XMC1000

32-bit Microcontroller Series for Industrial Applications

XMC1000 on-chip flash operation and uses

AP32280

Application Note

Scope and purpose

This document provides information on the XMC1000 on-chip flash operation and its application usage. Use this document as a reference when developing a flash application.

Applicable Products

- XMC1000 Microcontrollers Family
- DAVE ™

References

Infineon: DAVE™, http://www.infineon.com/DAVE

Infineon: XMC Family, http://www.infineon.com/XMC

Table of Contents

Table of Contents **2**

1 Flash Driver **4**
1.1 APIs from Flash Driver 4
1.1.1 FLASH003_ErasePage 5
1.1.1.1 Example Code 5
1.1.2 FLASH003_ProgVerifyPage 6
1.1.2.1 Example Code 6
1.1.3 FLASH003_WriteBlock 7
1.1.3.1 Example Code 7
1.1.4 FLASH003_WriteMultipleBlocks 8
1.1.4.1 Example Code 8
1.1.5 FLASH003_ReadWord 9
1.1.5.1 Example Code 9
1.1.6 FLASH003_ReadBlock 10
1.1.6.1 Example Code 10
1.1.7 FLASH003_ReadBytes 11
1.1.7.1 Example Code 11

2 In Application Programming (IAP) **12**
2.1 Implementation 12
2.1.1 Host Program 13
2.1.2 Communication Interface 13
2.2 Firmware Algorithm 14
2.2.1 Read Operation 15
2.2.1.1 Sequence 15
2.2.2 Program Operation 17
2.2.2.1 Sequence 17
2.2.3 Erase Operation 19
2.2.3.1 Sequence 19
2.3 Example Code 21
2.3.1 Read Operation 21
2.3.2 Program Operation 23
2.3.3 Erase Operation 26
2.3.4 Reset Operation 28
2.3.5 UART Receive Byte 28
2.3.6 UART Transmit Byte 28

3 EEPROM Emulation **29**
3.1 Flash Configuration 29
3.1.1 EEPROM Bank 29
3.1.2 EEPROM Data Block 30
3.2 FEE Algorithm 31
3.2.1 Write Data Block 31
3.2.2 Invalidate Data Block 31
3.2.3 Read Data Block 32
3.2.4 Error Handling and System Recovery 32

Table of Contents

3.3 Flash EEPROM Emulation (FEE) Driver 33
3.3.1 FEE001_Init 33
3.3.2 FEE001_GetStatus 34
3.3.2.1 Example Code 34
3.3.3 FEE001_Write 35
3.3.3.1 Example Code 35
3.3.4 FEE001_Read 36
3.3.4.1 Example Code 36
3.3.5 FEE001_ReadBlockWithCRC 37
3.3.5.1 Example Code 37
3.3.6 FEE001_GetPreviousData 38
3.3.6.1 Example Code 38
3.3.7 FEE001_StartGarbageCollection 39
3.3.7.1 Example Code 39
3.3.8 FEE001_InvalidateBlock 40
3.3.8.1 Example Code 40
3.4 Application Example: DALI Control Gear with non-volatile memory 41
3.4.1 FEE001 Configuration 41
3.4.2 Writing DALI variables 43
3.4.3 Reading DALI variables 43

4 Revision History 44

1 Flash Driver

In DAVE™, the flash driver is provided as Flash003 app. This Flash003 app provides APIs for the basic flash operation.

1.1 APIs from Flash Driver

The flash driver provides the following APIs:

- FLASH003_ErasePage
- FLASH003_ProgVerifyPage
- FLASH003_WriteBlock
- FLASH003_WriteMultipleBlocks
- FLASH003_ReadWord
- FLASH003_ReadBlock
- FLASH003_ReadBytes

The minimum size to erase the flash is one page size (256 bytes) and to write data into the flash, the minimum size is one block (16 bytes).

To achieve better data integrity for a write operation, the recommendation is to write the flash in page size by using FLASH003_ProgVerifyPage rather than writing in block size with FLASH003_WriteBlock. This is because FLASH003_ProgVerifyPage provides the mechanism to erase the destination address first if necessary and also provides verification of written data with program buffer after the write operation.

For a read operation, the data can be read out directly from the flash address but the read operation APIs provided include an ECC check, so it is possible to check for the return status of the APIs to know whether any ECC error has been detected.

Note: For more detailed information on the flash driver APIs, please refer to the DAVE App documentation available at DAVE3 IDE under: Help > Helps Content > DAVE Apps > FLASH003.

1.1.1 FLASH003_ErasePage

Table 1 FLASH003_ErasePage

Function	**uint32_t FLASH003_ErasePage (uint32_t Address)**
Parameter	**uint32_t Address** - Destination flash address. Must be page aligned
Return	0x00 - FLASH003_COMPLETE 0x01 - FLASH003_INVALID_PARAM 0x02 - FLASH003_ERROR
Description	This API erases one page size of the flash for a user specific address. To erase a larger flash area, the application must call this API multiple times.

1.1.1.1 Example Code

```
uint32_t EraseAddress;
uint32_t Status;
uint16_t count;

/* This is the flash address to be erased*/
EraseAddress = 0x10003000;

/* Erase for 100 pages */
For(count=0; count < 100; count++)
{
  Status = FLASH003_ErasePage(EraseAddress);
  if(Status == FLASH003_COMPLETE)
  {
  /* Increase to the next page address */
  EraseAddress += 0x100;
  }
}
```

1.1.2 FLASH003_ProgVerifyPage

First loads the data to be written into user buffer and call this API. This API will determine whether the page at the specified destination address is already erased. If the page is partially or fully programmed, the API will erase it and continue to program the new data to the flash. Once the data is written on the flash, it will verify the written data with the data on the program buffer.

Table 2 FLASH003_ProgVerifyPage

Function	**uint32_t FLASH003_ProgVerifyPage (uint32_t Address, const uint32_t pBuf[ ])**
Parameter	**uint32_t Address** - Destination flash address which must be page aligned **const uint32_t pBuf[]** - User buffer for data to be written - Buffer must be at least one page size
Return	0x00 - FLASH003_COMPLETE 0x01 - FLASH003_INVALID_PARAM 0x02 - FLASH003_ERROR
Description	This function erases, writes and verifies the page indicated by the Address parameter.

1.1.2.1 Example Code

```
uint32_t UserBuffer[64];
uint32_t Address = 0x10003000;
uint32_t Status;

/* Write 1 page of data into the flash */
Status = FLASH003_ProgVerifyPage(Address,&UserBuffer[0]);
```

1.1.3 FLASH003_WriteBlock

This API is for writing data in a single block. Note that there is no mechanism to erase the memory first, as there is with the FLASH003_ProgVerifyPage API function. Therefore please ensure that the destination memory block is erased before calling this API.

Attention: Writing data to an already programmed memory address may cause data corruption.

Table 3 FLASH003_WriteBlock

Function	**uint32_t FLASH003_WriteBlock (uint32_t Address, const uint32_t pBuf[ ])**
Parameter	**uint32_t Address** - Destination flash address which must be block aligned **const uint32_t pBuf[]** - User buffer for data to be written - Buffer should be in one block size
Return	0x00 - FLASH003_COMPLETE 0x01 - FLASH003_INVALID_PARAM 0x02 - FLASH003_ERROR
Description	This API allows one block size of data to be written into the flash.

1.1.3.1 Example Code

```
uint32_t UserBuffer[4]={20,21,22,23};
uint32_t Address = 0x10003000;
uint32_t Status;

/* Erase destination address */
Status = FLASH003_ErasePage(Address);

/* Write 1 block of data into the flash */
Status = FLASH003_WriteBlock(Address,UserBuffer);
```

1.1.4 FLASH003_WriteMultipleBlocks

Table 4 FLASH003_WriteMultipleBlocks

Function	**uint32_t FLASH003_WriteMultipleBlocks (uint32_t Address, const uint32_t pBuf[ ], uint32_t No_of_Blocks)**
Parameter	**uint32_t Address** - Destination flash address which must be block aligned **const uint32_t pBuf[ ]** - User buffer for data to be written **uint32_t No_of_Blocks** - Number of blocks to be written into the flash
Return	0x00 - FLASH003_COMPLETE 0x01 - FLASH003_INVALID_PARAM 0x02 - FLASH003_ERROR
Description	This API allows writing multiple blocks with single API call. The number of blocks to be written must be provided when calling this API.

1.1.4.1 Example Code

```
uint32_t UserBuffer[16]={20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35};
uint32_t Address = 0x10003000;
uint32_t Status;

/* Erase destination address */
Status = FLASH003_ErasePage(Address);

/* Write 4 blocks of data into the flash */
Status = FLASH003_WriteMultipleBlocks(Address,UserBuffer,4);
```

1.1.5 FLASH003_ReadWord

Table 5 FLASH003_ReadWord

Function	**uint32_t FLASH003_ReadWord(uint32_t Address, uint32_t* pBuf)**
Parameter	**uint32_t Address** - Source flash address which must be word aligned **uint32_t* pBuf** - User buffer for data read from flash
Return	0x00 - FLASH003_COMPLETE 0x01 - FLASH003_INVALID_PARAM 0x02 - FLASH003_ERROR
Description	This API reads a single data word (32 bit) from the flash with ECC checks.

1.1.5.1 Example Code

```
uint32_t rBuffer;
uint32_t address = 0x10003000;
uint32_t Status;

/* Read 1 word of data from the flash */
Status = FLASH003_ReadWord(address,&rBuffer);
```

1.1.6 FLASH003_ReadBlock

Table 6 FLASH003_ReadBlock

Function	**uint32_t FLASH003_ReadBlock(uint32_t Address, uint32_t pBuf[ ])**
Parameter	**uint32_t Address** - Source flash address which must be block aligned **uint32_t pBuf[ ]** - User buffer for data read from flash - Buffer size needs to be in multiples of the block size, at least one block
Return	0x00 - FLASH003_COMPLETE 0x01 - FLASH003_INVALID_PARAM 0x02 - FLASH003_ERROR
Description	This API reads one block size of data from the flash with ECC check.

1.1.6.1 Example Code

```
uint32_t rBuffer[4];
uint32_t address = 0x10003000;
uint32_t Status;

/* Read 1 block of data from the flash */
Status = FLASH003_ReadBlock(address,rBuffer);
```

1.1.7 FLASH003_ReadBytes

Table 7 FLASH003_ReadBytes

Function	**uint32_t FLASH003_ReadBytes (uint32_t Address, uint8_t pBuf[ ], uint32_t No_of_bytes)**
Parameter	**uint32_t Address** - Destination flash address which must be byte aligned **const uint32_t pBuf[ ]** - User buffer for data to be written **uint32_t No_of_Blocks** - Number of blocks to be read from the flash
Return	0x00 - FLASH003_COMPLETE 0x01 - FLASH003_INVALID_PARAM 0x02 - FLASH003_ERROR
Description	This API allows reading multiple data in byte size from the flash with ECC check.

1.1.7.1 Example Code

```
uint8_t rBuffer[16];
uint32_t address = 0x10003000;
uint32_t bytes = 16;
uint32_t Status;

/* Read 16 bytes of data from the flash */
Status = FLASH003_ReadBytes(address,rBuffer,bytes);
```

2 In Application Programming (IAP)

The IAP example project shown in this application note can be used as a reference for upgrading firmware. It utilizes the functions in the flash driver app (FLASH003) provided in DAVE™, to implement the firmware upgrade application.

A PC Host is required to download the new firmware or system application code into the microcontroller and the IAP firmware will program that data into the flash. After new firmware has been programmed into the microcontroller's flash, the PC Host resets the microcontroller to run the new firmware.

2.1 Implementation

The IAP firmware must:

- be implemented separately from the system application code
- be programmed at an address where the system application code will not overwrite it.
  - For example, the IAP firmware is programmed at address 0x10030000, while the system application code is programmed at address 0x10001000 to 0x1002FFFF.

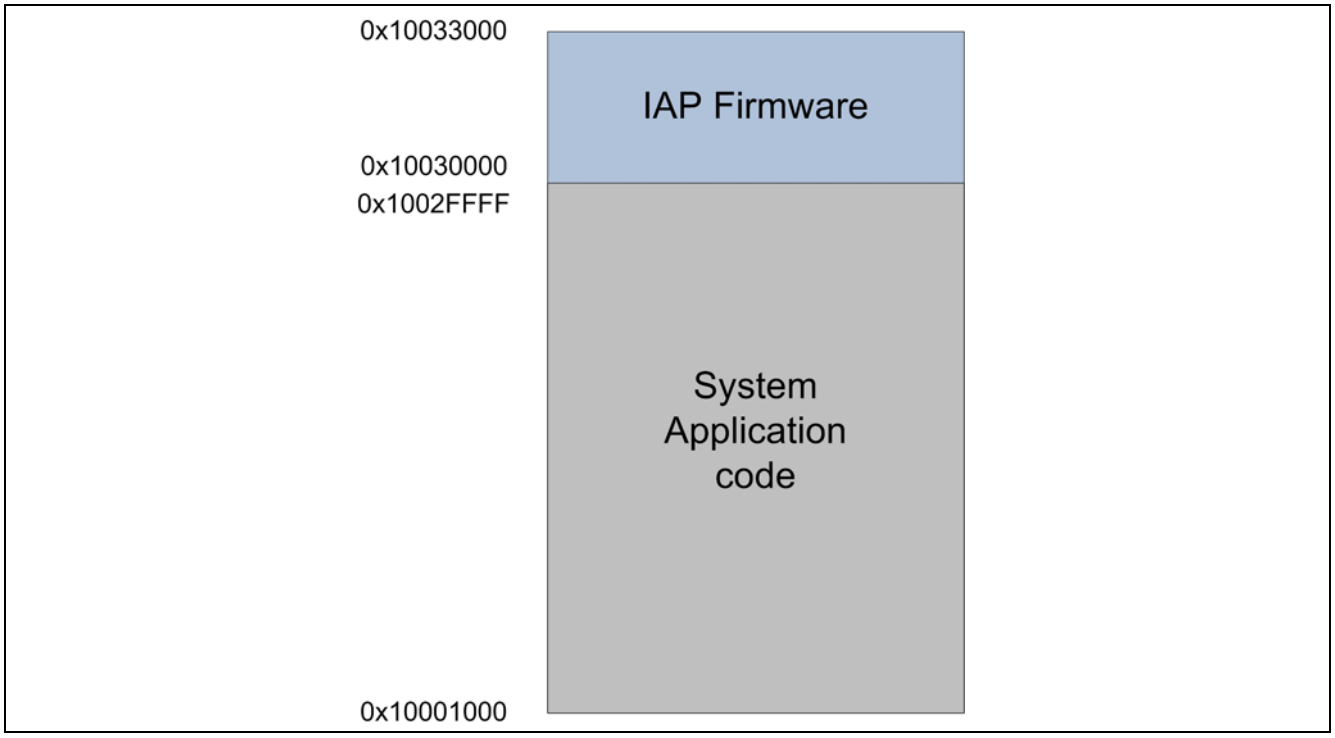

Figure 1 Memory allocation for IAP firmware and system application code

In the system application, a mechanism must be available to trigger the execution of the IAP application. This could be for example, an edge trigger from a GPIO pin to generate an interrupt, where this interrupt will set the microcontroller state to IAP mode. This will trigger the system application to jump to the IAP firmware and start the IAP application. Then, the IAP firmware will receive commands from the Host program and execute the necessary action.

The IAP example provided in this guide details the operations, such as reading the flash content, erasing the flash, programming the flash, and the microcontroller reset.

2.1.1 Host Program

The PC Host program used in this example is simulated with the Docklight terminal program (**http://www.docklight.de/**) running on a computer with a Windows 7 Operating System. All the required commands and the firmware data has to be pre-loaded on the Docklight terminal program.

The Host program will communicate with the microcontroller via the virtual COM Port provided by the Bootkit J-link on-board debugger.

2.1.2 Communication Interface

In this example, UART is used as the communication interface between the Host program and the microcontroller.

On the microcontroller, channel 1 of USIC 0 is configured as UART protocol with P1.3 as receive pin and P1.2 as transmit pin.

The UART configuration is:

- Full duplex communication
- 19200 baud
- 8 data bit
- 1 stop bit
- No Parity

2.2 Firmware Algorithm

For this example, the IAP firmware is programmed at the address starting 0x1002F000.

An external interrupt is used as the trigger mechanism to trigger the execution of the IAP application. When the microcontroller detects a low to high edge on the GPIO, which is configured as external interrupt, the interrupt will set a flag to indicate IAP has triggered. Then the system application will jump to the IAP application located at address 0x1002F000 to execute the IAP application. When the IAP application starts running, it will wait for the start command from the PC Host.

To begin the IAP session, the Host should send the start command (0x00) and the microcontroller will send with the IAP_ID (0x5A) command in return. After that, the Host can choose the operation it wants to execute. The operation includes read data, program data, erase data, and reset operation.

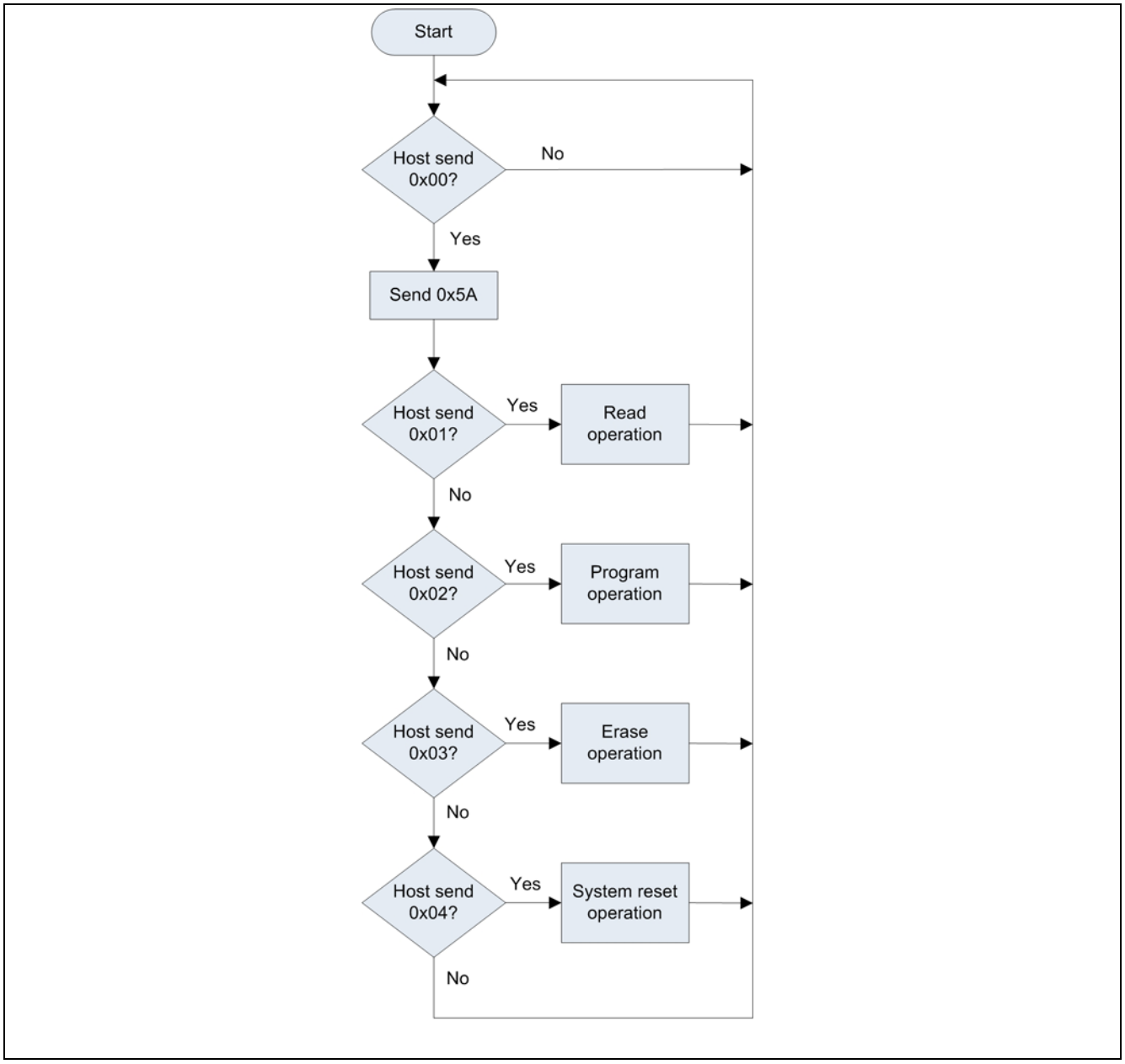

Figure 2 Operation selection flowchart

2.2.1 Read Operation

This operation reads the content in the flash memory, based on the specified data length and the flash address.

The address and data length are to be sent reading from most significant byte to the least significant byte.

For example:

- address 0x10002000
  - sent in the sequence 0x10, 0x00, 0x20, 0x00
- length 300 bytes
  - sent in the sequence 0x00, 0x01, 0x2C

2.2.1.1 Sequence

- After the Host has received the IAP_ID from the microcontroller to start the read operation, the Host sends the value 0x01.
- The microcontroller replies IAP_ACK_READ (0xA1) after receiving 0x01 from the Host.
- The Host sends 4 bytes of data as the start address
- The microcontroller replies:
  - IAP_OK (0xAF) if the address is within the range.
  - IAP_NOK (0xFA) if the address is not within the range. Execution ends.
- The Host send 3 bytes of data to specify the length, in bytes, to be read.
- The microcontroller replies either:
  - IAP_OK (0xAF) if the length is within the range.
  - IAP_NOK (0xFA) if the length is not within the range.
- If the length is within the range, the microcontroller reads the data from the flash and transmits it to the Host.

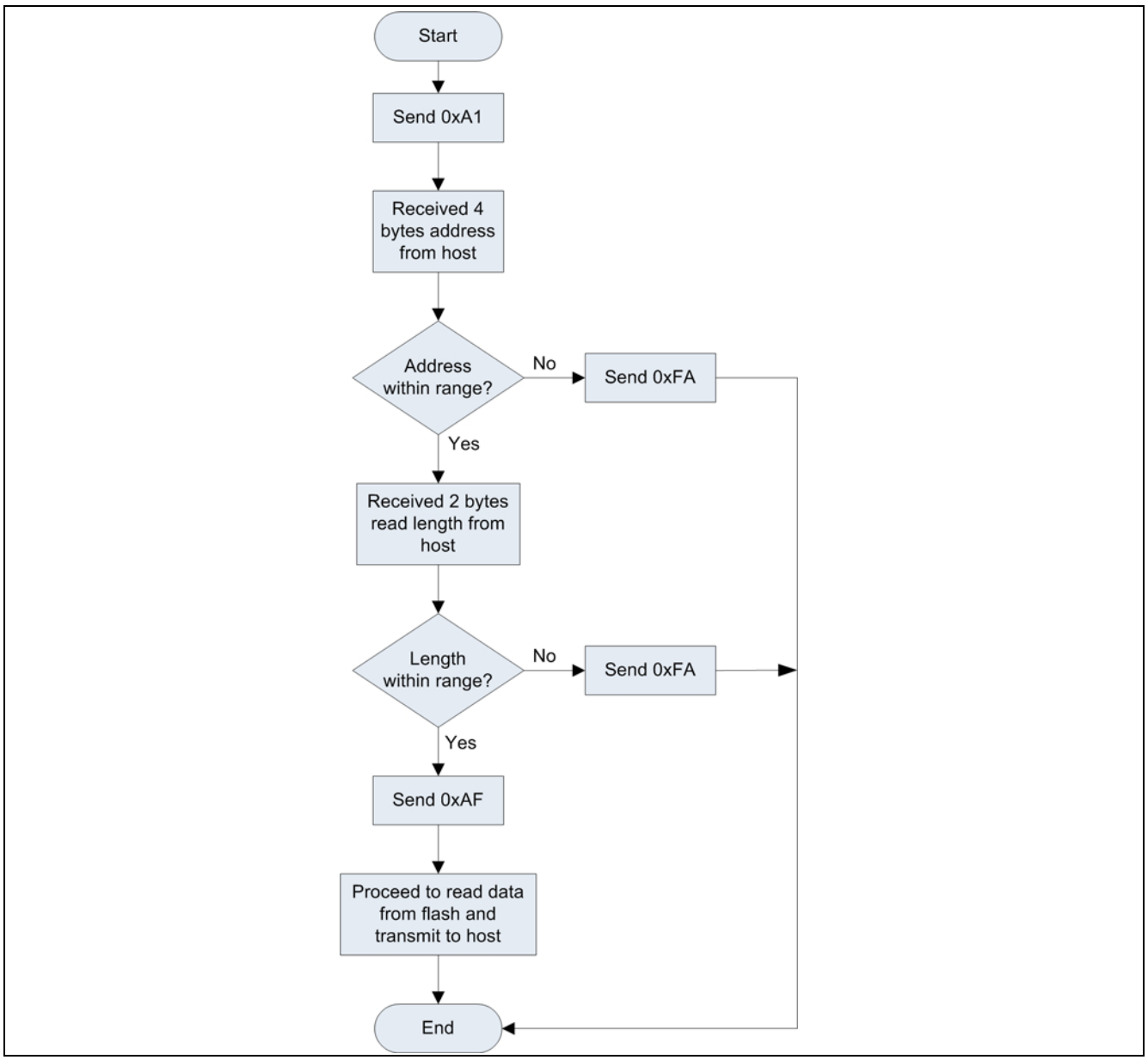

Figure 3 Read operation flowchart

2.2.2 Program Operation

Program operation is used to program data into the microcontroller flash.

The address and size must be sent from most significant byte to least significant byte.

This operation uses the API FLASH003_ProgVerifyPage from the flash driver and will program 256 bytes each time. Data from the Host must therefore be broken down into 256 bytes segments.

The program operation is successful when all data has been received and programmed into the flash.

If operation is not successful for any reason, the IAP firmware will return IAP_NOK and end the session.

2.2.2.1 Sequence

- After the Host has received the IAP_ID from the microcontroller, the value 0x02 is sent to start the program operation.
- The microcontroller replies IAP_ACK_PROGRAM (0xA2) after receiving 0x02 from the Host.
- The Host sends 4 bytes of data specifying the destination address to be programmed.
- The microcontroller replies:
  - IAP_OK (0xAF) if the address is within the range.
  - IAP_NOK (0xFA) if the address is not within the range. Execution ends.
- The Host sends 3 bytes of data to indicate the total size of data to be programmed in bytes.
- The microcontroller replies:
  - IAP_OK (0xAF) if the address is within the range.
  - IAP_NOK (0xFA) if the address is not within the range. Execution ends.
- If the Host received IAP_OK, it will start sending the first 256 bytes of data.
- For every 256 bytes of data received by the microcontroller, it will program the data into the flash and send IAP_OK to the Host.
- The Host will wait for IAP_OK before sending the next 256 bytes of data.
- The sequence for sending data is repeated until all data is sent from the Host.
- When the microcontroller has received and programmed all data from the Host into the flash, it will reply IAP_ACK_PROGRAM to indicate the end of operation.

Note: The Host program is required to read back the data written to the flash after the Program operation, in order to verify the data has been correctly written to the flash. No error handling is provided in this example.

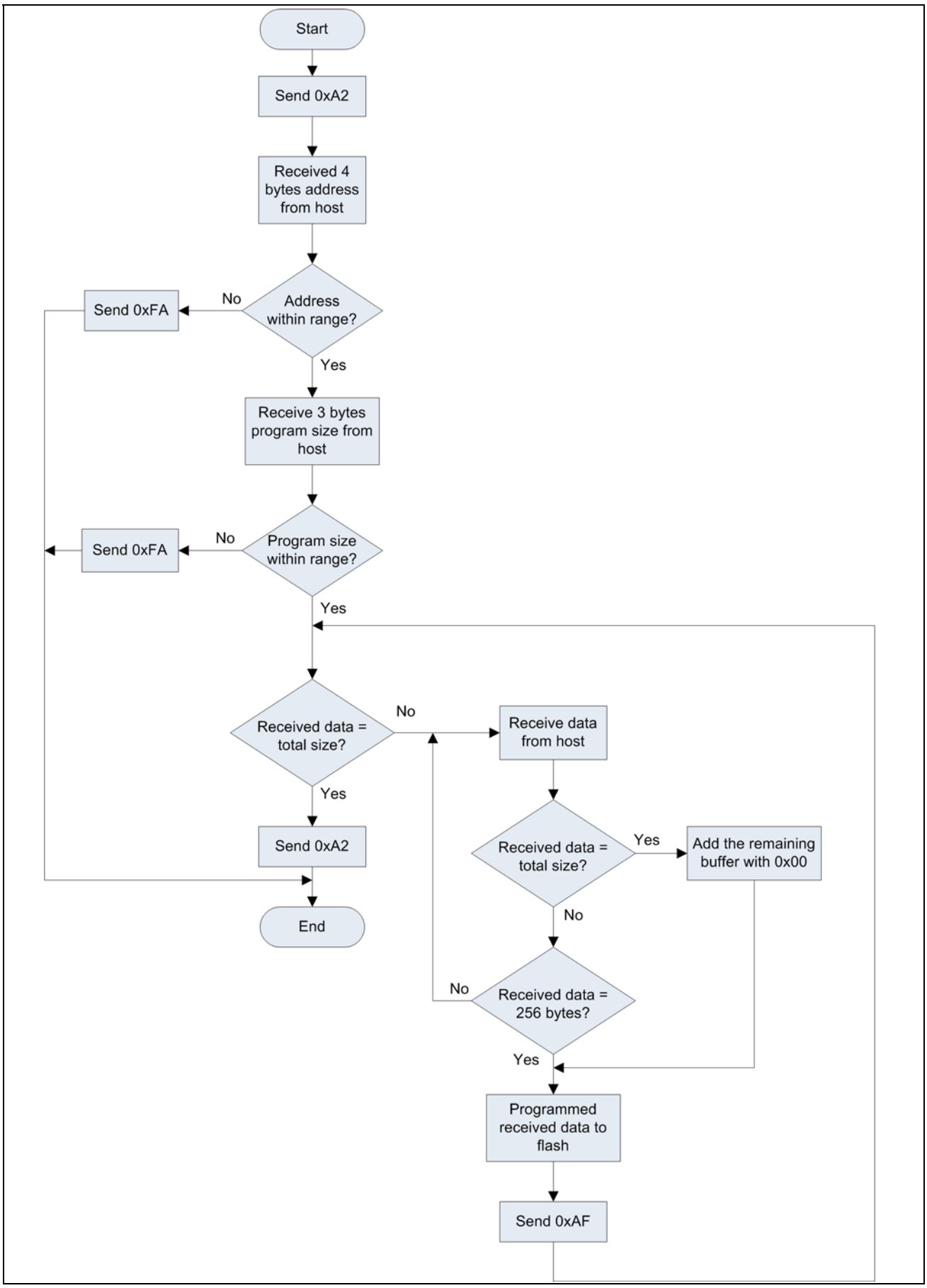
Start
Send 0xA2
Received 4 bytes address from host
Address within range?
No
Send 0xFA
Yes
Receive 3 bytes program size from host
Program size within range?
No
Send 0xFA
Yes
Received data = total size?
No
Receive data from host
Yes
Send 0xA2
Received data = total size?
Yes
Add the remaining buffer with 0x00
No
End
Received data = 256 bytes?
No
Yes
Programmed received data to flash
Send 0xAF

Figure 4 Program operation flowchart

2.2.3 Erase Operation

This operation will erase the content in the microcontroller flash at a specified address and length. The address and length must be sent from the most significant byte to least significant byte. Erase uses the Flash003_ErasePage API from the flash driver, so the size for each erase is a page size and the address must be page aligned.

2.2.3.1 Sequence

- After the Host has received IAP_ID from the microcontroller, it sends the value 0x03 to start the erase operation.
- The microcontroller replies IAP_ACK_ERASE (0xA3) after 0x03 is received from the Host.
- The Host sends 4 bytes of data to the specified start address.
- The microcontroller replies:
  - IAP_OK (0xAF) if the address is within the range.
  - IAP_NOK (0xFA) if the address is not within the range. Execution ends.
- The Host sends 2 bytes of data to indicate the number of pages to be erased.
- The microcontroller replies:
  - IAP_OK (0xAF) if the length is within the range and starts erasing the flash.
  - IAP_NOK (0xFA) if the length is not within the range. Execution ends.
- After erasing the required area, the microcontroller sends IAP_ACK_ERASE (0xA3) to indicate end of operation.

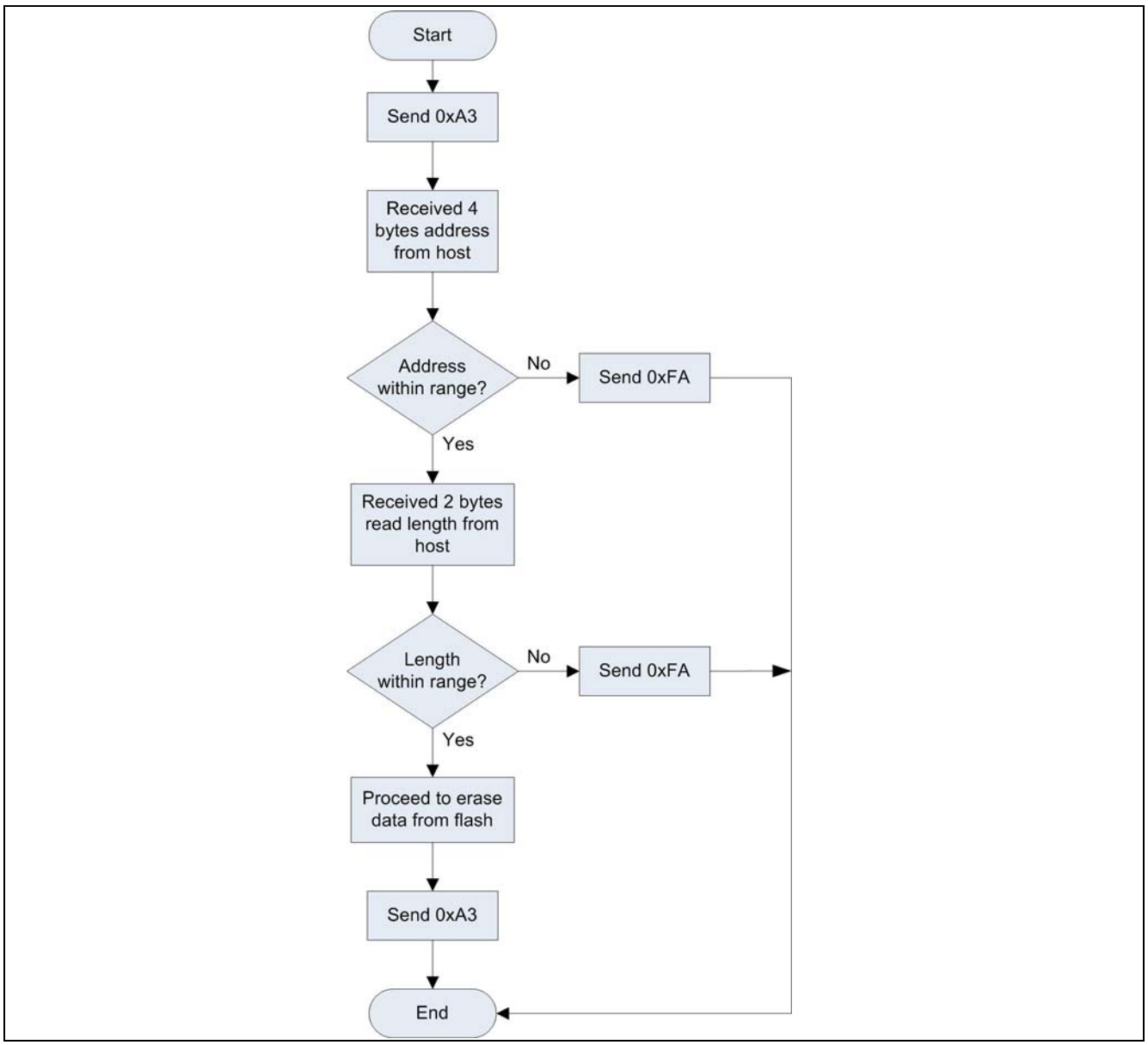

Figure 5 Erase operation flowchart

2.3 Example Code

*Attention: **The following example code is ONLY provided to illustrate how a flash driver can be used in an IAP application. The example code is not intended to be used in a 'real' application as is, and is therefore used entirely at the user's own risk.***

2.3.1 Read Operation

```
uint8_t IAP_ReadFlash (void)
{
	uint32_t Buffer[4] = {0,0,0,0};
	uint32_t Address;
	uint32_t Length;
	uint8_t c;
	uint8_t rBuffer[1];


	/* Receive 4 bytes address from Host */
	Buffer[0] = getMessage();
	Buffer[1] = getMessage();
	Buffer[2] = getMessage();
	Buffer[3] = getMessage();
	for(c=0;c<4;c++)
	{
		Address <<= 8;
		Address |= Buffer[c];
	}

	/* Check whether the address is within the range */
	if((Address >= 0x10001000)  && (Address <= 0x10032FF0))
	{
		ASC_vSendData(IAP_OK);
	}
	else
	{
		ASC_vSendData(IAP_NOK);
		return IAP_ERROR;
	}

	/* Receive 2 bytes length from Host */
	Buffer[0] = getMessage();
	Buffer[1] = getMessage();

	Length = Buffer[0];
	Length <<= 8;
```

```
	Length |= Buffer[1];

	Length = Address + Length;

	/* Check whether the length size is within the range */
	if(Length > 0x10300000)
	{
		ASC_vSendData(IAP_NOK);
		return IAP_ERROR;
	}
	else
	{
		ASC_vSendData(IAP_OK);
	}

	/* Read data and transmit to Host */
	while(Address <= Length)
	{
		FLASH003_ReadBytes(Address,rBuffer,1);
		Address = Address + 1;
		ASC_vSendData(rBuffer[0]);
	}

	return 0;

}// End of read operation
```

2.3.2 Program Operation

```
uint8_t IAP_ProgramFlash (void)
{
	status_t status;
	uint32_t Buffer[4];
	uint32_t Address;
	uint32_t Length;
	uint32_t LengthCheck;
	uint16_t c,d;
	uint8_t *AddressPtr;

	/* Receive 4 bytes address from Host */
	Buffer[0] = getMessage();
	Buffer[1] = getMessage();
	Buffer[2] = getMessage();
	Buffer[3] = getMessage();

	for(c=0;c<4;c++)
	{
		Address <<= 8;
		Address |= Buffer[c];
	}

	/* Check whether the address is within the range */
	if((Address >= 0x10001000)  && (Address <= 0x10030000))
	{
		ASC_vSendData(IAP_OK);
	}
	else
	{
		ASC_vSendData(IAP_NOK);
		return IAP_ERROR;
	}

	/* Receive 3 bytes for firmware length */
	Buffer[0] = getMessage();
	Buffer[1] = getMessage();
	Buffer[2] = getMessage();

	Length = Buffer[0];
	Length <<= 8;
	Length |= Buffer[1];
	Length <<= 8;
```

```
Length |= Buffer[2];

LengthCheck = Address + Length;

/* Check whether the length size is within the range */
if(LengthCheck > 0x10300000)
{
      ASC_vSendData(IAP_NOK);
      return IAP_ERROR;
}
else
{
      ASC_vSendData(IAP_OK);
}

d = 0;

/* Start received new firmware data from Host */
while(d < Length)
{
      AddressPtr = (uint8_t *)&UserData;
      for(c=0;c<256;c++)
      {
            *AddressPtr = getMessage();
            AddressPtr++;
            d++;

            if(d >=Length)
            {
                  while(c !=256)
                  {
                        UserData[c]= 0x00;
                        c++;
                  }

            }
      }

      /* Programmed the received data into flash */
      status = FLASH003_ProgVerifyPage(Address,&UserData[0]);
      if(status == FLASH003_COMPLETE)
      {
            ASC_vSendData(IAP_OK);
      }
```

```
            else
            {
                  ASC_vSendData(IAP_NOK);
                  return IAP_ERROR;
            }

            Address = Address + 256;
      }

      ASC_vSendData(IAP_ACK_PROGRAM);

      return 0;

} // End of program operation
```

2.3.3 Erase Operation

```
uint8_t IAP_EraseFlash (void)
{

	uint32_t Buffer[4] = {0};
	uint32_t Address;
	uint32_t Length;
	uint32_t LengthCheck;
	uint8_t c;

	/* Receive 4 bytes address from Host */
	Buffer[0] = getMessage();
	Buffer[1] = getMessage();
	Buffer[2] = getMessage();
	Buffer[3] = getMessage();
	for(c=0;c<4;c++)
	{
		Address <<= 8;
		Address |= Buffer[c];
	}

	/* Check whether the address is within the range */
	if((Address >= 0x10001000)  && (Address <= 0x10032FF0))
	{
		ASC_vSendData(IAP_OK);
	}
	else
	{
		ASC_vSendData(IAP_NOK);
		return IAP_ERROR;
	}

	/* Receive 2 bytes page size from Host */
	Buffer[0] = getMessage();
	Buffer[1] = getMessage();

	Length = Buffer[0];
	Length <<= 8;
	Length |= Buffer[1];

	/* Convert from page size to byte size*/
	LengthCheck = Length * 0x100;
	LengthCheck = Address + Length;
```

```
	/* Check whether the length size is within the range */
	if(LengthCheck > 0x10300000)
	{
		ASC_vSendData(IAP_NOK);
		return IAP_ERROR;
	}
	else
	{
		ASC_vSendData(IAP_OK);
	}


	/* Start flash erase execution */
	for(c=0; c<Length; c++)
	{
		FLASH003_ErasePage(Address);
		Address = Address + 256;
	}

	ASC_vSendData(IAP_ACK_ERASE);

return 0;

} // End of erase operation
```

2.3.4 Reset Operation

```
void IAP_SWReset (void)
{
	/* Set the MCU master reset bit */
	SCU_RESET->RSTCON |= SCU_RESET_RSTCON_MRSTEN_Msk;

} // End of SW reset operation
```

2.3.5 UART Receive Byte

```
uint8_t getMessage(void)
{
	uint8_t Readdata;

	while((UART001_GetFlagStatus(&UART001_Handle0,
		UART001_FIFO_STD_RECV_BUF_FLAG))!=UART001_SET);

	UART001_ClearFlag(&UART001_Handle0,UART001_FIFO_STD_RECV_BUF_FLAG);

	Readdata = UART001_ReadData(UART001_Handle0);

	return Readdata;
}
```

2.3.6 UART Transmit Byte

```
void ASC_vSendData (uint32_t uwData)
{
	UART001_WriteData(UART001_Handle0, uwData);

	while((UART001_GetFlagStatus(&UART001_Handle0,
		UART001_TRANS_BUFFER_IND_FLAG))!=UART001_SET);

	UART001_ClearFlag(&UART001_Handle0,UART001_TRANS_BUFFER_IND_FLAG);
}
```

3 EEPROM Emulation

Many applications require the frequent storage or update of data on to non-volatile memory during run-time. EEPROM memory can be used for this, but it will increase the overall cost of material. An alternative is to use the microcontroller on-chip flash memory to works as flash emulated EEPROM. DAVE3 provides an App for EEPROM emulation called FEE001.

As will be demonstrated with an example later in this section, the FEE001 App offers a very fast and easy implementation of EEPROM emulation. It is only necessary to add the FEE001 App into your own project to make the EEPROM emulation driver available. With EEPROM emulation, data that is required to be retained upon the next power up can be stored and retrieved easily.

3.1 Flash Configuration

3.1.1 EEPROM Bank

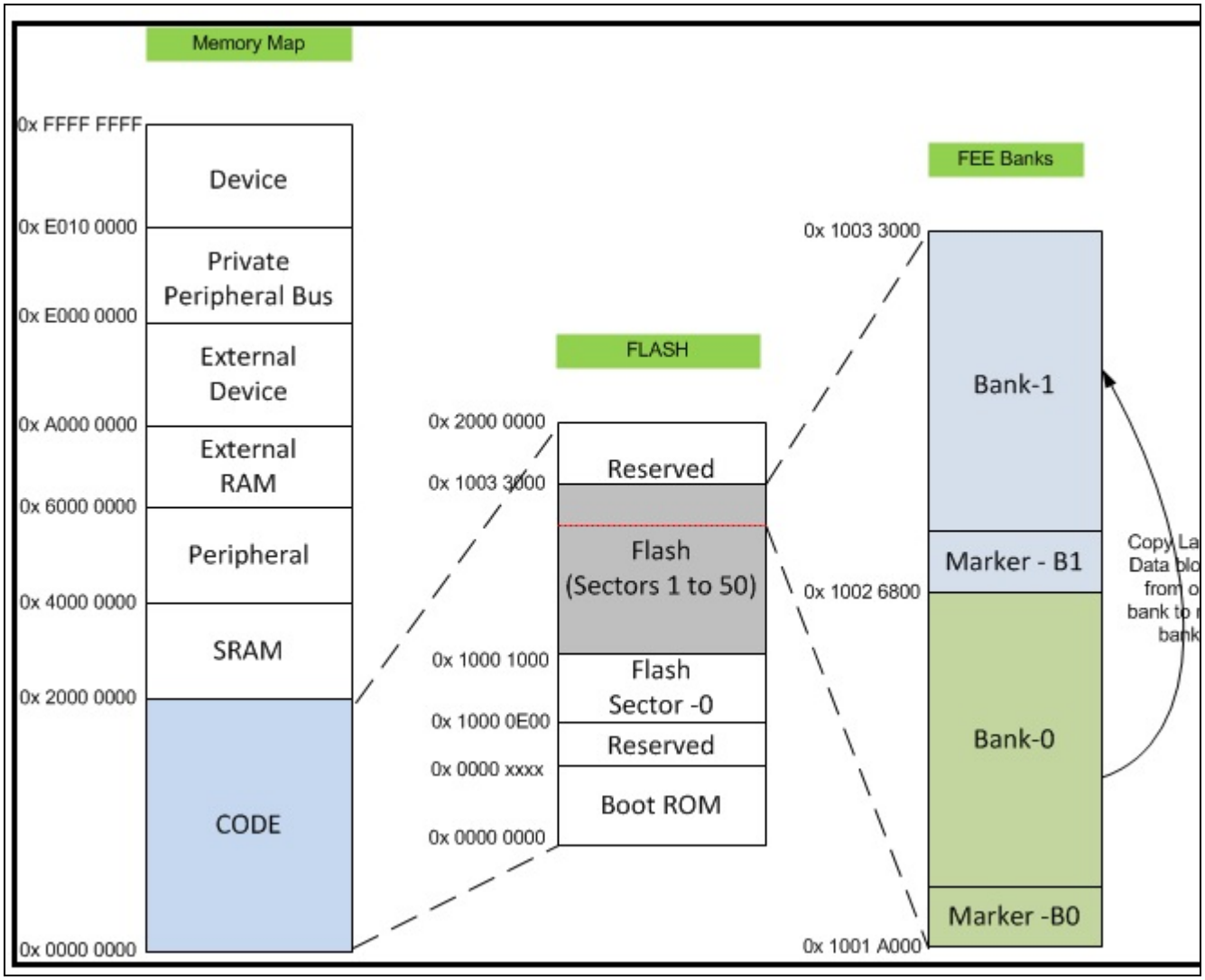

Figure 6 EEPROM emulation area in Flash

The data banks for EEPROM emulation will be located at the end of the last flash address as shown in the figure above. The flash size allocated for EEPROM emulation is user configurable via the FEE001 app User Interface (UI editor). The total size is then divided equally into two data banks and each data bank has a state marker programmed at the start of each bank.

3.1.2 EEPROM Data Block

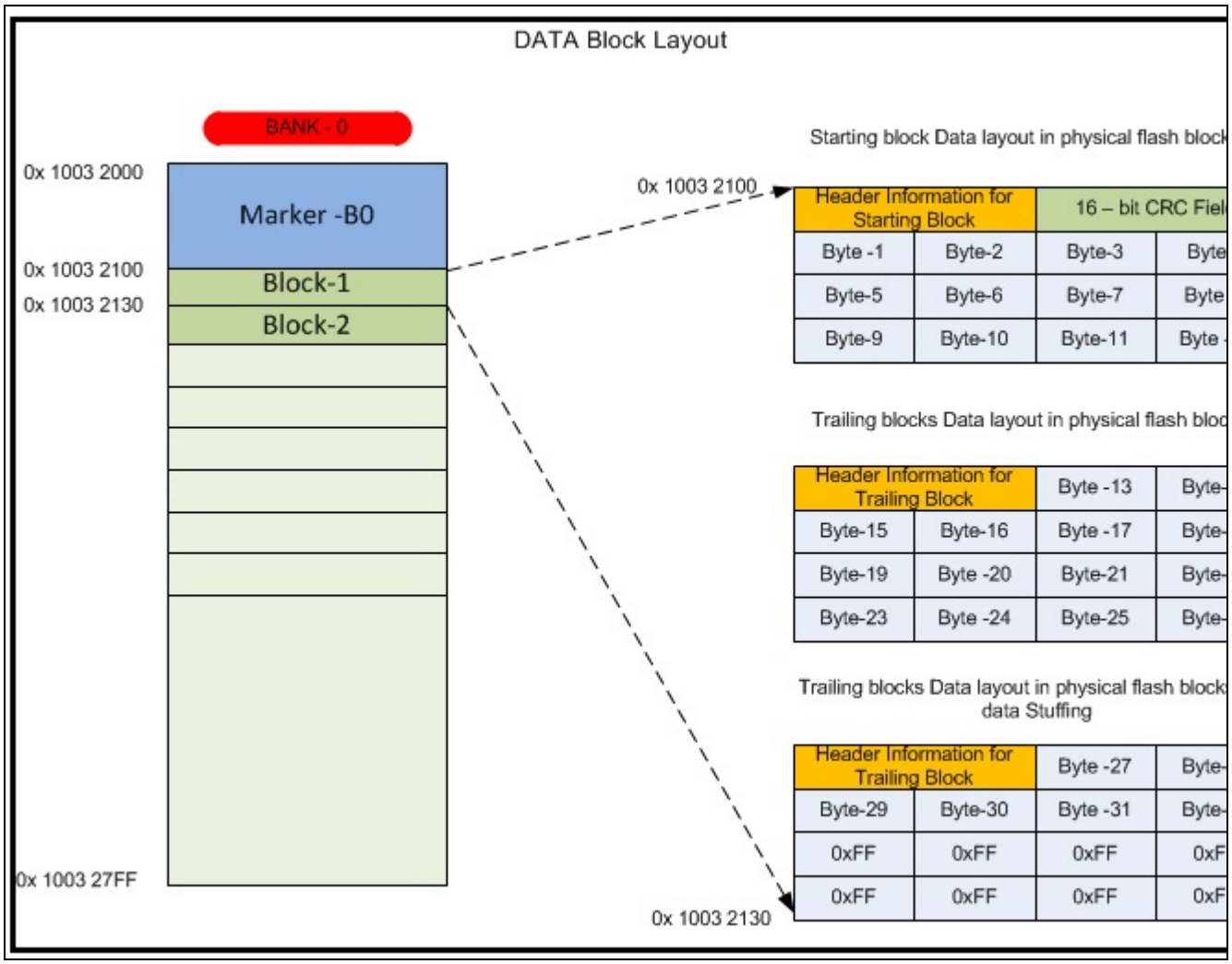

Figure 7 EEPROM data blocks structure

Data blocks are blocks that contain user data. Users can configure up to 10 data blocks with different sizes in the FEE001 app. Each data block also has a user defined numerical block ID.

The size of data blocks are in multiples of 16 bytes block. Each data block includes a 2 byte header.

The starting block includes an additional 2 bytes of CRC checksum data. The CRC checksum is calculated upon user configuration. If CRC is disabled, a dummy value is loaded into this two byte location.

The physical data blocks are specified as:

- Start Block = 2 Bytes Header + 2 Bytes CRC and 12 Bytes of Data
- Trail Blocks = 2 Bytes Header and 14 Bytes of Data

3.2 FEE Algorithm

The Flash EEPROM Emulation (FEE) driver uses a 'Double Bank Fixed State Marker' approach for the EEPROM emulation algorithm. The flash memory allocated for the EEPROM emulation will be divided into two equal size banks. When the EEPROM emulation starts on a fresh device, it will erase the flash address range allocated for both data banks and program the state marker on the first page of each bank. The state marker indicates the states of the data banks. It is used for system recovery when a power failure occurs. During startup, the FEE driver checks the state marker to determine which data bank has the latest valid data blocks.

At any point of time, only one data bank will be active.

- When a write request is triggered, the data block will be written into the active bank and the data bank address is automatically incremented.
- When the data bank is full and a new write request is triggered, the latest user data blocks in the data bank will be copied to the "new" data bank together with the data block from the new write request.
- The "old" data bank is erased after the completion of the copy process. This process is called "garbage collection" and it will be repeated again when the data bank is full.

To provide the flexibility for a user application to control the garbage collection process, garbage collection can be disabled and triggered manually by the application. However, if garbage collection is disabled and the data bank is full, new write requests to the emulated EEPROM will be aborted. New write requests will only get accepted after garbage collection has been executed.

3.2.1 Write Data Block

To store data into the emulated EEPROM, it is necessary to trigger a write request via the FEE001_Write API. The required input is the block ID and the buffer location of the data to be stored when calling the FEE001_Write function. If the CRC feature is enabled, the CRC value for the complete data block will be calculated and the value will be written in the CRC field of the starting block. Then the whole data block will be written into the data bank.

When there is insufficient space left to accept the new write request, the garbage collection process is triggered. The write request will only be accepted and be written into the flash after the garbage collection is completed. Therefore, the auto triggering of the garbage collection process will happen on a user write request.

3.2.2 Invalidate Data Block

Data blocks that are no longer required can be invalidated by calling the FEE001_InvalidateBlock API.

The invalidation process will write a single block of 16 bytes with all data bytes programmed to 0xFF, and a header with an Invalid state. Any future read operation on this block will result in Invalid data. Nevertheless, the invalidated data block still gets copied to the new data bank during the garbage collection process to keep track of the history of the previous operation.

3.2.3 Read Data Block

A data block is read with the FEE001_Read API. The length of data to be read out has to be specified. Users also have to provide the buffer location that will hold the data that is read from the emulated EEPROM. i.e. the data requested will be fetched from the emulated EEPROM and copied to the user's buffer. If the CRC feature is enabled, reading the data with a CRC check has to be made with FEE001_ReadBlockWithCRC API. For this API, the whole data block will be fetched from the EEPROM to selected buffer. The CRC of the data block will be calculated and compared with the CRC value that was written previously with the data block into the emulated EEPROM.

3.2.4 Error Handling and System Recovery

If there is power failure or if a system reset occurs, the state machine in the FEE driver will be interrupted and the data could be corrupted. Thanks to the state marker available on each data bank, the correct state of the FEE driver can be recovered during the initialization of the FEE driver. This allows the FEE driver to determine which data bank contains the latest data. The correct state can also be determined by the state marker if the power failure or system reset occurs during garbage collection. Therefore the garbage collection process can continue its operation after the system is restarted.

If a write operation is in progress when the power failure occurs, the data programmed would be incomplete and that data is therefore considered as invalid. In this case, the previous correctly written data block will be used as the latest data block.

Should the state marker get corrupted beyond recovery, both data banks will be erased and the EEPROM emulation will be start again, if the Erase All feature is enabled. However, if the Erase All feature is not enabled, data from both data banks will be preserved but EEPROM emulation is not possible until both banks are erased manually.

3.3 Flash EEPROM Emulation (FEE) Driver

Flash EEPROM emulation driver is provided in DAVE™ as FEE001 app.

The FEE001 app provides the following APIs:

- FEE001_Init
- FEE001_Write
- FEE001_Read
- FEE001_ReadBlockWithCRC
- FEE001_GetPreviousData
- FEE001_StartGarbageCollection
- FEE001_InvalidateBlock
- FEE001_GetStatus

3.3.1 FEE001_Init

This will be called within DAVE_Init(), so users are not required to call this API.

However, FEE001_GetStatus() shall be called after DAVE_Init() in order to get the FEE initialization status.

Table 8 FEE001_Init

Function	**void FEE001_Write (void)**
Parameter	None
Return	None
Description	This function initializes the EEPROM emulation driver. After a reset it is mandatory to call this API before the start of any EEPROM emulation operation. It initializes the global structures, detect the valid bank, and prepare the bank for operation.

3.3.2 FEE001_GetStatus

Table 9 FEE001_GetStatus

Function	**uint32_t FEE001_GetStatus(void)**
Parameter	None
Return	0x00 – FEE001_UNINIT 0x01 – FEE001_IDLE 0x02 – FEE001_BUSY 0x03 – FEE001_GC_FAILED
Description	This function will return the current status of the FEE001 App.

3.3.2.1 Example Code

```
status_t status;

status = FEE001_GetStatus();
```

3.3.3 FEE001_Write

Table 10 FEE001_Write

Function	uint32_t FEE001_Write (uint8_t BlockNumber, uint8_t *DataBufferPtr)
Parameter	**uint8_t BlockNumber** - Data block ID to be write **uint8_t *DataBufferPtr** - Location of the buffer with data to be programmed
Return	0x00 – FEE001_COMPLETE 0x02 – FEE001_UNINITIALIZED 0x04 – FEE001_INVALID_PARAM 0x05 – FEE001_ERROR 0x08 – FEE001_OPER_NOT_ALLOWED
Description	This function programs the user defined data block into the emulated EEPROM. This function also triggers the garbage collection when there is insufficient space left in the data bank for accepting a new write request. However, if 'Disable Garbage Collection' option is enabled, a write request to a filled data bank will be rejected and garbage collection will not triggers automatically.

3.3.3.1 Example Code

```
uint8_t BlockNumber;
uint8_t DataBuffer[12] =  {0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,
                           0xA6,0xA7,0xA8,0xA9,0xAA,0xAB};

if(FEE001_GetStatus() == FEE001_IDLE)
{
  FEE001_Write(BlockNumber,DataBuffer);
}
```

3.3.4 FEE001_Read

Table 11 FEE001_Read

Function	**uint32_t FEE001_Read (uint8_t BlockNumber, uint16_t BlockOffset, uint8_t *DataBufferPtr, uint16_t Length)**
Parameter	**uint8_t BlockNumber** - Data block ID to be read **uint16_t BlockOffset** - Position to start the read operation **uint8_t *DataBufferPtr** - Location where the data has to be copied **unit16_t Length** - Size of the data to be read in byte
Return	0x00 – FEE001_COMPLETE 0x02 – FEE001_UNINITIALIZED 0x04 – FEE001_INVALID_PARAM 0x05 – FEE001_ERROR 0x06 – FEE001_BLOCK_INCONSISTENT 0x07 – FEE001_BLOCK_INVALID 0x08 – FEE001_OPER_NOT_ALLOWED
Description	This function requests to read the data block from the EEPROM bank. This read does not perform a CRC check for the data bytes read out from the flash, even when CRC is enabled for the data block by the user.

3.3.4.1 Example Code

```
status_t status;
uint8_t BlockNumber;
uint8_t ReadBuffer[12];

if(FEE001_GetStatus() == FEE001_IDLE)
{
   status = FEE001_Read(BlockNumber,0,ReadBuffer,12);
}
```

3.3.5 FEE001_ReadBlockWithCRC

Table 12 FEE001_ReadBlockWith CRC

Function	**uint32_t FEE001_ReadBlockWithCRC (uint8_t BlockNumber, uint8_t *DataBufferPtr)**
Parameter	**uint8_t BlockNumber** - Data block ID to be read **uint8_t *DataBufferPtr** - Location where the data has to be copied
Return	0x00 – FEE001_COMPLETE 0x02 – FEE001_UNINITIALIZED 0x04 – FEE001_INVALID_PARAM 0x05 – FEE001_ERROR 0x06 – FEE001_BLOCK_INCONSISTENT 0x07 – FEE001_BLOCK_INVALID 0x08 – FEE001_OPER_NOT_ALLOWED 0x09 – FEE001_BLOCK_CRC_FAILED
Description	This function requests to read the data block from the EEPROM bank with CRC checks. This API will read the complete data bytes written into flash along with the 16 bit CRC from the header block. The CRC for the read bytes is calculated and compared with the 16 bit CRC read from the header block. The read will return success only if both the calculated CRC and written CRC matches.

3.3.5.1 Example Code

```
status_t status;
uint8_t BlockNumber;
uint8_t ReadBuffer[12];

if(FEE001_GetStatus() == FEE001_IDLE)
{
   status = FEE001_ReadBlockWithCRC(BlockNumber,ReadBuffer);
}
```

3.3.6 FEE001_GetPreviousData

Table 13 FEE001_GetPreviousData

Function	**uint32_t FEE001_GetPreviousData (uint8_t BlockNumber, uint16_t BlockOffset, uint8_t *DataBufferPtr, uint16_t Length)**
Parameter	**uint8_t BlockNumber** - Data block ID to be read **uint16_t BlockOffset** - Position to start the read operation **uint8_t *DataBufferPtr** - Location where the data has to be copied **unit16_t Length** - Size of the data to be read in byte
Return	0x00 – FEE001_COMPLETE 0x02 – FEE001_UNINITIALIZED 0x04 – FEE001_INVALID_PARAM 0x05 – FEE001_ERROR 0x06 – FEE001_BLOCK_INCONSISTENT 0x07 – FEE001_BLOCK_INVALID 0x08 – FEE001_OPER_NOT_ALLOWED
Description	This function reads the content of a data block that was previously written into the emulated EEPROM. This function is useful when the latest data block is corrupted or when an ECC error is identified. This API will search for the previous copy of the same data block in the same data bank. If the block is available then the user buffer will be filled with the data as per the requested length and offset. If the previous data block is also inconsistent or invalid, no further search is made.

3.3.6.1 Example Code

```
status_t status;
uint8_t BlockNumber, ReadBuffer[12];

if(FEE001_GetStatus() == FEE001_IDLE)
{
  status = FEE001_GetPreviousData(BlockNumber,5,ReadBuffer,12);
}
```

3.3.7 FEE001_StartGarbageCollection

Table 14 FEE001_StartGarbageCollection

Function	**uint32_t FEE001_FEE001_StartGarbageCollection (void)**
Parameter	None
Return	0x00 – FEE001_COMPLETE 0x05 – FEE001_ERROR
Description	This function triggers the garbage collection process at any time during the execution of EEPROM emulation. The function is accepted only if the EEPROM emulation driver is in an idle state.

3.3.7.1 Example Code

```
status_t status;

status = FEE001_StartGarbageCollection();
```

3.3.8 FEE001_InvalidateBlock

Table 15 FEE001_InvalidateBlock

Function	**uint32_t FEE001_InvalidateBlock (uint8_t BlockNumber)**
Parameter	**uint8_t BlockNumber** - Data block ID to be invalidated
Return	0x00 – FEE001_COMPLETE 0x02 – FEE001_UNINITIALIZED 0x04 – FEE001_INVALID_PARAM 0x05 – FEE001_ERROR 0x08 – FEE001_OPER_NOT_ALLOWED
Description	The function invalidates the user defined data block. The invalidation process will write a single block of 16 bytes with all data bytes programmed to 0xff and a header with an Invalid state. Any future read operation on this data block will result in Invalid data.

3.3.8.1 Example Code

```
uint8_t BlockNumber;


if(FEE001_GetStatus() == FEE001_IDLE)
{
  FEE001_InvalidateBlock(BlockNumber);
}
```

3.4 Application Example: DALI Control Gear with non-volatile memory

EEPROM emulation can be useful for numerous applications. Here we describe the use case of EEPROM emulation in a DALI (Digital Addressable Lighting Interface) control application.

In some applications, DALI variables have to be retained when a device is powered down. In this example, these variables will be kept in the emulated EEPROM as well as in the RAM. On powering up, these variables will be copied from the emulated EEPROM to the RAM during DALI initialization. The variables will be written to the emulated EEPROM when there are updates in the RAM. In this example also, the DALI Memory bank 0 and bank 1 will also be placed in the emulated EEPROM.

Note: The EEPROM emulation example shown was developed and tested on DAVE™3.1.10.

3.4.1 FEE001 Configuration

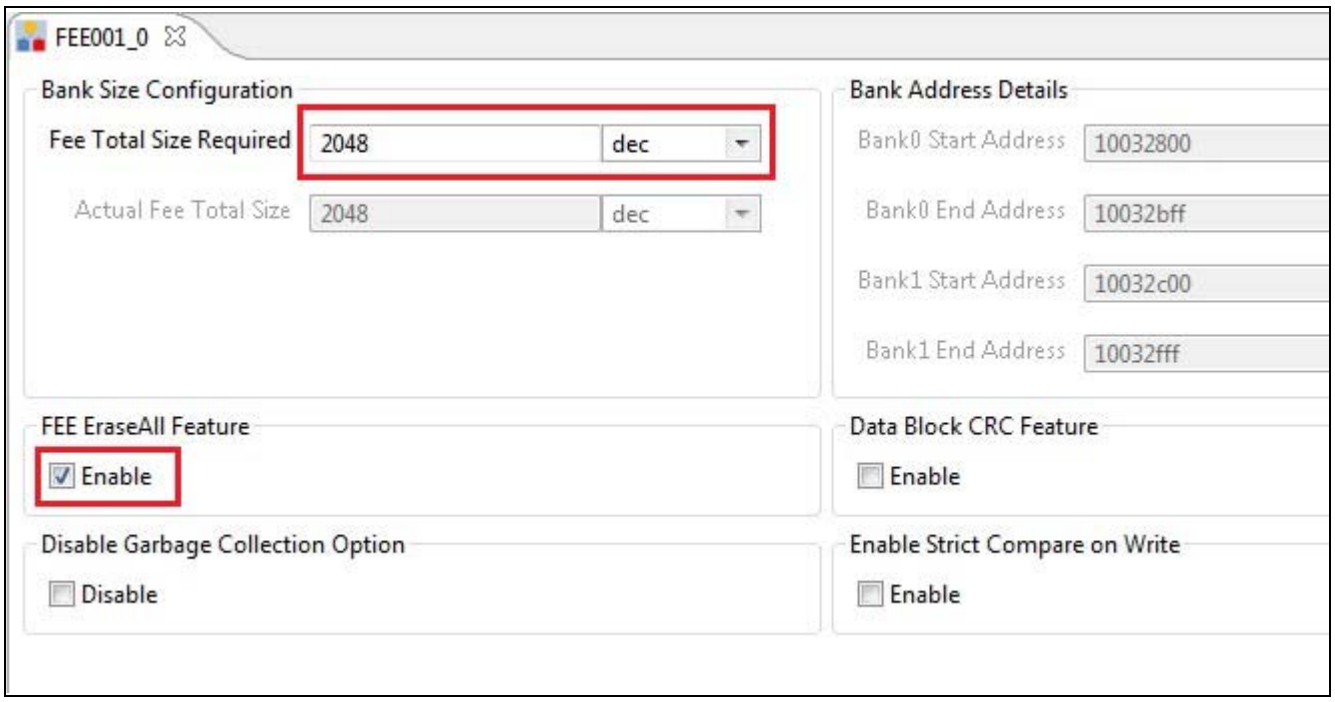

Figure 8 Data bank configuration

For this example, a total of 2048 bytes is located for EEPROM emulation and the Erase All feature is enabled, which means that the FEE driver can erase the data bank if the data is corrupted and recovery is not possible during initialization.

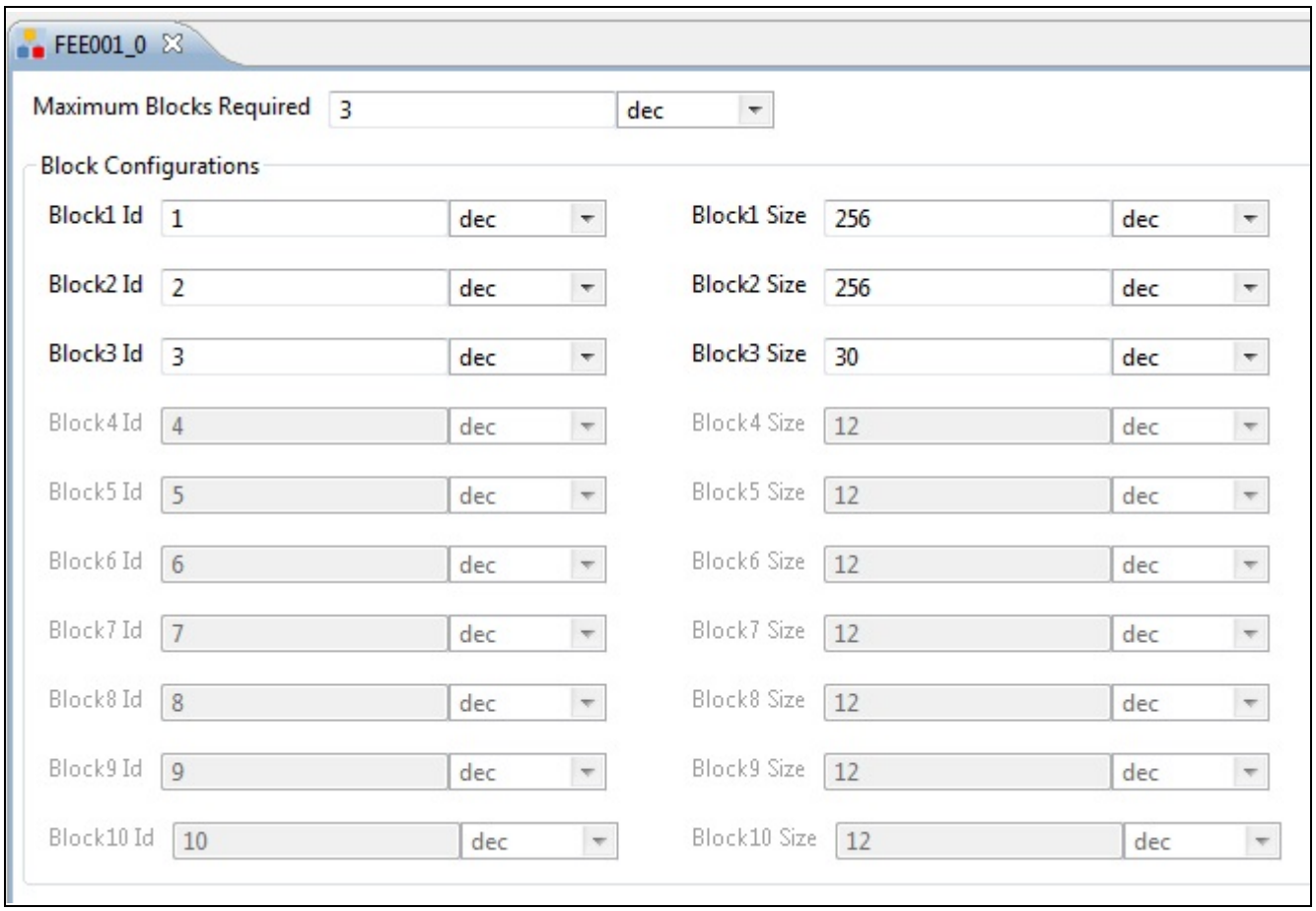

Figure 9 Data block configuration

For data block configuration, three data blocks will be configured as follow:

- Block1 ID: 1, Block1 Size: 256 bytes (for DALI Memory Bank 0)
- Block2 ID: 2, Block2 Size: 256 bytes (for DALI Memory Bank 1)
- Block3 ID: 3, Block3 Size: 30 bytes (for DALI variables)

3.4.2 Writing DALI variables

To save the updated DALI variables into emulated EEPROM, a DALI callback function, WriteBufferToVarSec is used. This function copies the data from RAM into a buffer, ReadWriteBuffer before writing into emulated EEPROM with FEE001_Write.

```
/* Call Back Function
 * Write to Variables Flash Sector (Flash EEPROM Emulation)
 */
void WriteBufferToVarSec(void)
{
    uint8_t count;

    Clear_ReadWriteBuffer();
    ReadWriteBuffer[0U] = DALICG02_FLASH_SECT_NEW_DATA;
    for(count=1;count<28;count++)
    {
        ReadWriteBuffer[count] = (uint8_t)DALICG02_Handle0.DALI102_Handle->aucDALICG02_FlashVa
    }
    ReadWriteBuffer[29U] = DALICG02_FLASH_SECT_PROG;

    status = FEE001_Write(3U, ReadWriteBuffer);
}
```

Figure 10 Write DALI variables to EEPROM

3.4.3 Reading DALI variables

The DALI variables are read from the emulated EEPROM via a DALI callback function, ReadVarSec. The DALI variables are first fetched from the emulated EEPROM to a buffer, ReadWriteBuffer via FEE001_Read. This data is then copied to DALI variable array aucDALICG02_FlashVariables_Sector_tbl which is located on RAM, and will be used by the DALI driver.

```
/* Call Back Function
 * Read Variables Flash Sector (Flash EEPROM Emulation)
 */
void ReadVarSec(void)
{
    uint8_t count;

    Clear_ReadWriteBuffer();
    status = FEE001_Read(3U,0U,ReadWriteBuffer,30U);
    for(count=0;count<30;count++)
    {
        DALICG02_Handle0.DALI102_Handle->aucDALICG02_FlashVariables_Sector_tbl[count] = (uint3
    }
}
```

Figure 11 Read DALI variables from EEPROM

4 Revision History

Current Version is V1.1, 2014-12

Page or Reference	Description of change
V1.0, 2014-10	
	Initial Version
V1.1, 2014-12	
REF	Added EEPROM Emulation chapter

www.infineon.com

Edition 2014-12

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference

AP32280

Legal Disclaimer

THE INFORMATION GIVEN IN THIS APPLICATION NOTE (INCLUDING BUT NOT LIMITED TO CONTENTS OF REFERENCED WEBSITES) IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office. Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.