only and cannot be changed.

The property tag and the new value to set are the two parameters required for the SetProperty command.

Table 5-4. Parameters for SetProperty Command

Byte #	Command
0 - 3	Property tag
4 - 7	Property value

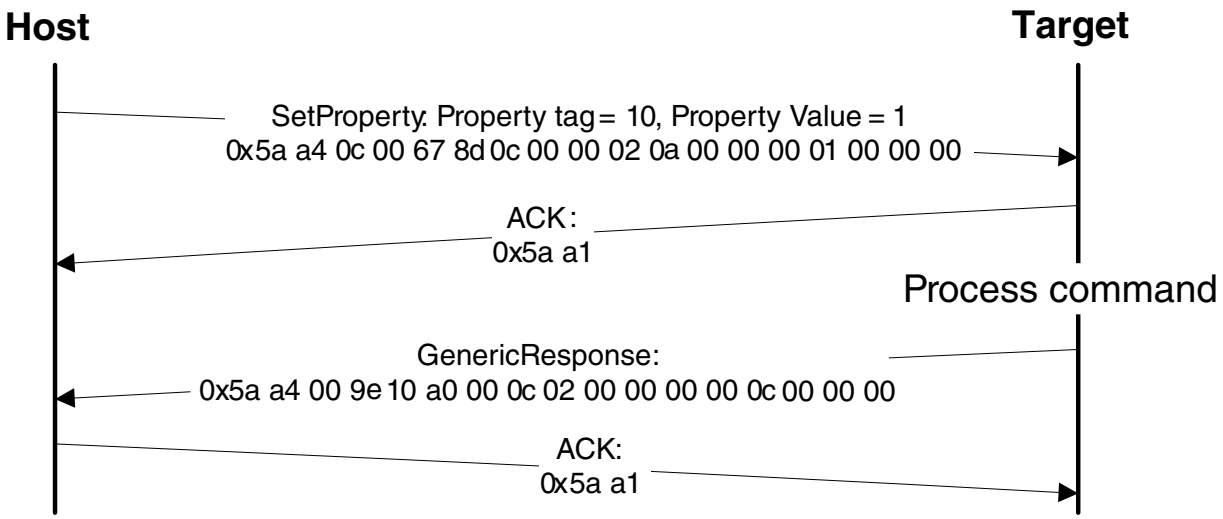

Figure 5-2. Protocol Sequence for SetProperty Command

Table 5-5. SetProperty Command Packet Format (Example)

SetProperty	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x0C 0x00
	crc16	0x67 0x8D
Command packet	commandTag	0x0C – SetProperty with property tag 10
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	propertyTag	0x0000000A - VerifyWrites
	propertyValue	0x00000001

The SetProperty command has no data phase.

Response: The target returns a GenericResponse packet with one of following status codes:

Table 5-6. SetProperty Response Status Codes

Status Code
kStatus_Success
kStatus_ReadOnly
kStatus_UnknownProperty
kStatus_InvalidArgument

5.4 FlashEraseAll command

The FlashEraseAll command performs an erase of the entire flash memory. If any flash regions are protected, then the FlashEraseAll command fails and returns an error status code. Executing the FlashEraseAll command releases flash security if it (flash security) was enabled, by setting the FTFA_FSEC register. However, the FSEC field of the flash configuration field is erased, so unless it is reprogrammed, the flash security is re-enabled after the next system reset. The Command tag for FlashEraseAll command is 0x01 set in the commandTag field of the command packet.

The FlashEraseAll command requires no parameters.

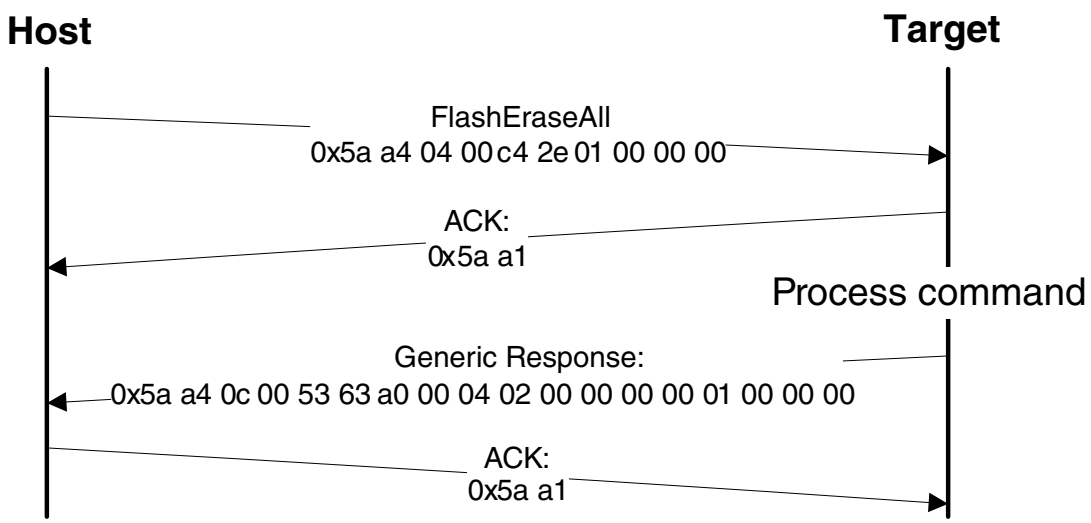

Figure 5-3. Protocol Sequence for FlashEraseAll Command

Table 5-7. FlashEraseAll Command Packet Format (Example)

FlashEraseAll	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x04 0x00
	crc16	0xC4 0x2E
Command packet	commandTag	0x01 - FlashEraseAll
	flags	0x00
	reserved	0x00
	parameterCount	0x00

The FlashEraseAll command has no data phase.

Response: The target returns a GenericResponse packet with status code either set to kStatus_Success for successful execution of the command, or set to an appropriate error status code.

5.5 FlashEraseRegion command

The FlashEraseRegion command performs an erase of one or more sectors of the flash memory.

The start address and number of bytes are the 2 parameters required for the FlashEraseRegion command. The start and byte count parameters must be 4-byte aligned ([1:0] = 00), or the FlashEraseRegion command fails and returns kStatus_FlashAlignmentError(101). If the region specified does not fit in the flash memory space, the FlashEraseRegion command fails and returns kStatus_FlashAddressError(102). If any part of the region specified is protected, the FlashEraseRegion command fails and returns kStatus_MemoryRangeInvalid(10200).

Table 5-8. Parameters for FlashEraseRegion Command

Byte #	Parameter
0 - 3	Start address
4 - 7	Byte count

The FlashEraseRegion command has no data phase.

Response: The target returns a GenericResponse packet with one of following error status codes.

Table 5-9. FlashEraseRegion Response Status Codes

Status Code
kStatus_Success (0)
kStatus_MemoryRangeInvalid (10200)
kStatus_FlashAlignmentError (101)
kStatus_FlashAddressError (102)
kStatus_FlashAccessError (103)
kStatus_FlashProtectionViolation (104)
kStatus_FlashCommandFailure (105)

5.6 FlashEraseAllUnsecure command

The FlashEraseAllUnsecure command performs a mass erase of the flash memory, including protected sectors. Flash security is immediately disabled if it (flash security) was enabled, and the FSEC byte in the flash configuration field at address 0x40C is programmed to 0xFE. However, if the mass erase enable option in the FSEC field is disabled, then the FlashEraseAllUnsecure command fails.

The FlashEraseAllUnsecure command requires no parameters.

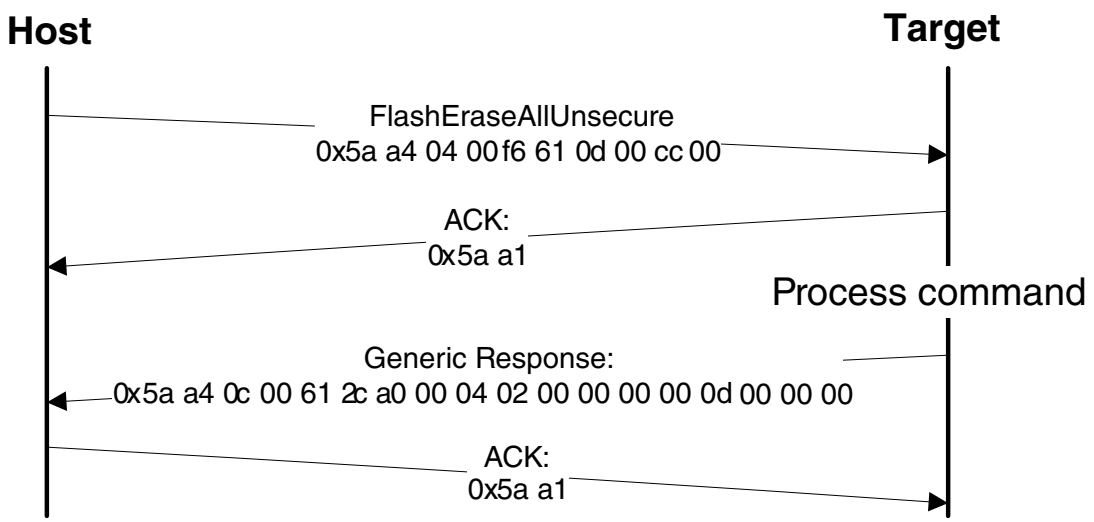

Figure 5-4. Protocol Sequence for FlashEraseAll Command

Table 5-10. FlashEraseAllUnsecure Command Packet Format (Example)

FlashEraseAllUnsecure	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x04 0x00
	crc16	0xF6 0x61
Command packet	commandTag	0x0D - FlashEraseAllUnsecure
	flags	0x00
	reserved	0x00
	parameterCount	0x00

The FlashEraseAllUnsecure command has no data phase.

Response: The target returns a GenericResponse packet with status code either set to kStatus_Success for successful execution of the command, or set to an appropriate error status code.

5.7 ReadMemory command

The ReadMemory command returns the contents of memory at the given address, for a specified number of bytes. This command can read any region of memory accessible by the CPU and not protected by security.

The start address and number of bytes are the two parameters required for ReadMemory command.

Table 5-11. Parameters for read memory command

Byte	Parameter	Description
0-3	Start address	Start address of memory to read from
4-7	Byte count	Number of bytes to read and return to caller

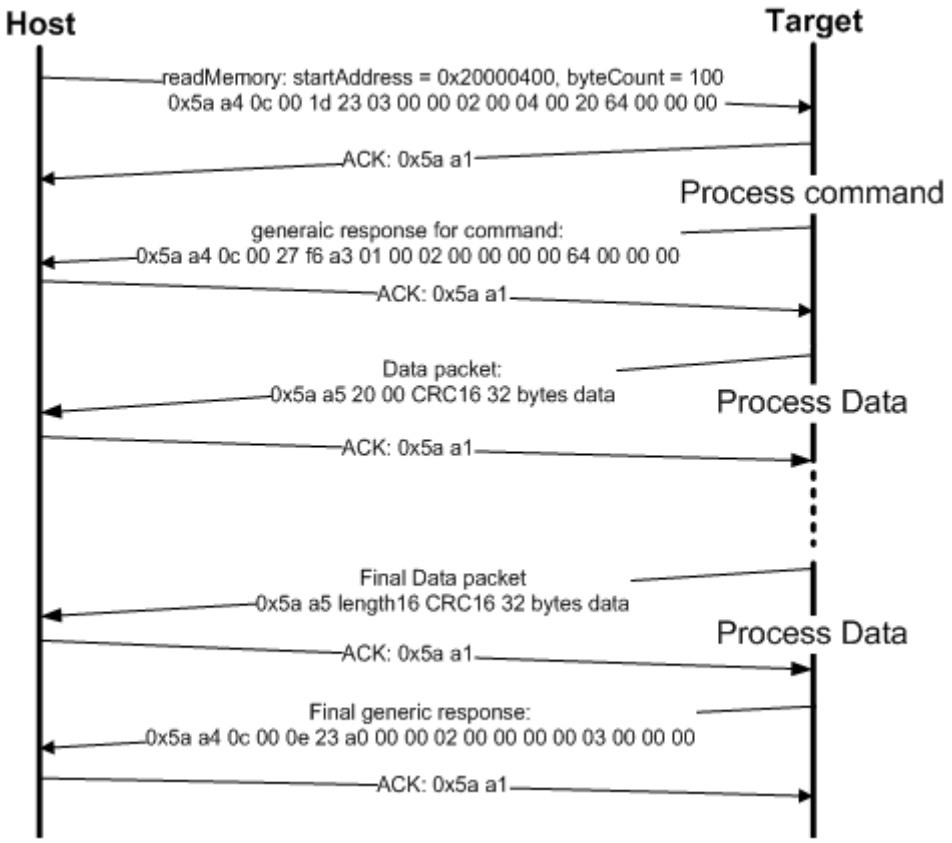

Figure 5-5. Command sequence for read memory

ReadMemory	Parameter	Value
Framing packet	Start byte	0x5A0xA4,
	packetType	kFramingPacketType_Command
	length	0x0C 0x00
	crc16	0x1D 0x23
Command packet	commandTag	0x03 - readMemory
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	startAddress	0x20000400
	byteCount	0x00000064

Data Phase: The ReadMemory command has a data phase. Since the target works in slave mode, the host need pull data packets until the number of bytes of data specified in the byteCount parameter of ReadMemory command are received by host.

Response: The target returns a GenericResponse packet with a status code either set to kStatus_Success upon successful execution of the command, or set to an appropriate error status code.

5.8 WriteMemory command

The WriteMemory command writes data provided in the data phase to a specified range of bytes in memory (flash or RAM). However, if flash protection is enabled, then writes to protected sectors fail.

Special care must be taken when writing to flash.

- First, any flash sector written to must have been previously erased with a FlashEraseAll, FlashEraseRegion, or FlashEraseAllUnsecure command.
- First, any flash sector written to must have been previously erased with a FlashEraseAll or FlashEraseRegion command.
- Writing to flash requires the start address to be 4-byte aligned ([1:0] = 00).
- The byte count is rounded up to a multiple of 4, and trailing bytes are filled with the flash erase pattern (0xff).
- If the VerifyWrites property is set to true, then writes to flash also performs a flash verify program operation.

When writing to RAM, the start address does not need to be aligned, and the data is not padded.

The start address and number of bytes are the 2 parameters required for WriteMemory command.

Table 5-13. Parameters for WriteMemory Command

Byte #	Command
0 - 3	Start address
4 - 7	Byte count

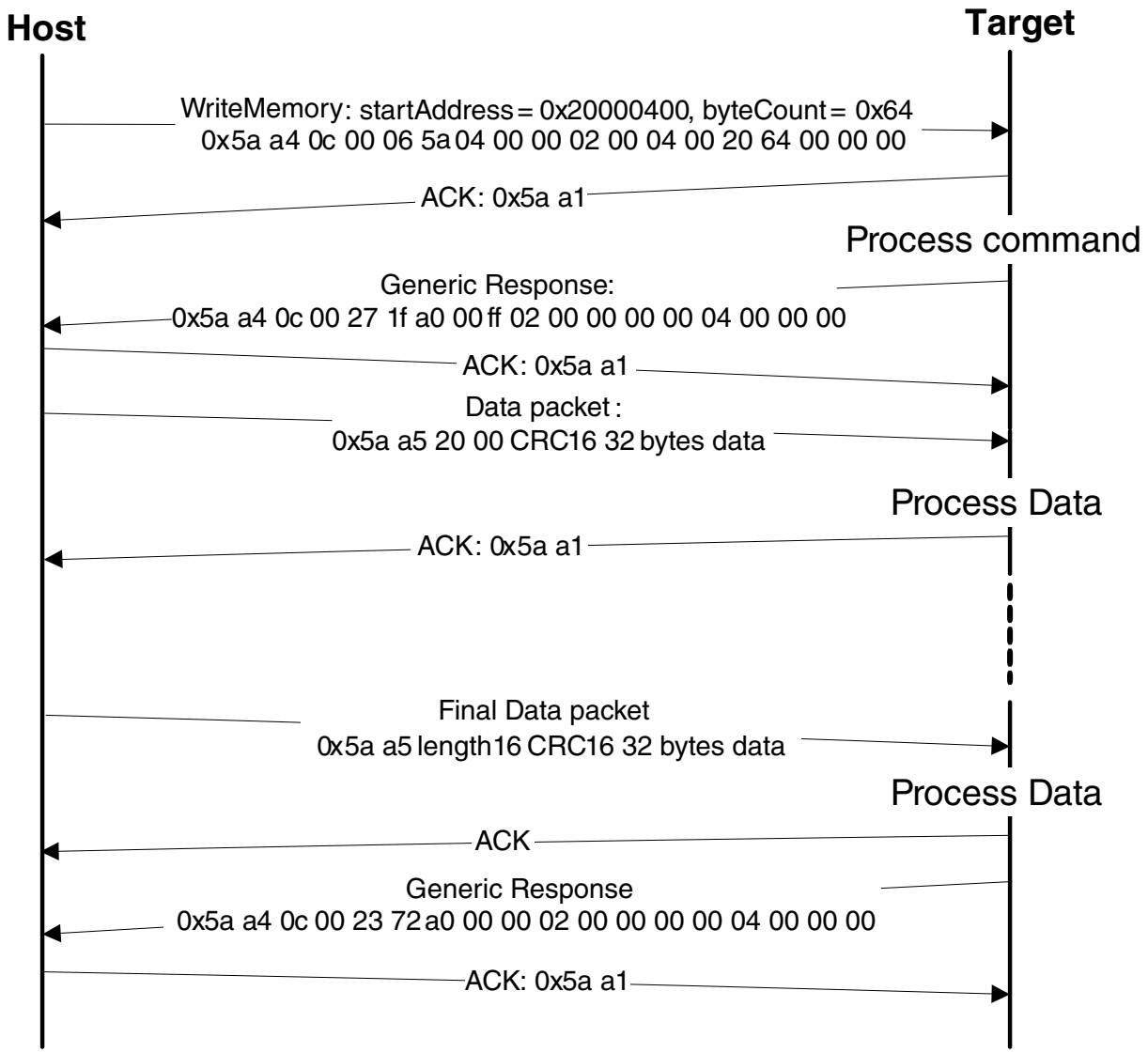

Figure 5-6. Protocol Sequence for WriteMemory Command

Table 5-14. WriteMemory Command Packet Format (Example)

WriteMemory	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x0C 0x00
	crc16	0x06 0x5A
Command packet	commandTag	0x04 - writeMemory
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	startAddress	0x20000400
	byteCount	0x00000064

Data Phase: The WriteMemory command has a data phase; the host sends data packets until the number of bytes of data specified in the byteCount parameter of the WriteMemory command are received by the target.

Response: The target returns a GenericResponse packet with a status code set to kStatus_Success upon successful execution of the command, or to an appropriate error status code.

5.9 FillMemory command

The FillMemory command fills a range of bytes in memory with a data pattern. It follows the same rules as the WriteMemory command. The difference between FillMemory and WriteMemory is that a data pattern is included in FillMemory command parameter, and there is no data phase for the FillMemory command, while WriteMemory does have a data phase.

Table 5-15. Parameters for FillMemory Command

Byte #	Command
0 - 3	Start address of memory to fill
4 - 7	Number of bytes to write with the pattern • The start address should be 32-bit aligned. • The number of bytes must be evenly divisible by 4. (Note: for a part that uses FTFE flash, the start address should be 64-bit aligned, and the number of bytes must be evenly divisible by 8).
8 - 11	32-bit pattern

- To fill with a byte pattern (8-bit), the byte must be replicated 4 times in the 32-bit pattern.
- To fill with a short pattern (16-bit), the short value must be replicated 2 times in the 32-bit pattern.

For example, to fill a byte value with 0xFE, the word pattern would be 0xFEFEFEFE; to fill a short value 0x5AFE, the word pattern would be 0x5AFE5AFE.

Special care must be taken when writing to flash.

- First, any flash sector written to must have been previously erased with a FlashEraseAll, FlashEraseRegion, or FlashEraseAllUnsecure command.
- First, any flash sector written to must have been previously erased with a FlashEraseAll or FlashEraseRegion command.

- Writing to flash requires the start address to be 4-byte aligned ([1:0] = 00).
- If the VerifyWrites property is set to true, then writes to flash also performs a flash verify program operation.

When writing to RAM, the start address does not need to be aligned, and the data is not padded.

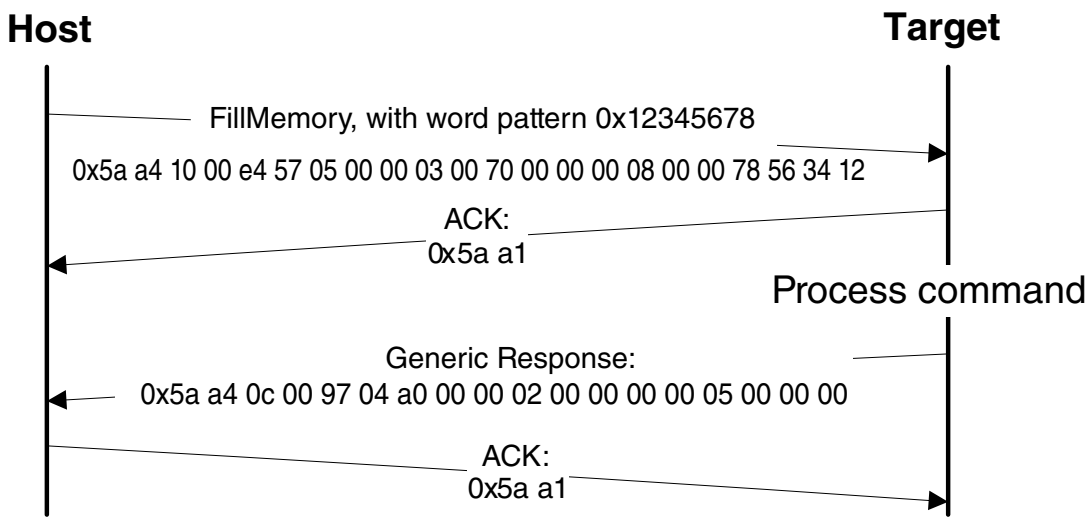

Figure 5-7. Protocol Sequence for FillMemory Command

Table 5-16. FillMemory Command Packet Format (Example)

FillMemory	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x10 0x00
	crc16	0xE4 0x57
Command packet	commandTag	0x05 – FillMemory
	flags	0x00
	Reserved	0x00
	parameterCount	0x03
	startAddress	0x00007000
	byteCount	0x00000800
	patternWord	0x12345678

The FillMemory command has no data phase.

Response: upon successful execution of the command, the target (Kinetis Flashloader) returns a GenericResponse packet with a status code set to kStatus_Success, or to an appropriate error status code.

5.10 FlashSecurityDisable command

The FlashSecurityDisable command performs the flash security disable operation, by comparing the 8-byte backdoor key (provided in the command) against the backdoor key stored in the flash configuration field (at address 0x400 in the flash).

The backdoor low and high words are the only parameters required for FlashSecurityDisable command.

Table 5-17. Parameters for FlashSecurityDisable Command

Byte #	Command
0 - 3	Backdoor key low word
4 - 7	Backdoor key high word

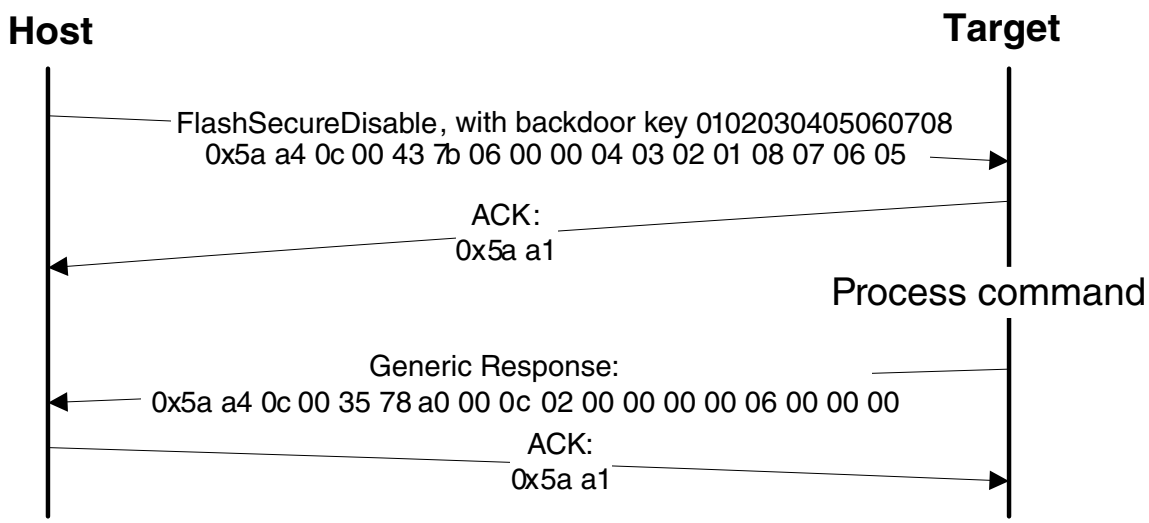

Figure 5-8. Protocol Sequence for FlashSecurityDisable Command

Table 5-18. FlashSecurityDisable Command Packet Format (Example)

FlashSecurityDisable	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x0C 0x00
	crc16	0x43 0x7B
Command packet	commandTag	0x06 - FlashSecurityDisable
	flags	0x00
	reserved	0x00

Table continues on the next page...

Table 5-18. FlashSecurityDisable Command Packet Format (Example) (continued)

FlashSecurityDisable	Parameter	Value
	parameterCount	0x02
	Backdoorkey_low	0x04 0x03 0x02 0x01
	Backdoorkey_high	0x08 0x07 0x06 0x05

The FlashSecurityDisable command has no data phase.

Response: The target returns a GenericResponse packet with a status code either set to kStatus_Success upon successful execution of the command, or set to an appropriate error status code.

5.11 ReceiveSBFile command

The Receive SB File command (ReceiveSbFile) starts the transfer of an SB file to the target. The command only specifies the size in bytes of the SB file that is sent in the data phase. The SB file is processed as it is received by the bootloader.

Table 5-19. Parameters for Receive SB File Command

Byte #	Command
0 - 3	Byte count

Data Phase: The Receive SB file command has a data phase; the host sends data packets until the number of bytes of data specified in the byteCount parameter of the Receive SB File command are received by the target.

Response: The target returns a GenericResponse packet with a status code set to the kStatus_Success upon successful execution of the command, or set to an appropriate error code.

5.12 Execute command

The execute command results in the bootloader setting the program counter to the code at the provided jump address, R0 to the provided argument, and a Stack pointer to the provided stack pointer address. Prior to the jump, the system is returned to the reset state.

The Jump address, function argument pointer, and stack pointer are the parameters required for the Execute command.

Table 5-20. Parameters for Execute Command

Byte #	Command
0 - 3	Jump address
4 - 7	Argument word
8 - 11	Stack pointer address

The Execute command has no data phase.

Response: Before executing the Execute command, the target validates the parameters and return a GenericResponse packet with a status code either set to kStatus_Success or an appropriate error status code.

5.13 Call command

The Call command executes a function that is written in memory at the address sent in the command. The address needs to be a valid memory location residing in accessible flash (internal or external) or in RAM. The command supports the passing of one 32-bit argument. Although the command supports a stack address, at this time the call still takes place using the current stack pointer. After execution of the function, a 32-bit return value is returned in the generic response message.

Table 5-21. Parameters for Call Command

Byte #	Command
0 - 3	Call address
4 - 7	Argument word
8 - 11	Stack pointer

Response: The target returns a GenericResponse packet with a status code either set to the return value of the function called or set to kStatus_InvalidArgument (105).

5.14 Reset command

The Reset command results in the bootloader resetting the chip.

The Reset command requires no parameters.

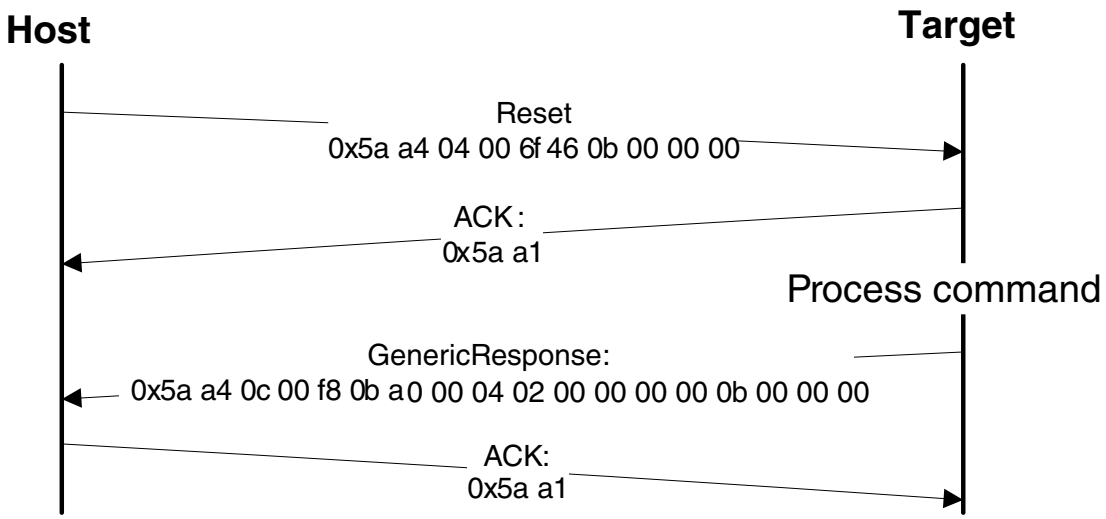

Figure 5-9. Protocol Sequence for Reset Command

Table 5-22. Reset Command Packet Format (Example)

Reset	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x04 0x00
	crc16	0x6F 0x46
Command packet	commandTag	0x0B - reset
	flags	0x00
	reserved	0x00
	parameterCount	0x00

The Reset command has no data phase.

Response: The target returns a GenericResponse packet with status code set to kStatus_Success, before resetting the chip.

5.15 FlashProgramOnce command

The FlashProgramOnce command writes data (that is provided in a command packet) to a specified range of bytes in the program once field. Special care must be taken when writing to the program once field.

- The program once field only supports programming once, so any attempted to reprogram a program once field gets an error response.
- Writing to the program once field requires the byte count to be 4-byte aligned or 8-byte aligned.

The FlashProgramOnce command uses three parameters: index 2, byteCount, data.

Table 5-23. Parameters for FlashProgramOnce Command

Byte #	Command
0 - 3	Index of program once field
4 - 7	Byte count (must be evenly divisible by 4)
8 - 11	Data
12 - 16	Data

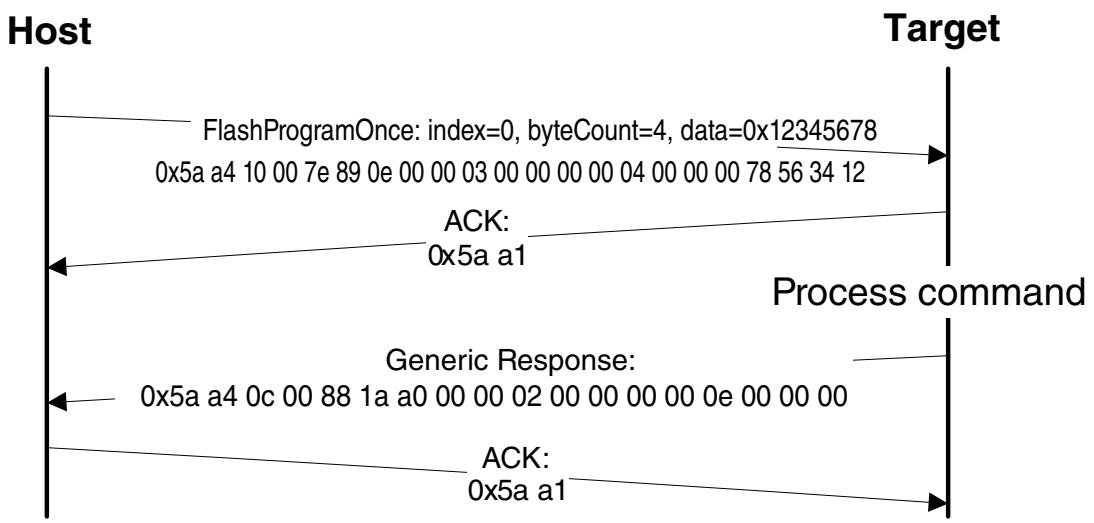

Figure 5-10. Protocol Sequence for FlashProgramOnce Command

Table 5-24. FlashProgramOnce Command Packet Format (Example)

FlashProgramOnce	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x10 0x00
	crc16	0x7E4 0x89
Command packet	commandTag	0x0E – FlashProgramOnce
	flags	0
	reserved	0
	parameterCount	3
	index	0x0000_0000

Table continues on the next page...

Table 5-24. FlashProgramOnce Command Packet Format (Example) (continued)

FlashProgramOnce	Parameter	Value
	byteCount	0x0000_0004
	data	0x1234_5678

Response: upon successful execution of the command, the target (Kinetis Flashloader) returns a GenericResponse packet with a status code set to kStatus_Success, or to an appropriate error status code.

5.16 FlashReadOnce command

The FlashReadOnce command returns the contents of the program once field by given index and byte count. The FlashReadOnce command uses 2 parameters: index and byteCount.

Table 5-25. Parameters for FlashReadOnce Command

Byte #	Parameter	Description
0 - 3	index	Index of the program once field (to read from)
4 - 7	byteCount	Number of bytes to read and return to the caller

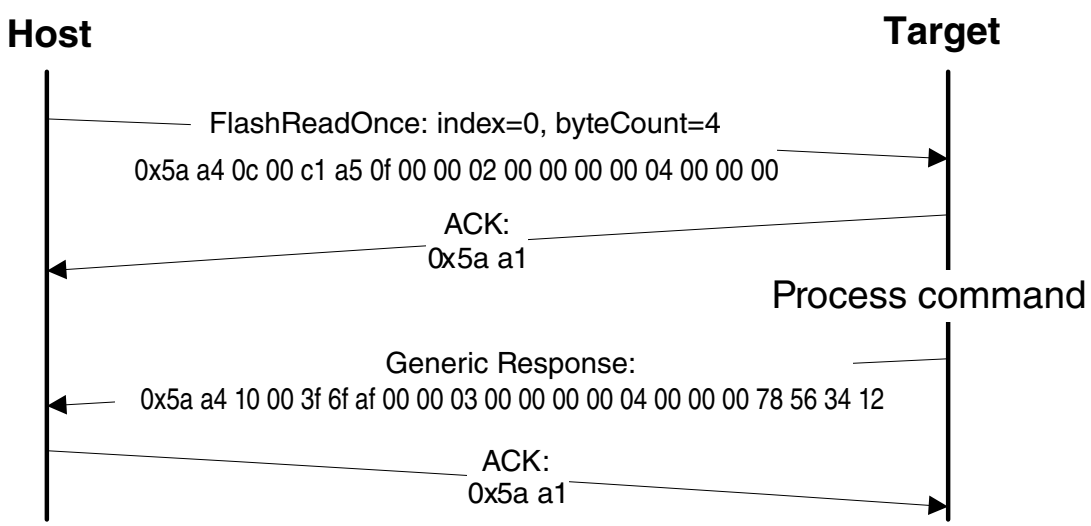

Figure 5-11. Protocol Sequence for FlashReadOnce Command

Table 5-26. FlashReadOnce Command Packet Format (Example)

FlashReadOnce	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4
	length	0x0C 0x00
	crc	0xC1 0xA5
Command packet	commandTag	0x0F – FlashReadOnce
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	index	0x0000_0000
	byteCount	0x0000_0004

Table 5-27. FlashReadOnce Response Format (Example)

FlashReadOnce Response	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4
	length	0x10 0x00
	crc	0x3F 0x6F
Command packet	commandTag	0xAF
	flags	0x00
	reserved	0x00
	parameterCount	0x03
	status	0x0000_0000
	byteCount	0x0000_0004
	data	0x1234_5678

Response: upon successful execution of the command, the target returns a FlashReadOnceResponse packet with a status code set to kStatus_Success, a byte count and corresponding data read from Program Once Field upon successful execution of the command, or returns with a status code set to an appropriate error status code and a byte count set to 0.

5.17 FlashReadResource command

The FlashReadResource command returns the contents of the IFR field or Flash firmware ID, by given offset, byte count, and option. The FlashReadResource command uses 3 parameters: start address, byteCount, option.

Table 5-28. Parameters for FlashReadResource Command

Byte #	Parameter	Command
0 - 3	start address	Start address of specific non-volatile memory to be read
4 - 7	byteCount	Byte count to be read
8 - 11	option	0: IFR 1: Flash firmware ID

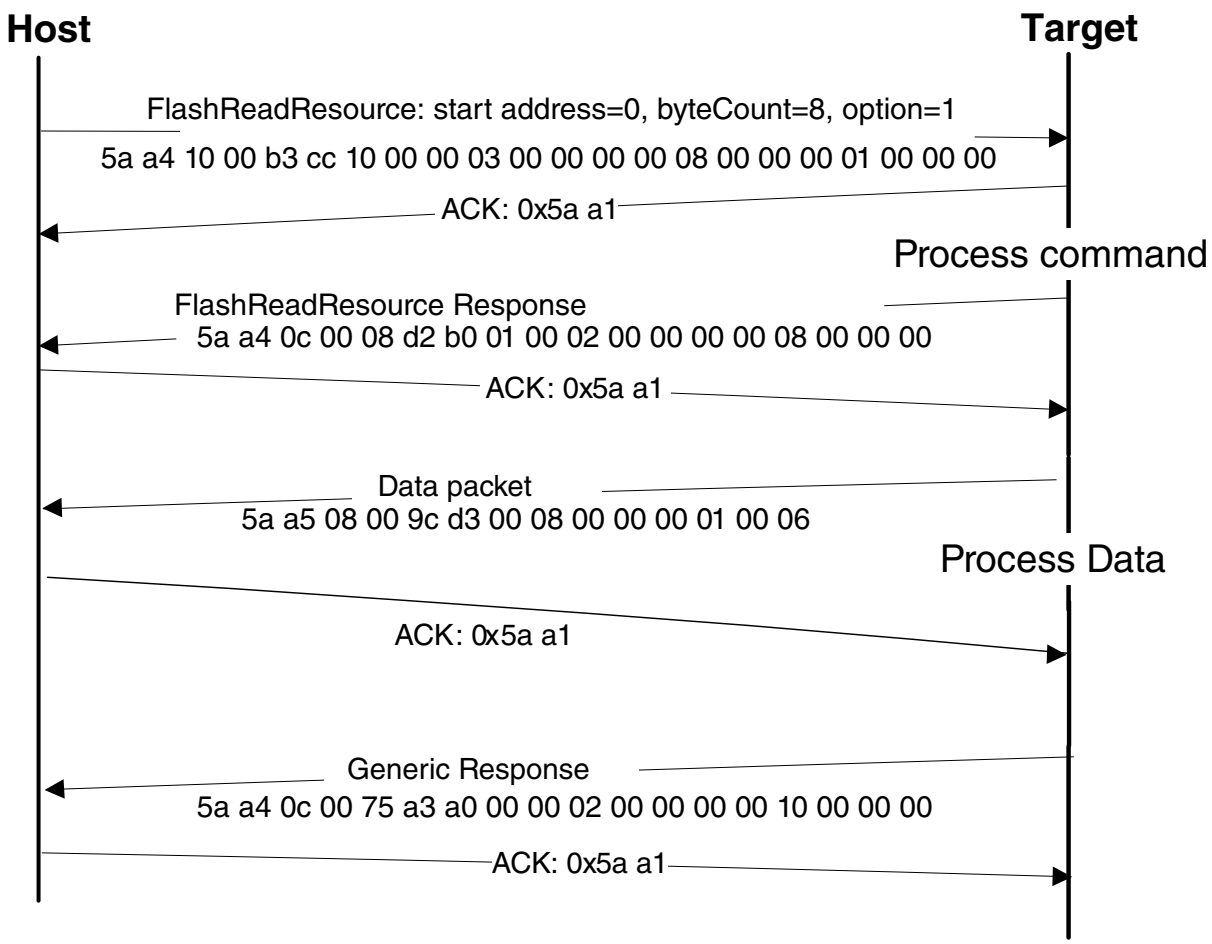

Figure 5-12. Protocol Sequence for FlashReadResource Command

Table 5-29. FlashReadResource Command Packet Format (Example)

FlashReadResource	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4
	length	0x10 0x00
	crc	0xB3 0xCC
Command packet	commandTag	0x10 – FlashReadResource
	flags	0x00

Table continues on the next page...

Table 5-29. FlashReadResource Command Packet Format (Example) (continued)

FlashReadResource	Parameter	Value
	reserved	0x00
	parameterCount	0x03
	startAddress	0x0000_0000
	byteCount	0x0000_0008
	option	0x0000_0001

Table 5-30. FlashReadResource Response Format (Example)

FlashReadResource Response	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4
	length	0x0C 0x00
	crc	0xD2 0xB0
Command packet	commandTag	0xB0
	flags	0x01
	reserved	0x00
	parameterCount	0x02
	status	0x0000_0000
	byteCount	0x0000_0008

Data phase: The FlashReadResource command has a data phase. Because the target (Kinetis bootloader) works in slave mode, the host must pull data packets until the number of bytes of data *specified in the byteCount parameter of FlashReadResource command* are received by the host.

Chapter 6
Supported peripherals

6.1 Introduction

This section describes the peripherals supported by the Kinetis bootloader. To use an interface for bootloader communications, the peripheral must be enabled in the BCA.If the BCA is invalid (such as all 0xFF bytes), then all peripherals are enabled by default.

6.2 I2C Peripheral

The Kinetis bootloader supports loading data into flash via the I2C peripheral, where the I2C peripheral serves as the I2C slave. A 7-bit slave address is used during the transfer.

Customizing an I2C slave address is also supported. This feature is enabled if the Bootloader Configuration Area (BCA) is enabled (tag field is filled with ‘kcfg’) and the i2cSlaveAddress field is filled with a value other than 0xFF. Otherwise, 0x10 is used as the default I2C slave address.

The Kinetis Flashloader uses 0x10 as the I2C slave address, and supports 400 kbps as the I2C baud rate.

The maximum supported I2C baud rate depends on corresponding clock configuration field in the BCA. Typical supported baud rate is 400 kbps with factory settings. Actual supported baud rate may be lower or higher than 400 kbps, depending on the actual value of the clockFlags and the clockDivider fields.

Because the I2C peripheral serves as an I2C slave device, each transfer should be started by the host, and each outgoing packet should be fetched by the host.

- An incoming packet is sent by the host with a selected I2C slave address and the direction bit is set as write.

- An outgoing packet is read by the host with a selected I2C slave address and the direction bit is set as read.
- 0x00 is sent as the response to host if the target is busy with processing or preparing data.

The following flow charts demonstrate the communication flow of how the host reads ping packet, ACK and response from the target.

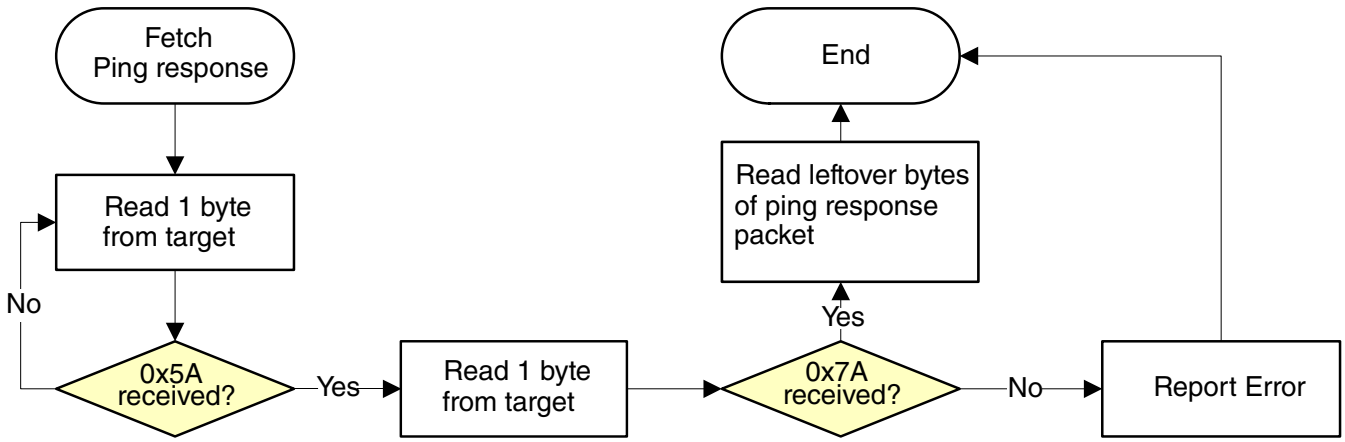

Figure 6-1. Host reads ping response from target via I2C

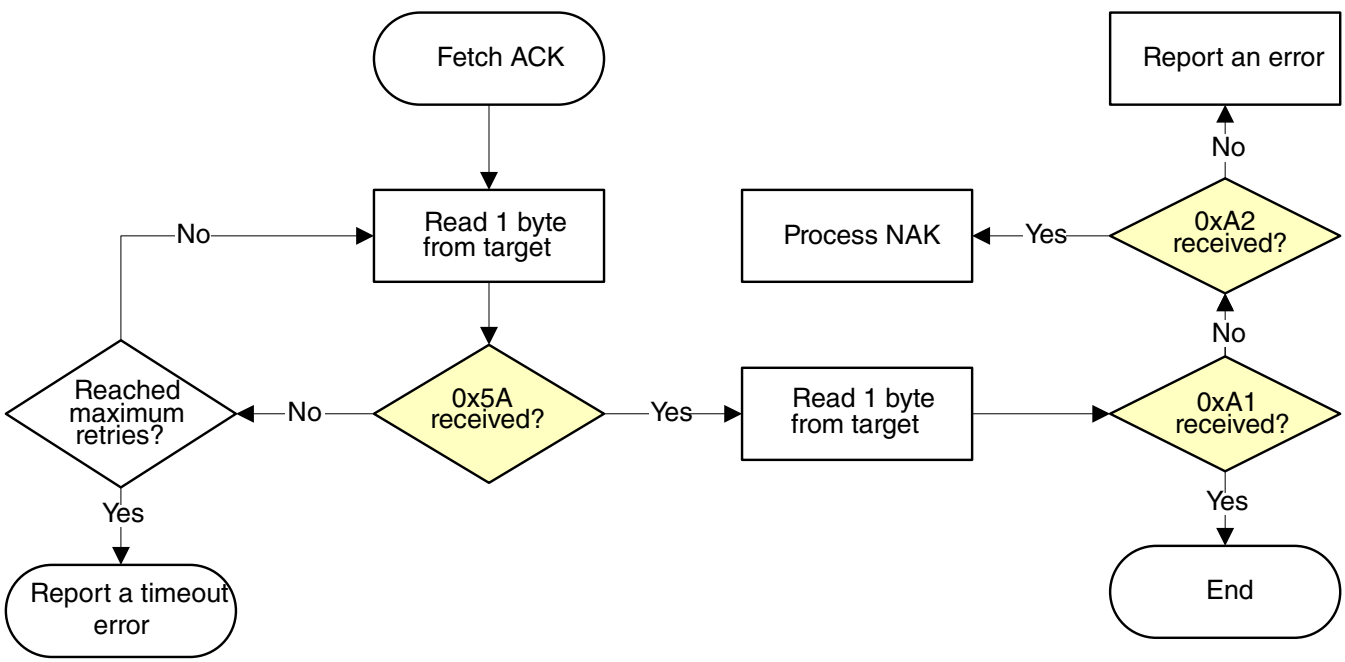

Figure 6-2. Host reads ACK packet from target via I2C

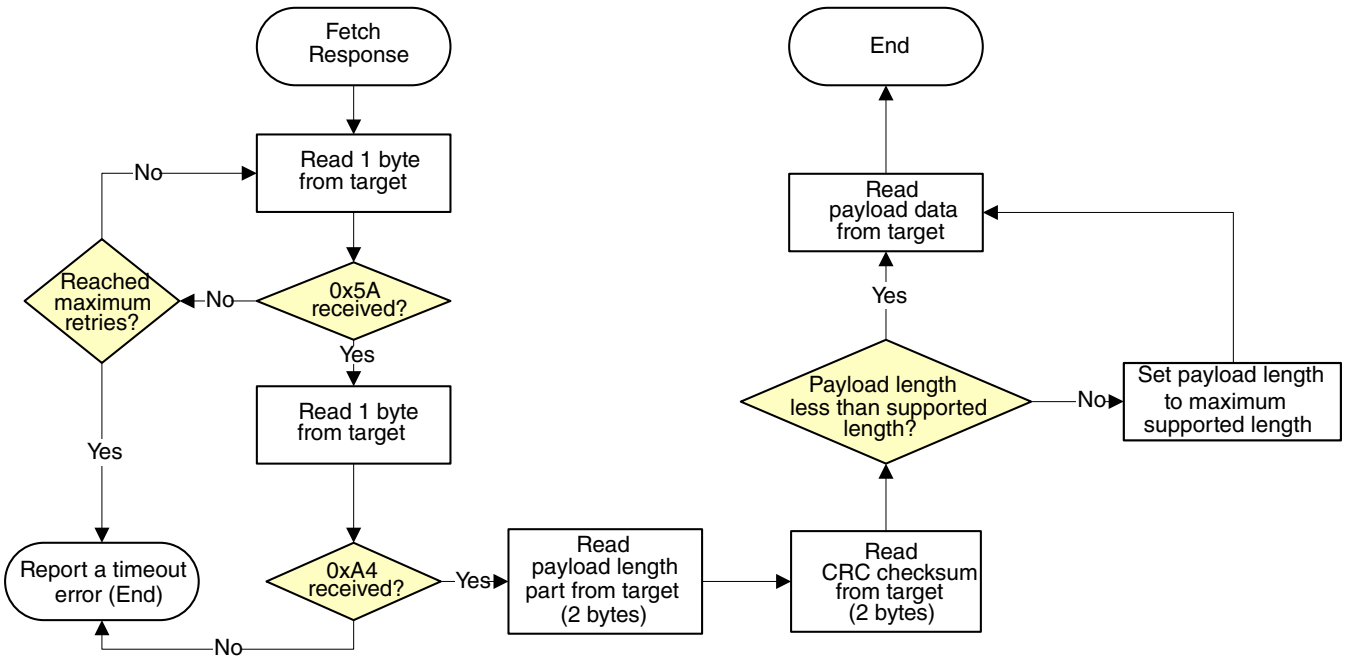

Figure 6-3. Host reads response from target via I2C

6.3 SPI Peripheral

The Kinetis bootloader supports loading data into flash via the SPI peripheral, where the SPI peripheral serves as a SPI slave.

Maximum supported baud rate of SPI depends on the clock configuration fields in the Bootloader Configuration Area (BCA). The typical supported baud rate is 400 kbps with the factory settings. The actual baud rate is lower or higher than 400 kbps, depending on the actual value of the clockFlags and clockDivider fields in the BCA.

The Kinetis Flashloader supports 400 kbps as the SPI baud rate.

Because the SPI peripheral serves as a SPI slave device, each transfer should be started by the host, and each outgoing packet should be fetched by the host.

The transfer on SPI is slightly different from I2C:

- Host receives 1 byte after it sends out any byte.
- Received bytes should be ignored when host is sending out bytes to target
- Host starts reading bytes by sending 0x00s to target
- The byte 0x00 is sent as response to host if target is under the following conditions:
  - Processing incoming packet
  - Preparing outgoing data
  - Received invalid data

The following flowcharts demonstrate how the host reads a ping response, an ACK and a command response from target via SPI.

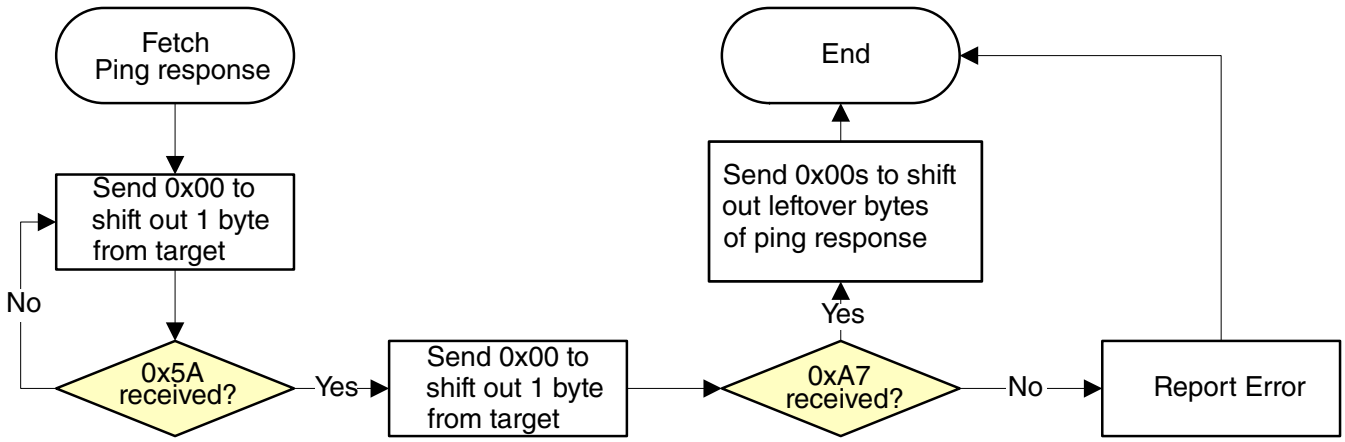

Figure 6-4. Host reads ping packet from target via SPI

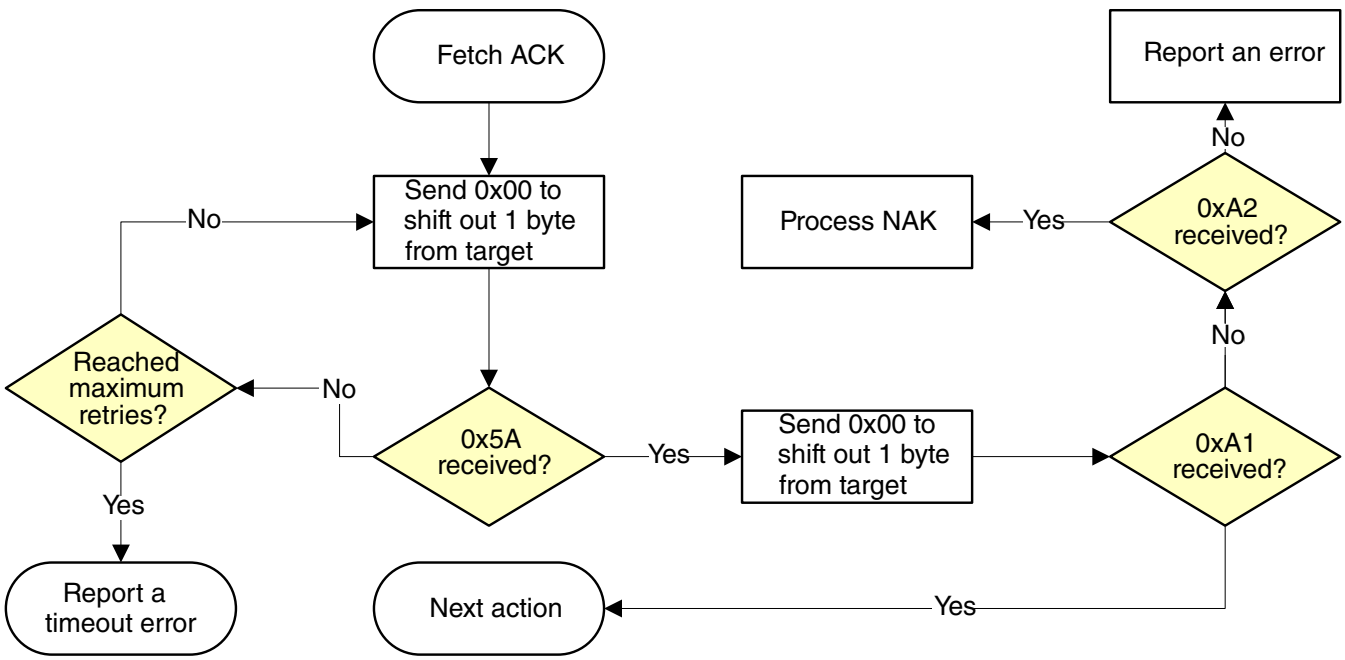

Figure 6-5. Host reads ACK from target via SPI

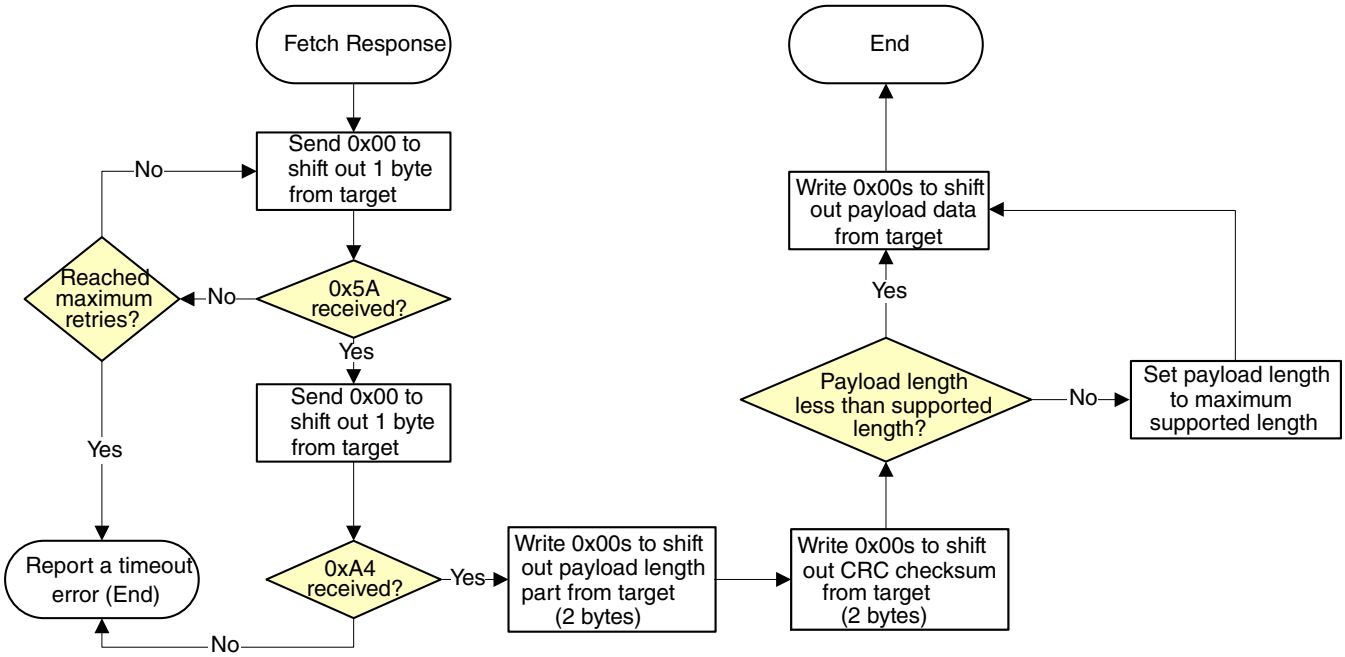

Figure 6-6. Host reads response from target via SPI

6.4 UART Peripheral

The Kinetis bootloader integrates an autobaud detection algorithm for the LPUART peripheral, thereby providing flexible baud rate choices.

Autobaud feature: If LPUART*n* is used to connect to the bootloader, then the LPUART*n*_RX (PTB2)(PTA1) pin must be kept high and not left floating during the detection phase in order to comply with the autobaud detection algorithm. After the bootloader detects the ping packet (0x5A 0xA6) on LPUART*n*_RX, the bootloader firmware executes the autobaud sequence. If the baudrate is successfully detected, then the bootloader sends a ping packet response [(0x5A 0xA7), protocol version (4 bytes), protocol version options (2 bytes) and crc16 (2 bytes)] at the detected baudrate. The Kinetis bootloader then enters a loop, waiting for bootloader commands via the LPUART peripheral.

NOTE

The data bytes of the ping packet must be sent continuously (with no more than 80 ms between bytes) in a fixed LPUART transmission mode (8-bit data, no parity bit and 1 stop bit). If the bytes of the ping packet are sent one-by-one with more than 80 ms delay between them, then the autobaud detection

algorithm may calculate an incorrect baud rate. In this case, the autobaud detection state machine should be reset.

Supported baud rates: The baud rate is closely related to the MCU core and system clock frequencies. Typical baud rates supported are 9600, 19200, 38400, and 57600. Of course, to influence the performance of autobaud detection, the clock configuration in BCA can be changed.

Packet transfer: After autobaud detection succeeds, bootloader communications can take place over the LPUART peripheral. The following flow charts show:

- How the host detects an ACK from the target
- How the host detects a ping response from the target
- How the host detects a command response from the target

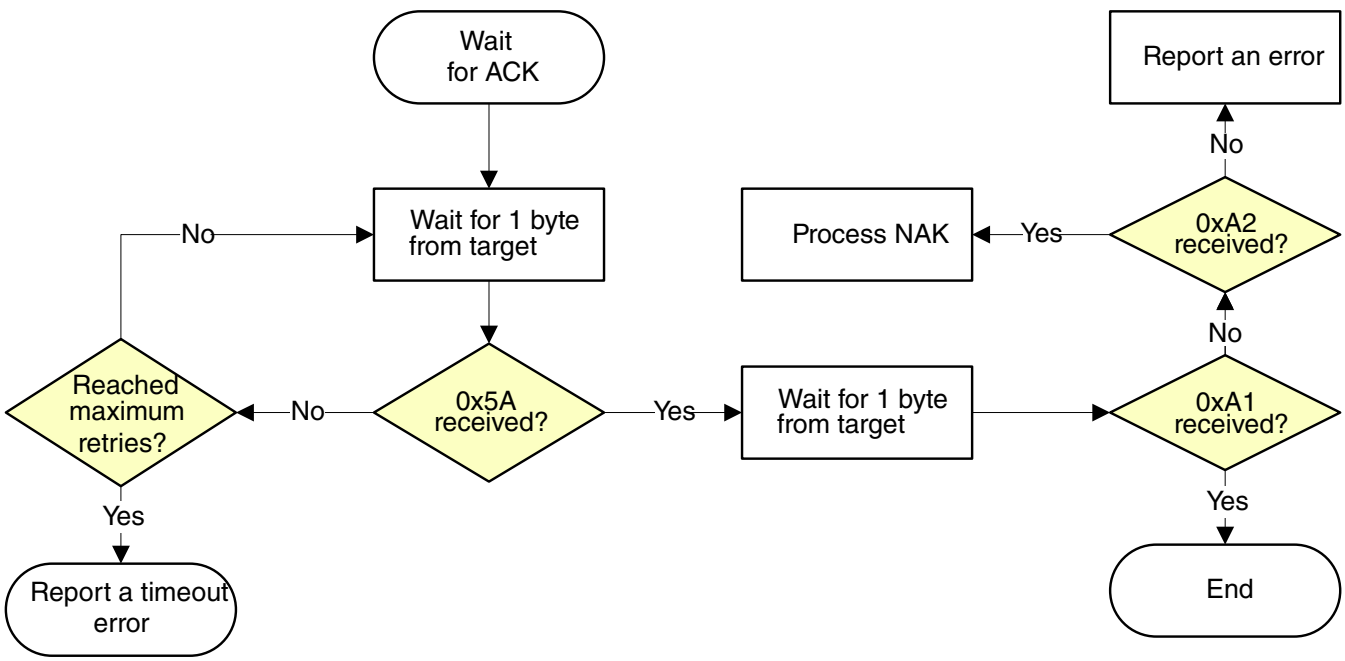

Figure 6-7. Host reads an ACK from target via LPUART

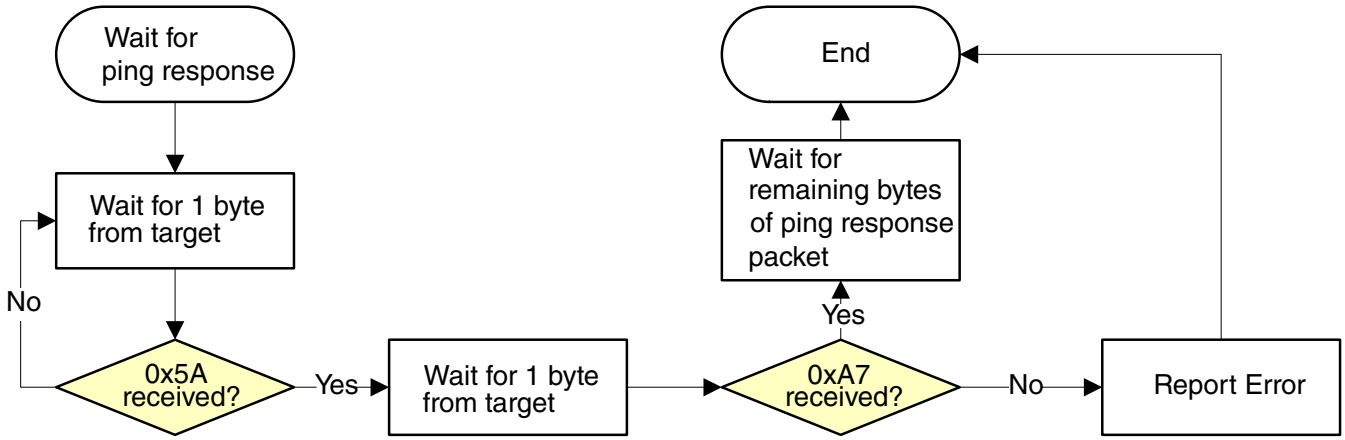

Figure 6-8. Host reads a ping response from target via LPUART

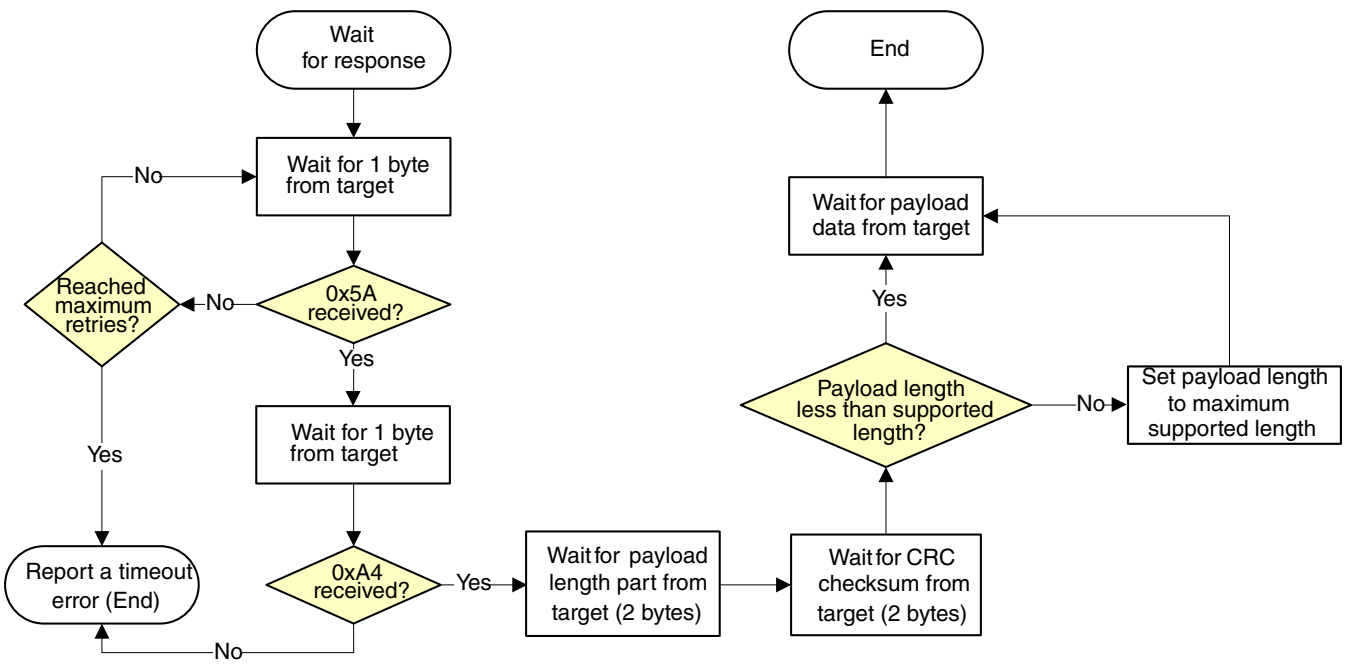

Figure 6-9. Host reads a command response from target via LPUART

6.5 USB Peripheral

The Kinetis bootloader supports loading data into flash via the USB peripheral. The target is implemented as a USB HID class.

USB HID does not use framing packets; instead the packetization inherent in the USB protocol itself is used. The ability for the device to NAK Out transfers (until they can be received) provides the required flow control; the built-in CRC of each USB packet provides the required error detection.

6.5.1 Clock configuration

The bootloader supports the crystal-less USB feature. If the USB peripheral is enabled, the bootloader enables the 48-MHz IRCHIRC (by setting SIM_SOPT2[USBSRC | PLLFSLSEL] to 1). The flashloaderROM also enables the USB clock recovery feature (by setting USB_CLK_RECOVER_CTRL[CLOCK_RECOVER_EN] to 1 and USB_CLK_RECOVER_IRC_EN[IRC_EN] to 1).

6.5.2 Device descriptor

The Kinetis bootloader configures the default USB VID/PID/Strings as below:

Default VID/PID:

- VID = 0x15A2
- PID = 0x0073

Default Strings:

- Manufacturer [1] = "Freescale Semiconductor Inc."
- Product [2] = "Kinetis bootloader"

You can customize the USB VID/PID/Strings with the Bootloader Configuration Area (BCA) of the flash. For example, the USB VID and PID can be customized by writing the new VID to the usbVid(BCA + 0x14) field and the new PID to the usbPid(BCA + 0x16) field of the BCA in flash. To change the USB strings, you need to prepare a structure (like the one shown below) in the flash, and then write the address of the g_languages structure to the usbStringsPointer(BCA + 0x18) field of the BCA.

```
g_languages = { USB_STR_0,
sizeof(USB_STR_0),
(uint_16)0x0409,
(const uint_8 **)g_string_descriptors,
g_string_desc_size};
the USB_STR_0, g_string_descriptors and g_string_desc_size are defined as below.
USB_STR_0[4] =    {0x02,
0x03,
0x09,
0x04
};
g_string_descriptors[4] =
{ USB_STR_0,
USB_STR_1,
USB_STR_2,
USB_STR_3};
g_string_desc_size[4] =
{ sizeof(USB_STR_0),
sizeof(USB_STR_1),
sizeof(USB_STR_2),
```

```
sizeof(USB_STR_3)};
```

You can make your own structure of USB_STR_1, USB_STR_2, USB_STR_3:

- USB_STR_1 is used for the manufacturer string.
- USB_STR_2 is used for the product string.
- USB_STR_3 is used for the serial number string.

By default, the 3 strings are defined as below:

```
USB_STR_1[] =
{  sizeof(USB_STR_1),
USB_STRING_DESCRIPTOR,
'F',0,
'r',0,
'e',0,
'e',0,
's',0,
'c',0,
'a',0,
'l',0,
'e',0,
' ',0,
'S',0,
'e',0,
'm',0,
'i',0,
'c',0,
'o',0,
'n',0,
'd',0,
'u',0,
'c',0,
't',0,
'o',0,
'r',0,
' ',0,
'I',0,
'n',0,
'c',0,
'.',0
};

USB_STR_2[] =
{  sizeof(USB_STR_2),
USB_STRING_DESCRIPTOR,
'M',0,
'K',0,
' ',0,
'M',0,
'a',0,
's',0,
's',0,
' ',0,
'S',0,
't',0,
'o',0,
'r',0,
'a',0,
'g',0,
'e',0
};

USB_STR_3[] =
{  sizeof(USB_STR_3),
```

```
        USB_STRING_DESCRIPTOR,
        '0',0,
        '1',0,
        '2',0,
        '3',0,
        '4',0,
        '5',0,
        '6',0,
        '7',0,
        '8',0,
        '9',0,
        'A',0,
        'B',0,
        'C',0,
        'D',0,
        'E',0,
        'F',0
        };
```

6.5.3 Endpoints

The HID peripheral uses 3 endpoints:

- Control (0)
- Interrupt IN (1)
- Interrupt OUT (2)

The Interrupt OUT endpoint is optional for HID class devices, but the Kinetis bootloader uses it as a pipe, where the firmware can NAK send requests from the USB host.

6.5.4 HID reports

There are 4 HID reports defined and used by the bootloader USB HID peripheral. The report ID determines the direction and type of packet sent in the report; otherwise, the contents of all reports are the same.

Report ID	Packet Type	Direction
1	Command	OUT
2	Data	OUT
3	Command	IN
4	Data	IN

For all reports, these properties apply:

Usage Min	1

Table continues on the next page...

Usage Max	1
Logical Min	0
Logical Max	255
Report Size	8
Report Count	34

Each report has a maximum size of 34 bytes. This is derived from the minimum bootloader packet size of 32 bytes, plus a 2-byte report header that indicates the length (in bytes) of the packet sent in the report.

NOTE

In the future, the maximum report size may be increased, to support transfers of larger packets. Alternatively, additional reports may be added with larger maximum sizes.

The actual data sent in all of the reports looks like:

0	Report ID
1	Packet Length LSB
2	Packet Length MSB
3	Packet[0]
4	Packet[1]
5	Packet[2]
	...
N+3-1	Packet[N-1]

This data includes the Report ID, which is required if more than one report is defined in the HID report descriptor. The actual data sent and received has a maximum length of 35 bytes. The Packet Length header is written in little-endian format, and it is set to the size (in bytes) of the packet sent in the report. This size does not include the Report ID or the Packet Length header itself. During a data phase, a packet size of 0 indicates a data phase abort request from the receiver.

Chapter 7
Peripheral interfaces

7.1 Introduction

The block diagram shows connections between components in teh architecture of the peripheral interface.

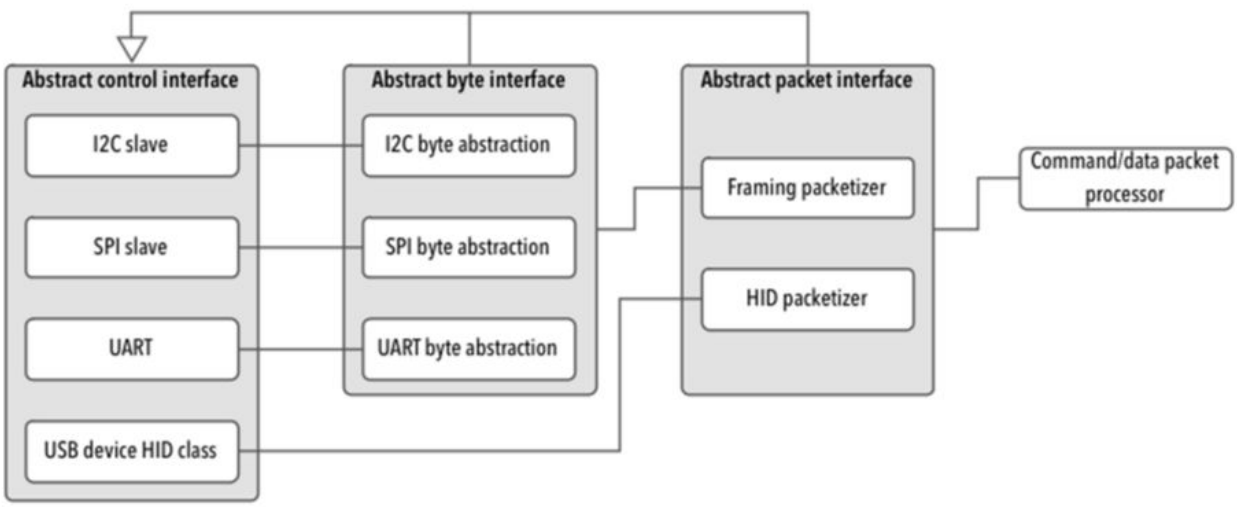

Figure 7-1. Components peripheral interface

In this diagram, the byte and packet interfaces are shown to inherit from the control interface.

All peripheral drivers implement an abstract interface built on top of the driver's internal interface. The outermost abstract interface is a packet-level interface. It returns the payload of packets to the caller. Drivers which use framing packets have another abstract interface layer that operates at the byte level. The abstract interfaces allow the higher layers to use exactly the same code regardless which peripheral is being used.

The abstract packet interface feeds into the command and data packet processor. This component interprets the packets returned by the lower layer as command or data packets.

7.2 Abstract control interface

This control interface provides a common method to initialize and shutdown peripheral drivers. It also provides the means to perform the active peripheral detection. No data transfer functionality is provided by this interface. That is handled by the interfaces that inherit the control interface.

The main reason this interface is separated out from the byte and packet interfaces is to show the commonality between the two. It also allows the driver to provide a single control interface structure definition that can be easily shared.

```
struct BoatloaderInitInfo
{
    void * contextArea; //!< Pointer to memory region for use by the driver.
    uint32_t available; //!< Size of the memory region the driver can use.
    uint32_t used;      //!< Actual number of bytes used by the driver (filled in by the
driver).
};

struct PeripheralDescriptor {
    //! @brief Bit mask identifying the peripheral type.
    //!
    //! See #_peripheral_types for a list of valid bits.
    uint32_t typeMask;

    //! @brief The instance number of the peripheral.
    uint32_t instance;

    //! @brief Control interface for the peripheral.
    const peripheral_control_interface_t * controlInterface;

    //! @brief Byte-level interface for the peripheral.
    //!
    //! May be NULL since not all periperhals support this interface.
    const peripheral_byte_inteface_t * byteInterface;

    //! @brief Packet level interface for the peripheral.
    const peripheral_packet_interface_t * packetInterface;
};

struct PeripheralControlInterface
{
    status_t (*minimalInit)(const PeripheralDescriptor * self, BoatloaderInitInfo * info);
    void (*minimalShutdown)(const PeripheralDescriptor * self);
    bool (*pollForActivity)(const PeripheralDescriptor * self);
```

```
    status_t (*init)(const PeripheralDescriptor * self, BoatloaderInitInfo * info);
    void (*shutdown)(const PeripheralDescriptor * self);
```

Table 7-1. Abstract control interface

Interface	Description
minimalInit()	Initialize the driver only enough to detect start of communications.
minimalShutdown()	Shutdown the driver from its minimal init state.
pollForActivity()	Check whether communications has started.
init()	Fully initialize the driver.
shutdown()	Shutdown the fully initialized driver.

After minimalShutdown() is called, the driver is expected to no longer use any memory that it allocated through the BootloaderInitInfo structure.

7.3 Abstract byte interface

This interface exists to give the framing packetizer, which is explained in the later section, a common interface for the peripherals that use framing packets.

The abstract byte interface inherits the abstract control interface.

```
struct PeripheralByteInterface
{
    status_t (*init)(const peripheral_descriptor_t * self, bootloader_init_info_t * info);
    status_t (*read)(uint8_t * buffer, uint32_t requestedBytes, uint32_t * actualBytes);
    status_t (*write)(const uint8_t * buffer, uint32_t byteCount);
};
```

Table 7-2. Abstract byte interface

Interface	Description
init()	Initialize the interface.
read()	Return the requested number of bytes. Blocks until all bytes available.
write()	Write the requested number of bytes.

The read() interface returns a pointer into the driver's internal buffer. No data is copied. The driver must ensure that the returned range of bytes is not overwritten until the next call into read(). Each call of this interface tells the driver that it may reuse the range of bytes that it last returned.

7.4 Abstract packet interface

The abstract packet interface inherits the abstract control interface.

```
struct PeripheralPacketInterface
{
    status_t (*init)(const PeripheralDescriptor * self, BoatloaderInitInfo * info);
    status_t (*readPacket)(const PeripheralDescriptor * self, uint8_t ** packet, uint32_t *
packetLength, packet_type_t packetType);
    status_t (*writePacket)(const PeripheralDescriptor * self, const uint8_t * packet,
uint32_t byteCount, packet_type_t packetType);
    void (*abortDataPhase)(const PeripheralDescriptor * self);
    status_t (*finalize)(const PeripheralDescriptor * self);
    uint32_t (*getCurrentMaxBufferSize)(const PeripheralDescriptor * self);
    status_t (*requestNewMaxBufferSize)(const PeripheralDescriptor * self, uint32_t
newBufferSize);
};
```

Table 7-3. Abstract packet interface

Interface	Description
init()	Initialize the peripheral.
readPacket()	Read a full packet from the peripheral.
writePacket()	Send a complete packet out the peripheral.
abortDataPhase()	Abort receiving of data packets.
finalize()	Shut down the peripheral when done with use.
getCurrentMaxBufferSize()	Returns the current maximum buffer size.
requestNewMaxBufferSize()	Requests to set a new maximum buffer size.

7.5 Framing packetizer

The framing packetizer processes framing packets received via the byte interface with which it talks. It builds and validates a framing packet as it reads bytes. And it constructs outgoing framing packets as needed to add flow control information and command or data packets. The framing packet also supports data phase abort.

7.6 USB HID packetizer

The USB HID packetizer implements the abstract packet interface for USB HID, taking advantage of the USB's inherent flow control and error detection capabilities. The USB HID packetizer provides a link layer that supports variable length packets and data phase abort.

7.7 Command/data processor

This component reads complete packets from the abstract packet interface, and interprets them as either command packets or data packets. The actual handling of each command is done by command handlers called by the command processor. The command handler tells the command processor whether a data phase is expected and how much data it is expected to receive.

If the command/data processor receives a unexpected command or data packet, it ignores it. In this case, the communications link resynchronizes upon reception of the next valid command.

Chapter 8
Memory interface

8.1 Abstract interface

The bootloader uses a common, abstract interface to implement the memory read/write/fill commands. This is to keep the command layer from having to know the details of the memory map and special routines.

This shared memory interface structure is used for both the high-level abstract interface, as well as low-level entries in the memory map.

```
struct MemoryInterface
{
    status_t (*read)(uint32_t address, uint32_t length, uint8_t * buffer);
    status_t (*write)(uint32_t address, uint32_t length, const uint8_t * buffer);
    status_t (*fill)(uint32_t address, uint32_t length, uint32_t pattern);
}
```

The global bootloader context contains a pointer to the high-level abstract memory interface, which is one of the MemoryInterface structures. The internal implementation of this abstract interface uses a memory map table, referenced from the global bootloader context that describes the various regions of memory that are accessible and provides region-specific operations.

The high-level functions are implemented to iterate over the memory map entries until it finds the entry for the specified address range. Read and write operations are not permitted to cross region boundaries, and an error is returned if such an attempt is made.

The BootloaderContext::memoryMap member is set to an array of these structures:

```
struct MemoryMapEntry
{
    uint32_t startAddress;
    uint32_t endAddress;
    const MemoryInterface * interface;
};
```

This array must be terminated with an entry with all fields set to zero.

The same MemoryInterface structure is also used to hold the memory-type-specific operations.

Note that the MemoryMapEntry::endAddress field must be set to the address of the last byte of the region, because a <= comparison is used.

During bootloader startup, the memory map is copied into RAM and modified to match the actual sizes of flash and RAM on the chip.

8.2 Flash driver interface

The flash driver uses the common memory interface to simplify the interaction with flash. It takes care of high level features such as read back verification, flash protection awareness, and so on. The flash memory functions map to the interface functions as so:

```
const memory_region_interface_t g_flashMemoryInterface = {
    .read = &normal_mem_read,
    .write = &flash_mem_write,
    .fill = &flash_mem_fill
};
```

Bootloader startup code is responsible for initializing the flash memory.

API	Description
normal_mem_read()	Performs a normal memory read.
flash_mem_write()	Calls the low-level flash_program() API. Also performs program verification if enabled with the Set Property command.
flash_mem_fill()	Performs intelligent fill operations on flash memory ranges. If the fill patterns are all 1's, special action is taken. If the range is a whole number of sectors, then those sectors are erased rather than filled. Any part of an all-1's fill that is not sector-aligned and -sized is ignored (the assumption being that it has been erased to 1's already). Fills for patterns other than all 1's call into flash_program().

Both flash_mem_write() and flash_mem_fill() checks the flash protection status for the sectors being programmed or erased and return an appropriate error if the operation is not allowed.

8.3 Low level flash driver

The low level flash driver (LLFD) handles erase and write operations on a word basis. It cannot perform writes of less than a full word.

Bootloader startup code is responsible for initializing and shutting down the LLFD.

```
status_t flash_init();
status_t flash_erase_all();
status_t flash_erase(uint32_t start, uint32_t lengthInBytes);
status_t flash_program(uint32_t start, uint32_t * src, uint32_t lengthInBytes);
status_t flash_get_security_state(flash_security_state_t * state);
status_t flash_security_bypass(const uint8_t * backdoorKey);
status_t flash_verify_erase_all(flash_margin_value_t * margin);
status_t flash_verify_erase(uint32_t start, uint32_t lengthInBytes, flash_margin_value_t
margin);
status_t flash_verify_program(uint32_t start, uint32_t lengthInBytes,
                              const uint8_t * expectedData, flash_margin_value_t margin,
                              uint32_t failedAddress, uint8_t *failedData);
status_t flash_is_region_protected(uint32_t start, uint32_t lengthInBytes,
                                   flash_protection_state_t * protection_state)
status_t flash_get_property(flash_property_t whichProperty, uint32_t * value)
```

Chapter 9
Kinetis bootloader porting

9.1 Introduction

This chapter discusses the steps required to port the Kinetis bootloader to an unsupported Kinetis MCU. Freescale is working to bring bootloader support to the entire Kinetis portfolio, but some devices still require user porting until all legacy device ports are complete. Each step of the porting process is discussed in detail in the following sections.

9.2 Choosing a starting point

The first step is to download the latest bootloader release. Freescale releases updates for the bootloader multiple times per year, so having the latest package is important for finding the best starting point for your port. To find the most recent bootloader release, freescale.com/KBOOT .

The easiest way to port the bootloader is to choose a supported target that is the closest match to the desired target MCU.

NOTE

Just because a supported device has a similar part number to the desired target MCU, it may not necessarily be the best starting point. To determine the best match, reference the datasheet and reference manual for all of the supported Kinetis devices.

9.3 Preliminary porting tasks

All references to paths in the rest of this chapter are relative to the root of the extracted Kinetis bootloader package. The container folder is named FSL_Kinetis_Bootloader_<version>. Before jumping in and modifying source code, the following tasks should be performed.

9.3.1 Download device header files

The most manual process in porting the bootloader to a new target is editing the device header files. This process is very time consuming and error prone, so Freescale provides CMSIS-compatible packages for all Kinetis devices that contain bootloader-compatible device header files. These packages can be found on the product page for the MCU.

NOTE

It is not recommended to proceed with a port if a package does not yet exist for the desired target MCU.

In the downloaded package, locate the folder with the header files. The folder is named after the MCU (for example, “MK64F12”) and contains a unique header file for each peripheral in addition to regs.h and system_<device>.h files. Copy the entire folder into the `/src/include/device` folder of the bootloader tree.

9.3.2 Copy the closest match

Copy the folder of the MCU that most closely matches the target MCU in the /targets folder of the bootloader source tree. Rename it to coincide with the target MCU part number.

Once the files are copied, browse the newly created folder. Rename all files that have reference to the device from which they were copied. The following files need to be renamed:

- clock_config_<old_device>.c —> clock_config_<new_device>.c
- hardware_init_<old_device>.c —> hardware_init _<new_device>.c
- memory_map_<old_device>.c —> memory_map _<new_device>.c
- peripherals_<old_device>.c —> peripherals _<new_device>.c
- startup_<old_device>.c —> startup _<new_device>.c

9.3.3 Provide device startup file (vector table)

A device-specific startup file is a key piece to the port. The bootloader may not function correctly without the correct vector table. A startup file from the closest match MCU can be used as a template, but it is strongly recommended that the file be thoroughly checked before using it in the port due to differences in interrupt vector mappings between Kinetis devices.

The startup file should be created and placed into a folder that references the target MCU and toolchain in the /src/startup folder of the bootloader source tree. Startup files are always assembly (*.s) and are named startup_<device>.s.

9.3.4 Clean up the IAR project

The folder copy performed in step 1.2.2 copies more than just source code files. Inside of the newly created /targets/<device> folder, locate the IAR workspace file (bootloader.eww) and open it. This image shows an example of what a workspace looks like and the files that need to be touched.

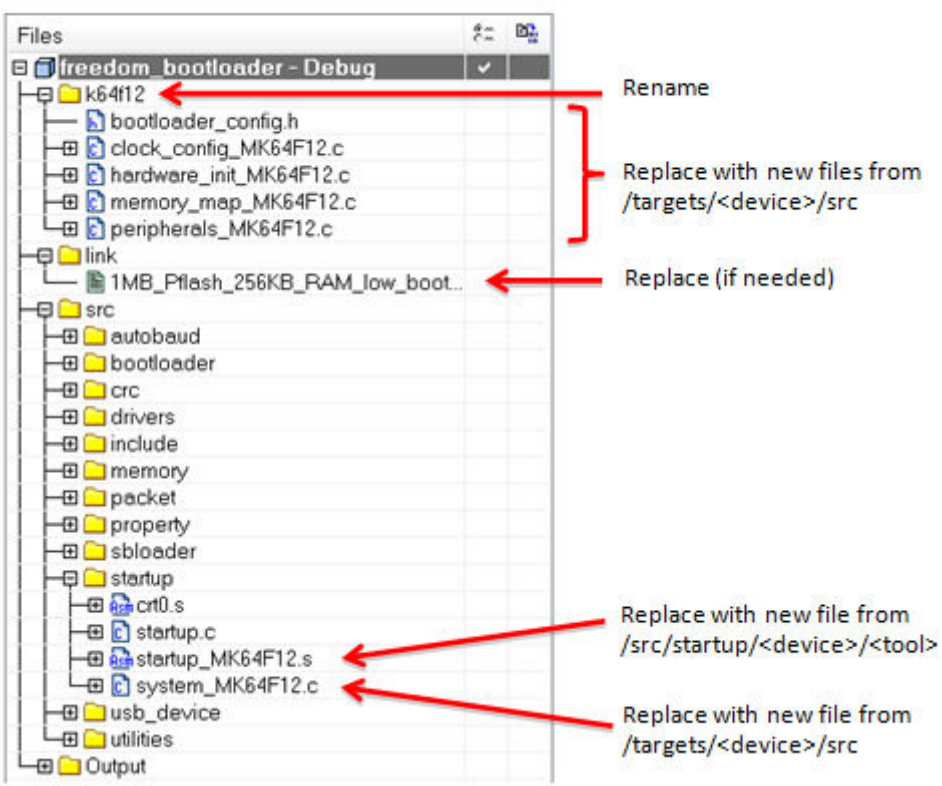

Figure 9-1. IAR workspace

Once changes have been made, update the project to reference the target MCU. This can be found in the project options.

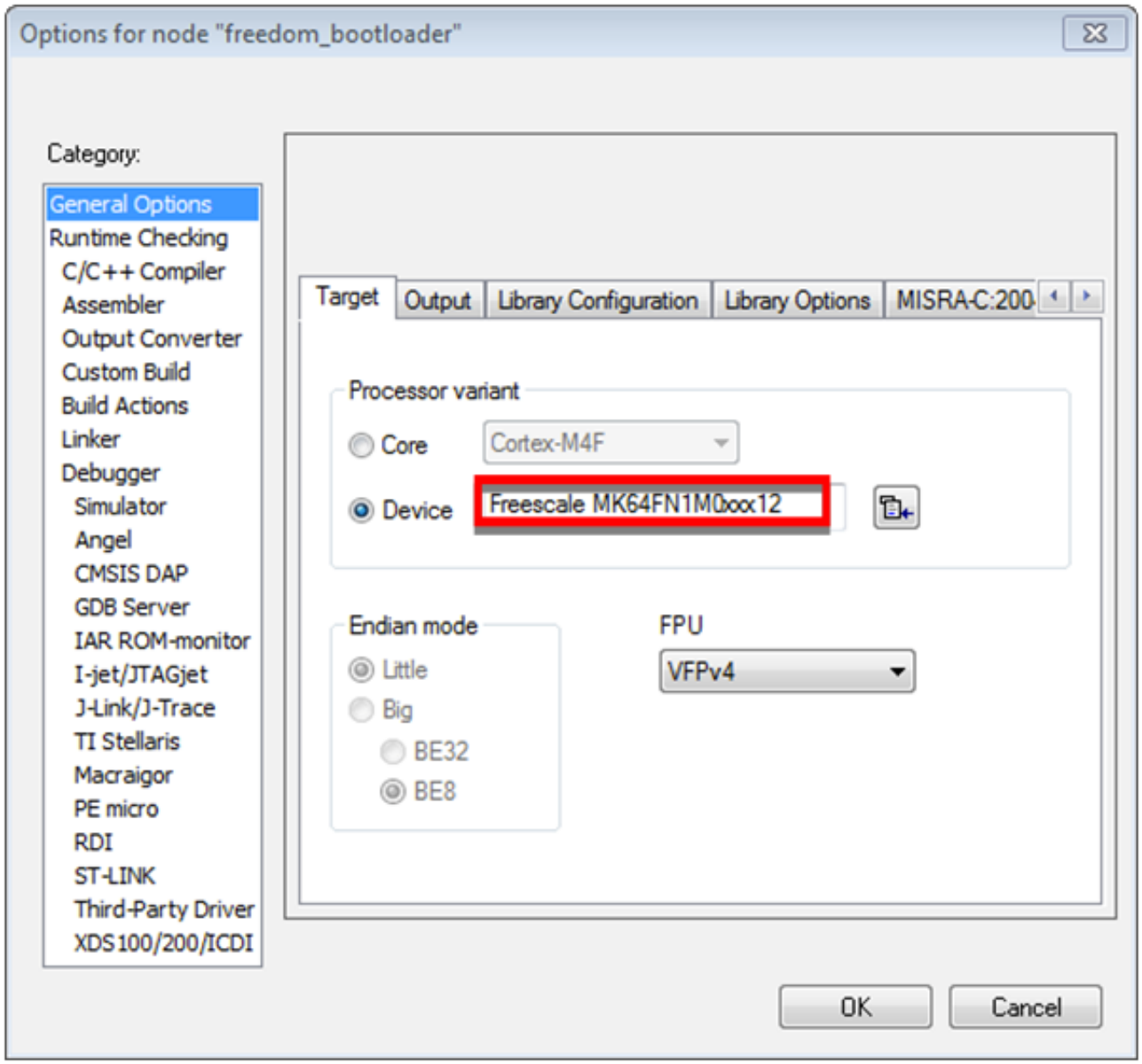

Figure 9-2. Project options

9.3.5 Bootloader peripherals

There is a C/C++ preprocessor define that is used by the bootloader source to configure the bootloader based on the target MCU. This define must be updated to reference the correct set of device-specific header files.

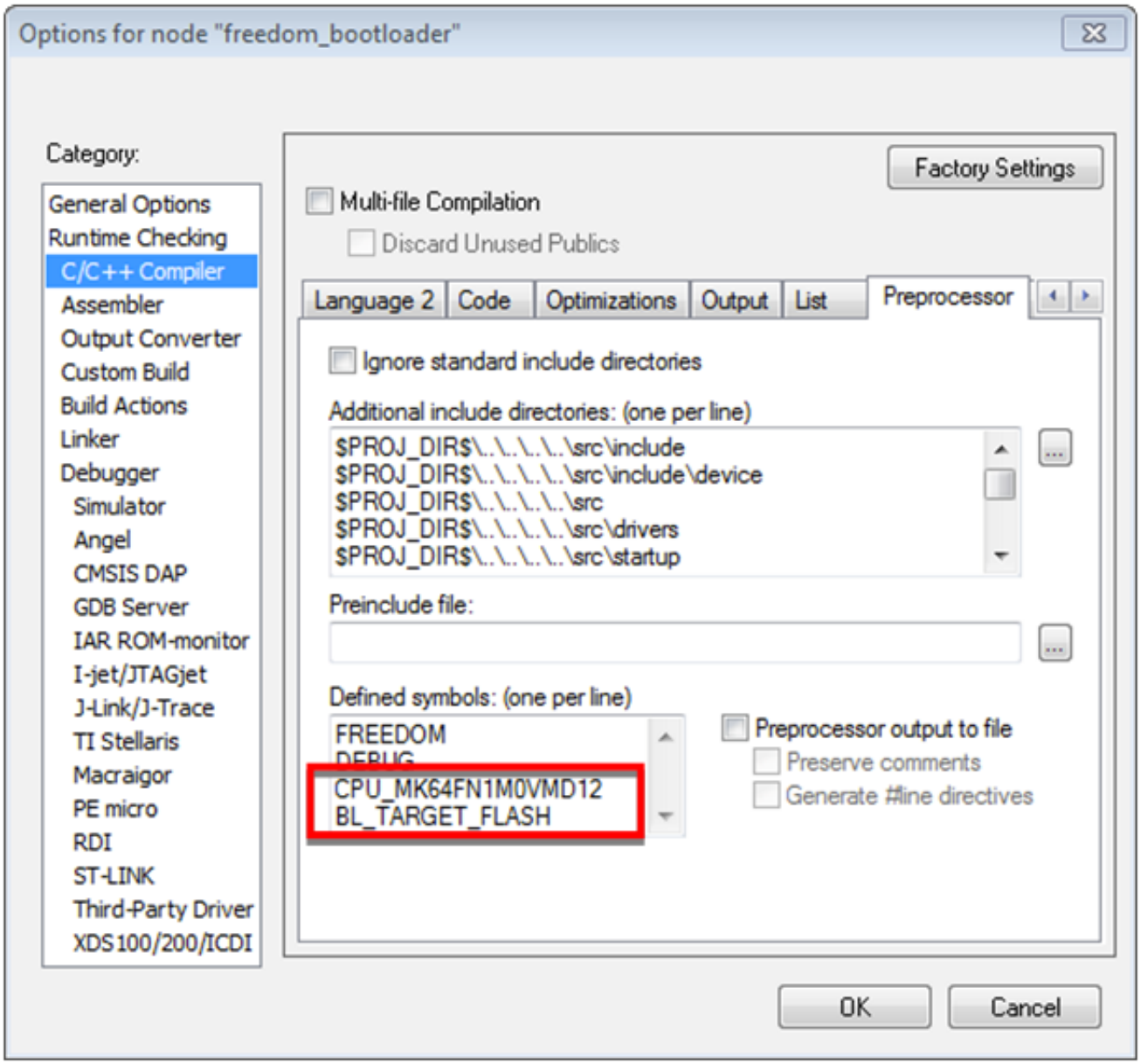

Figure 9-3. Options for node "freedom_bootloader"

The linker file needs to be replaced if the memory configuration of the target MCU differs from the closest match. This is done in the linker settings, which is also part of the project options.

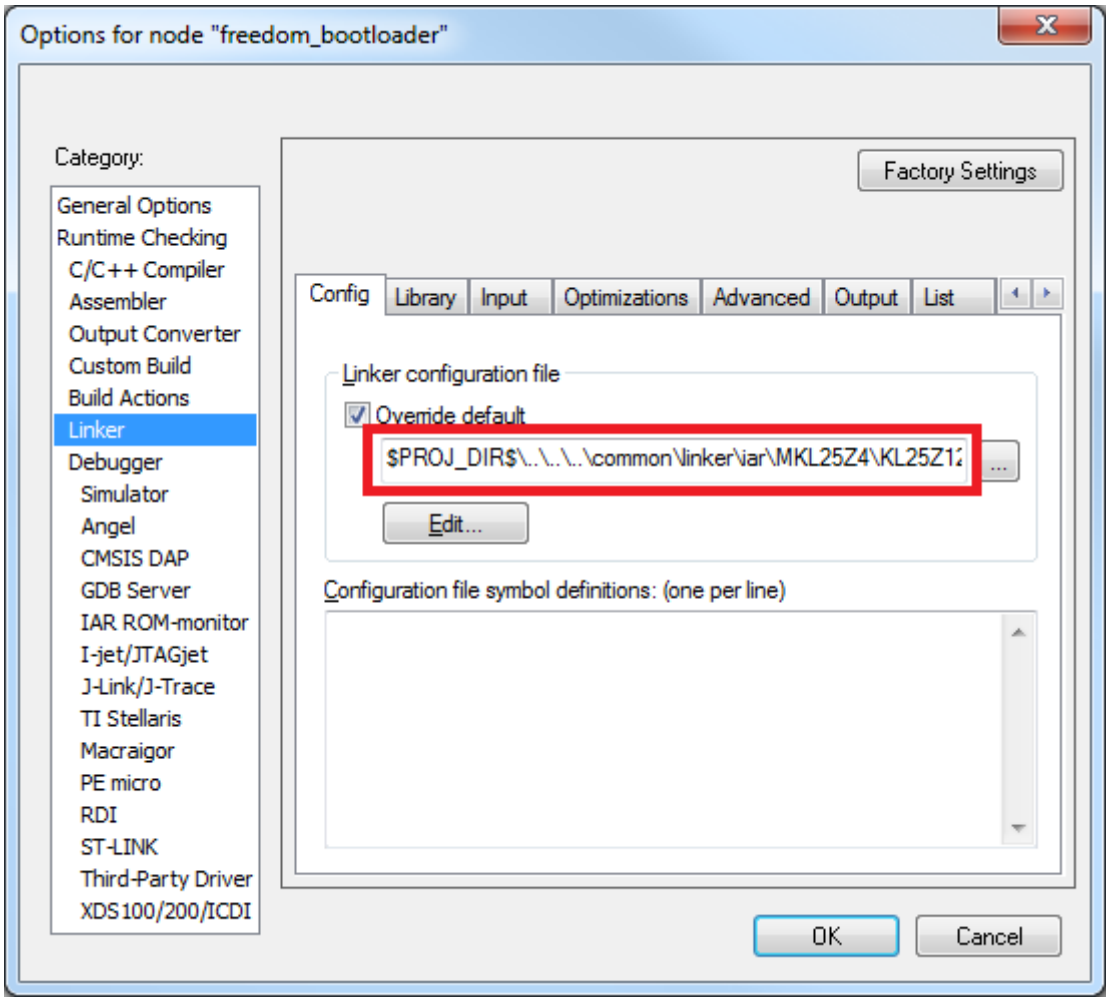

Figure 9-4. Porting guide change linker file

9.4 Primary porting tasks

Actual porting work can begin when the basic file structure and source files are in place. This section describes which files need to be modified and how to modify them.

9.4.1 Header file modification

In section 1.2.1, the Freescale-provided CMSIS header files were downloaded and copied to the bootloader tree. For these header files to be used by the bootloader, the fsl_device_registers.h file in `/src/include/device/src` needs to be modified.

The file is organized by MCU family and points the bootloader to the device-specific header files. A new #elif case needs to be added to the bottom of the list (before the #else that indicates error) that references the target MCU. Note the define used to identify the target MCU must match the define added in section 1.2.5, in Figure 3. With the new section in place, reference the content used for other devices to determine what needs to be added to the new section.

9.4.2 Bootloader peripherals

There are two steps required to enable and configure the desired peripherals on the target MCU:

- Choosing which peripherals can be used by the bootloader.
- Configuring the hardware at a low level to enable access to those peripherals.

9.4.2.1 Supported peripherals

The bootloader uses the peripherals_<device>.c file to define which peripheral interfaces are active in the bootloader. The source file itself includes a single table, `g_peripherals[]`, that contains active peripheral information and pointers to configuration structures. This file is found in `/targets/<device>/src`.

It's important to only place configurations for peripherals that are present on the target MCU. Otherwise, the processor generates fault conditions when trying to initialize a peripheral that is not physically present.

In terms of the content of each entry in the `g_peripherals[]` table, it is recommended to reuse existing entries and only modify the `.instance` member. For example, starting with the following UART0 member, it can be changed to UART1 by simply changing `.instance` from "0" to "1".

```
{
      .typeMask = kPeripheralType_UART,
      .instance = 0,
      .pinmuxConfig = uart_pinmux_config,
      .controlInterface = &g_scuartControlInterface;
      .byteInterface = &g_scuartByteInterfacek;
      .packetInterface = &g_framingPacketInterface;
      }
```

When the table has all required entries, it must be terminated with a null `{ 0 }` entry.

9.4.2.2 Peripheral initialization

Once the desired peripheral configuration has been selected, the low level initialization must be accounted for. The bootloader automatically enables the clock and configures the peripheral, so the only thing required for the port is to tell the bootloader which pins to use for each peripheral. This is handled in the hardware_init_<device>.c file in `/targets/<device>/src`. The hardware_init_<device>.c file also selects the boot pin used by the bootloader, which may need to be changed for the new target MCU.

This file most likely requires significant changes to account for the differences between devices when it comes to pin routing. Each function should be checked for correctness and modified as needed.

9.4.2.3 Clock initialization

The Kinetis bootloader typically uses the MCU's default clock configuration. This is done to avoid dependencies on external components and simplify use. In some situations, the default clock configuration cannot be used due to accuracy requirements of supported peripherals. On devices that have on-chip USB, the default system configuration is not sufficient and the bootloader configures the device to run from the high-precision internal reference clock (IRC48).

The bootloader uses the clock_config_<device>.c file in `/targets/<device>` to override the default clock behavior. If the target MCU of the port supports USB, this file can be used. If not, the functions within clock_config_<device>.c can be stubbed out or set to whatever the port requires.

9.4.3 Bootloader configuration

The bootloader must be configured in terms of the features it supports and the specific memory map for the target device. Features can be turned on or off by using #define statements in the bootloader_config.h file in `/targets/<device>/src`. The supported features can be seen in command.c (g_commandHandlerTable[] table) in the `/src/bootloader/src folder`. All checks that reference a BL_* feature can be turned on or off. Examples of these features are BL_MIN_PROFILE, BL_HAS_MASS_ERASE and BL_FEATURE_READ_MEMORY.

One of the most important bootloader configuration choices is where to set the start address (vector table) of the user application. This is determined by the BL_APP_VECTOR_TABLE_ADDRESS define in bootloader_config.h. Most

bootloader configurations choose to place the user application at address 0xA000 since that accommodates the full featured bootloader image. It's possible to move this start address if the resulting port reduces features (and thus, code size) of the bootloader.

9.4.4 Bootloader memory map configuration

The MCU device memory map and flash configuration must be defined for proper operation of the bootloader. The device memory map is defined in the g_memoryMap[] structure of the memory_map_<device>.c file, which can be found in /targets/<device>/src. An example memory map configuration is shown.

```
memory_map_entry_t g_memoryMap[] =
{
    // Flash array (1024KB)
    { 0x00000000, 0x000fffff, &g_flashMemoryInterface },
    // SRAM (256KB)
    { 0x1fff0000, 0x2002ffff, &g_normalMemoryInterface },
    // AIPS peripherals
    { 0x40000000, 0x4007ffff, &g_deviceMemoryInterface },
    // GPIO
    { 0x400ff000, 0x400fffff, &g_deviceMemoryInterface },
    // M4 private peripherals
    { 0xe0000000, 0xe00fffff, &g_deviceMemoryInterface },
    // Terminator
    { 0 }
};
```

In addition to the device memory map, the bootloader needs information about the specific flash configuration of the target MCU. This includes things such as sector size, features, and FlexRAM.

The fsl_flash_features.h file needs to be modified to provide the bootloader with this information. This file is located in /src/drivers/flash/src. To determine which features the flash on the target MCU supports, utilize the device's reference manual. Many Kinetis devices share similar flash configurations so it may be possible to use an existing flash configuration for the port's target MCU. Use the same CPU define referenced in sections 1.2.5 and 1.3.1 to enable a flash configuration.

The correct flash density and SRAM initialization files must be selected according to the target device. Both of these files are split based on Cortex-M4 and Cortex-M0+ based devices, so the likelihood of having to change them is low. However, if required, the files highlighted in this figure can be replaced with their alternatives.

The flash_densities_k_series.c file is located in /src/drivers/flash/src and it's alternative is flash_densities_kl_series.c, which corresponds to devices with a Cortex-M0+ core.

The sram_init_cm4.c file is located in `/src/memory/src` and it's alternative is sram_init_cm0plus.c.

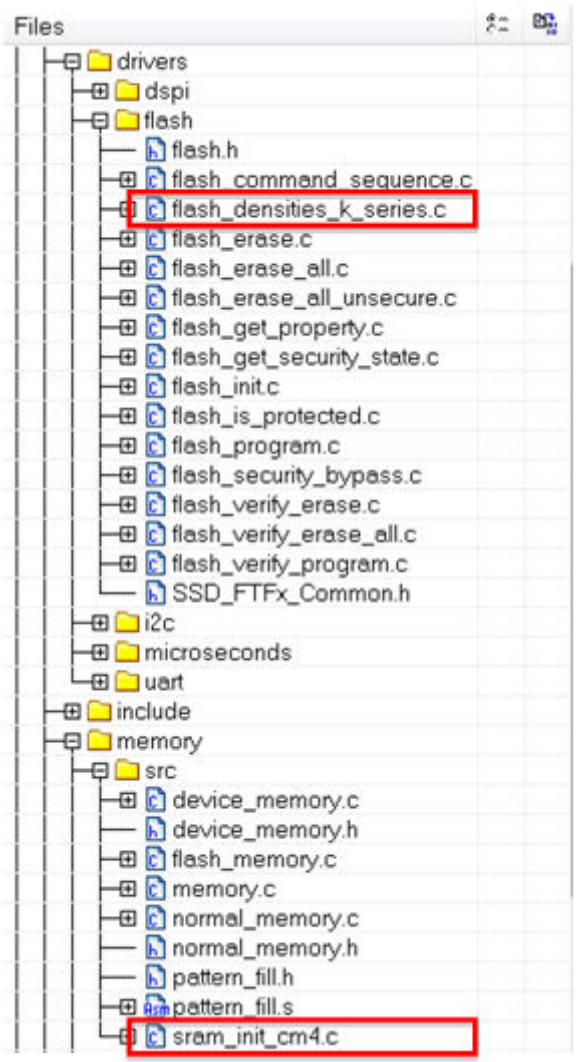

Figure 9-5. Memory map configuration

Chapter 10
Creating a custom flash-resident bootloader

10.1 Introduction

In some situations the ROM-based or full-featured flash-resident bootloader cannot meet the requirements of a use application. Examples of such situations include special signaling requirements on IO, peripherals not supported by the bootloader, or the more basic need to have as small of a code footprint as possible (in the case of the flash-resident bootloader). This section discusses how to customize the flash-resident bootloader for a specific use case.

10.2 Where to start

The Kinetis bootloader package comes with various preconfigured projects, including configurations for a flashloader (if applicable for the device) and a flash-resident bootloader. These projects enable all supported features by default, but can easily be modified to suit the needs of a custom application.

The IAR workspace containing these preconfigured options is located in the <install_dir>/targets/<mcu> folder, where <install_dir> is the folder name of the Kinetis bootloader package once extracted (typically FSL_Kinetis_Bootloader_<version>) and <mcu> is the family of the MCU target. Inside of this folder there is a bootloader.eww file, which is the IAR workspace. The example shows the projects available in the workspace for the K22F512 MCU family. There are configurations for both TWR and FRDM platforms, assuming the boards exist for the specific MCU family.

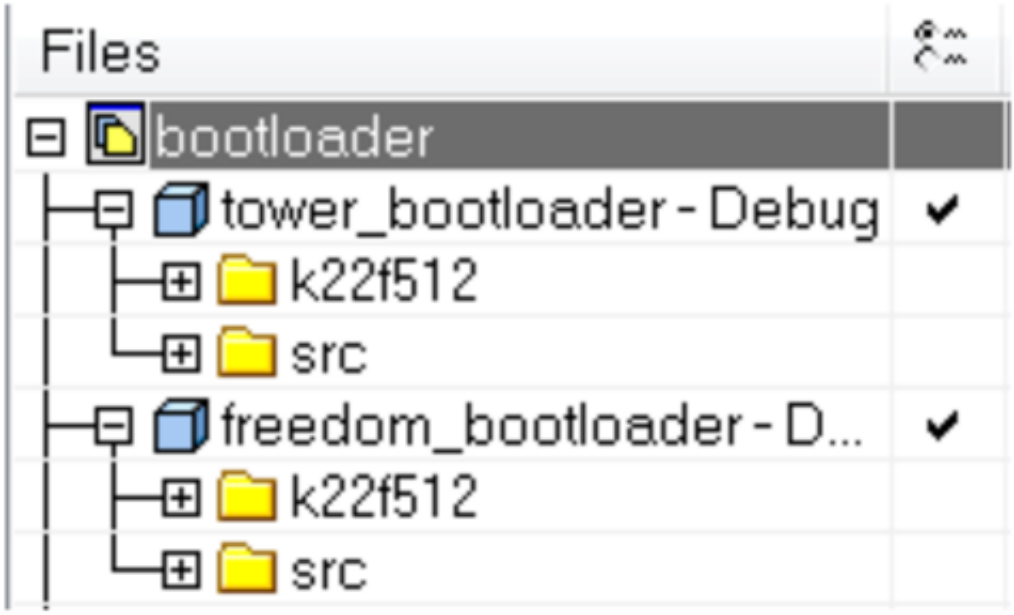

Figure 10-1. Projects available in workspace

Each of the projects in the workspace is configured to support all features of the bootloader. This means every peripheral interface that the MCU supports is enabled. This makes the bootloader very rich in features, but it also has the largest code footprint, which can be considerable on MCUs with smaller flash configurations.

10.3 Flash-resident bootloader source tree

It is important to understand the source tree to understand where modifications are possible. Here is an example of a source treee for one of the bootloader configurations.

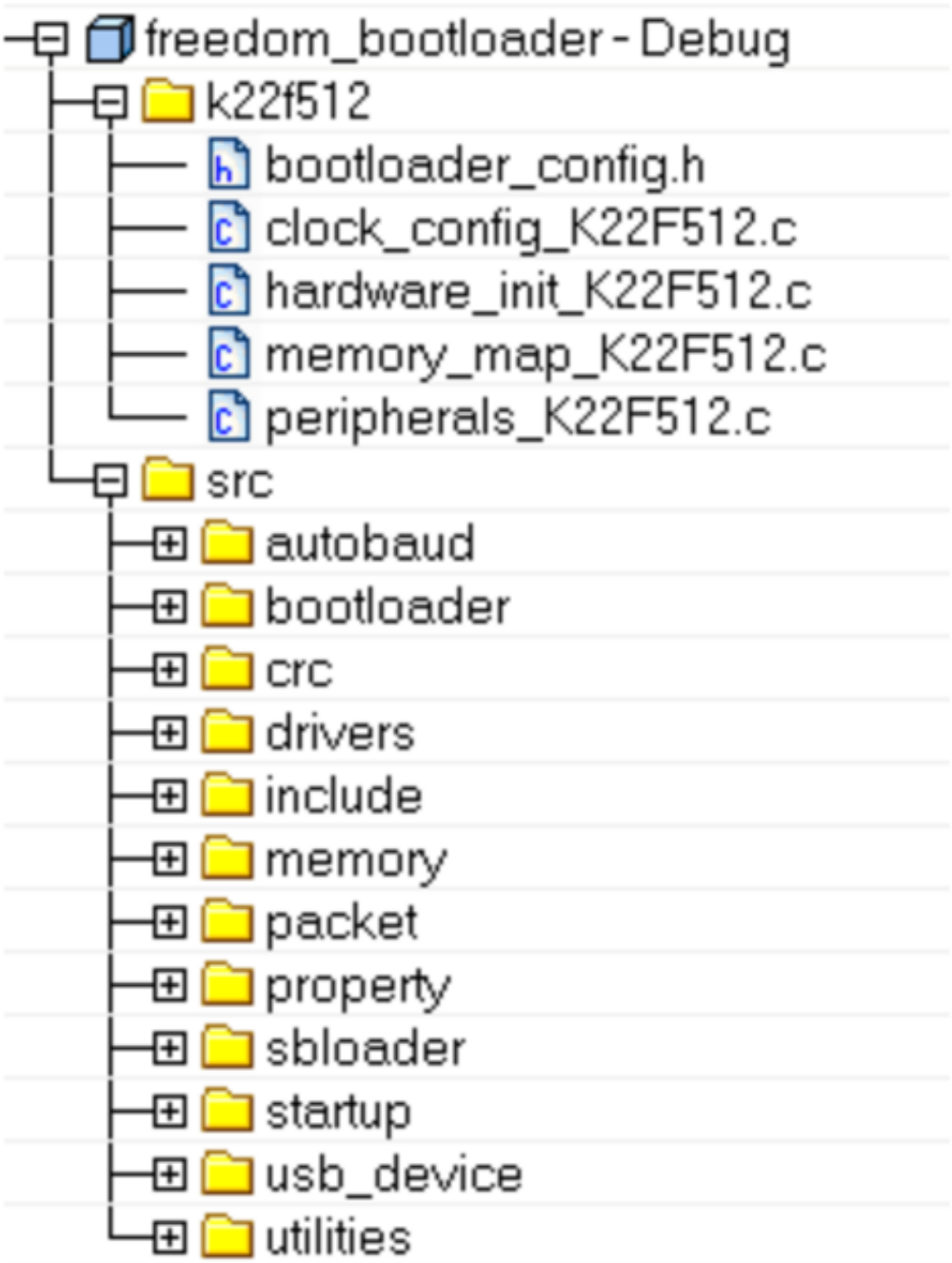

Figure 10-2. Source tree for bootloader configuration

There are two folders in each bootloader project: a MCU-specific folder and a “src” folder. All files in the MCU-specific folder are located in the <install_dir>/targets/<mcu>/src folder, and are very specific to the target MCU. The “src” folder is located at the top level of the bootloader tree, and the subfolders in the project correspond to the real folder/file structure on the PC. The files in the “src” folder are the core files of the bootloader, and include everything from peripheral drivers to individual commands.

The bootloader source is separated in a way that creates a clear line between what a user needs to modify and what they do not. Among other things, the files in the MCU-specific folder allow the application to select which peripherals are active as well as how to configure the clock, and are intended to be modified by the user. The files in the “src” folder can be modified, but should only require modification in cases where very specific customization is needed in the bootloader.

10.4 Modifying source files

The files that cover the majority of the customization options needed by applications are located in the MCU-specific folder. These files allow modification to the basic configuration elements of the bootloader application, and are not associated with the core functionality of the bootloader.

In the MCU-specific folder, the source files contain this information:

- **bootloader_config.h** – Bootloader configuration options such as encryption, timeouts, CRC checking, the UART module number and baud rate, and most importantly, the vector table offset for the user application.
- **clock_config_<mcu>.c** – Configures the clock for the MCU. This includes system, bus, etc.
- **hardware_init_<mcu>.c** – Enables and configures peripherals used by the application. This includes pin muxing, peripheral initialization, and the pin used as a bootloader re-entry (bootstrap) mechanism.
- **memory_map_<mcu>.c** – Contains a table that stores the memory map information for the targeted MCU.
- **peripherals_<mcu>.c** – Contains the table used by the bootloader to check which peripheral interfaces are enabled. This is the file used to disable any unwanted or unused peripheral interfaces.

10.5 Example

One of the most common customizations performed on the Kinetis bootloader is removing unused or unwanted peripheral interfaces. The default configuration of the bootloader enables multiple interfaces, including UART, SPI, I2C and (on some devices) USB. This example will describe how to modify the provided bootloader projects to use only the UART interface. The same methodology can be used to select any of the supported interfaces.

10.6 Modifying the peripherals_<mcu>.c file

The first step required is modifying the g_peripherals table in the peripherals_<mcu>.c file, located in <install_dir>/targets/<mcu>/src. The file only contains a table, defining which peripheral interfaces are used by the bootloader and their specific configuration, including instance. If only one interface is desired, all entries other than the desired interface can be removed.

```
const peripheral_descriptor_t g_peripherals[] =
{
    // UART1
    {
        .typeMask = kPeripheralType_UART,
        .instance = 1,
        .pinmuxConfig = uart_pinmux_config,
        .controlInterface = &g_scuartControlInterface,
        .byteInterface = &g_scuartByteInterface,
        .packetInterface = &g_framingPacketInterface
    },
    { 0 } // Terminator
};
```

The UART instance can be changed here as well. If another UART is desired, simply change the instance variable. The same applies to other peripheral interfaces.

10.7 Removing unused files from the project

To optimize code size, files related to the unusual peripheral interfaces must be removed from the project. The files that should be removed are:

- <peripheral_name>_peripheral_interface.c located under the src/bootloader/src folder.
- Folders for unused drivers, under the src/drivers folder.
- If the USB is not present, simply remove the “usb_device” folder.

Figure 10-3. Removing unused files

With these items complete, simply rebuild the bootloader project. Unused peripheral sources will be excluded from the executable, leaving only the desired interface.

Chapter 11
Revision history

Revision History

Table 11-1. Revision History

Revision number	Date	Substantive changes
Rev. 0	12/2014	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Document Number: KBTLDR110RM
Revision 0, 12/2014