CY8CMBR2xxx

Device Programming Specifications

Document #: 001-78561 Rev. **

April 19, 2012

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone (USA): 800.858.1810
Phone (Intnl): 408.943.2600
http://www.cypress.com

License

Copyrights

Disclaimer

Flash Code Protection

Cypress products meet the specifications contained in their particular Cypress Data Sheets. Cypress believes that its family of PSoC products is one of the most secure families of its kind on the market today, regardless of how they are used. There may be methods that can breach the code protection features. Any of these methods, to our knowledge, would be dishonest and possibly illegal. Neither Cypress nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Cypress is willing to work with the customer who is concerned about the integrity of their code. Code protection is constantly evolving. We at Cypress are committed to continuously improving the code protection features of our products.

Contents

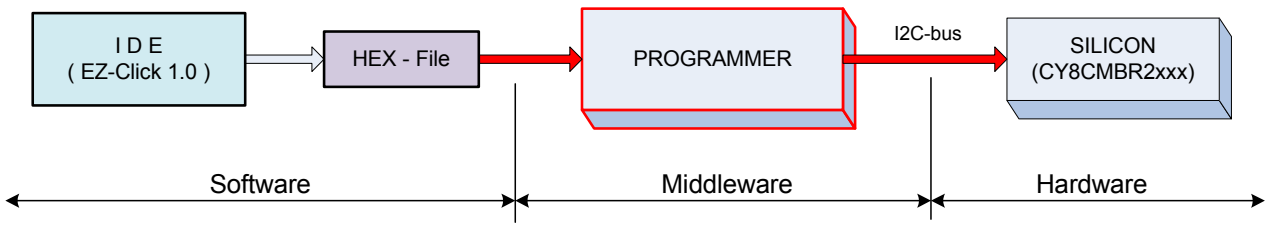
1. Introduction

This programming reference manual gives you the information you need to program the nonvolatile memory of CY8CMBR2xxx devices. It describes a communication protocol that an external programmer can access, explains the programming algorithm, and gives electrical specifications of physical connections.

1.1 Programmer

A *programmer* is a hardware-software system that stores a binary program (hexadecimal file) in the silicon's program (flash) memory. The programmer is an essential component of the engineer's prototyping environment or an integral element of the manufacturing environment (mass programming). The high-level diagram of the development environment is illustrated in Figure 1-1.

Figure 1-1. Programmer in Development Environment



In the manufacturing environment, the IDE block is absent because its main purpose is to produce a hex file.

As shown in Figure 1-1, the programmer performs three functions:

- Parses the hex file; extracts necessary information
- Interfaces with the silicon as an I2C-master
- Implements the programming algorithm by translating the hex data into I2C-signals

The structure of the programmer depends on its exploiting requirements. It can be software- or firmware-centric:

Software-centric: The programmer's hardware works as a bridge between the protocol (such as USB) and I2C. All I2C commands are passed to the hardware through the protocol from an external device (software). The bridge is not involved in the parsing of the hex file and programming algorithm—the upper layer (software) performs this task. Examples of such programmers are the Cypress MiniProg3 and TrueTouch Bridge.

Firmware-centric: This is an independent hardware design in which all the functions of the programmer, including storage for the hex file, are contained in one device. Its main purpose is to be a mass programmer in manufacturing.

This document does not include the specific implementation of the programmer, instead focusing on data flow, algorithms, and physical interfacing. Specifically, it covers the following topics, which correspond to the three functions of the programmer:

- Data to be programmed
- Interface with the chip
- Algorithm used to program the target device

1.2 Introduction to CY8CMBR2xxx

The CY8CMBR2xxx family is an application-specific integrated circuit (ASIC) device for CapSense® end-applications. This device doesn't require any coding; instead it has configuration registers programmed via the I2C-bus.

The nonvolatile subsystem of the silicon consists of a flash memory system with a maximum of 64 bytes. The flash memory system stores the LED configuration and the device configuration information.

The part can be programmed after it is installed in the system by way of the I2C interface (in-system programming). The characteristics of I2C-slave interface are:

- Communication speed up to 100 kHz
- No bus stalling – no clock stretching
- The I2C-address is configurable (during programming)

This document focuses on the specific programming operations without referencing the silicon architecture. Many important topics are detailed in the Appendices. Most of the other material also appears in the CY8CMBR2xxx Technical Reference Manual (TRM).

The Appendices in this document are:

- Appendix A. Intel Hex File Format
- Appendix B. I2C Protocol – Packets and Signal
- Appendix C. I2C Timing Specifications for CY8CMBR2xxx family
- Appendix D. Electrical Specifications of the CY8CMBR2xxx family.

Document Revision History

Document Title: CY8CMBR2xxx Device Programming Specifications

Document Number: 001-78561

Revision	Issue Date	Origin of Change	Description of Change
**	04/19/2012	ANDI	New specification

2. Required Data

This chapter describes the information that the programmer must extract from the hex file to program the CY8CMBR2xxx silicon.

2.1 Hex File Origin

Customers use the EZ-Click GUI to develop their projects. After development is completed, the nonvolatile configuration of the silicon is saved in the file. Only two records in this file actually target the flash memory:

- LED configuration registers
- Device configuration registers

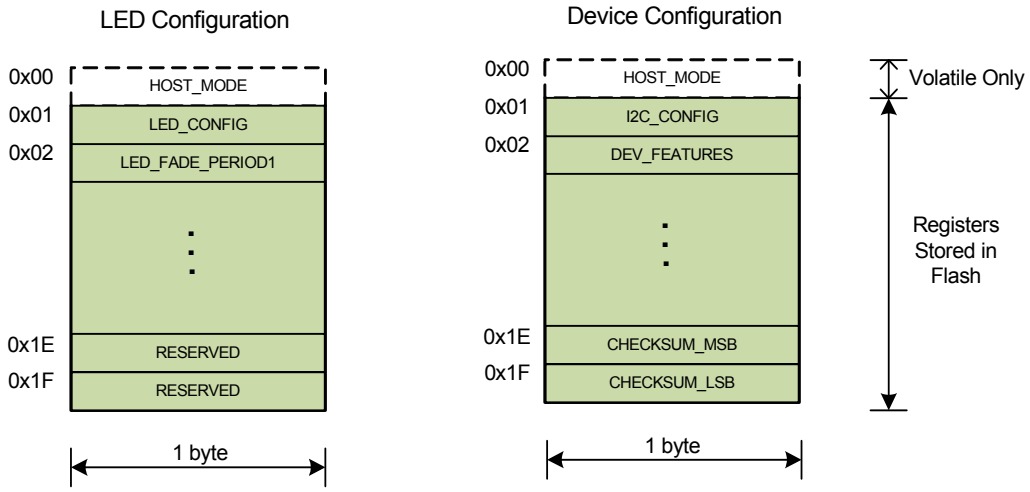
Other records are auxiliary and are used to keep the integrity of the programming flow.

2.2 Nonvolatile Subsystem

The flash memory is organized into two banks of 31 bytes each. The programming granularity is one bank at a time. The first bank represents the registers of LED Configuration mode, and the second bank the registers of Device Configuration mode. During silicon's start-up (after HW/SW reset) the pre-programmed functionality is loaded into corresponding volatile memory (registers).

Figure 2-1 shows the flash organization and how it maps onto the IO space of the silicon. Note that only registers in green are stored in the flash.

Figure 2-1. Nonvolatile Subsystem



The flash memory is mapped directly to IO registers of the silicon. The first 31 bytes of flash configure 0x01-0x1F registers of LED Configuration mode; the second 31 bytes of flash configure 0x01-0x1F registers of Device Configuration mode. The total flash capacitance is 62 bytes, and *Programmer* extracts all of this data from hex-file.

Note: The register 0x00 (HOST_MODE) is volatile only; it is not stored in flash. That holds true for both modes (LED/Device). The programmer uses it to select the required operating mode (Normal, LED, Device, Product Line Testing, Debug Data).

For more information about registers and operating modes, refer to the *Technical Reference Manual.*

2.3 Organization of the Hex File

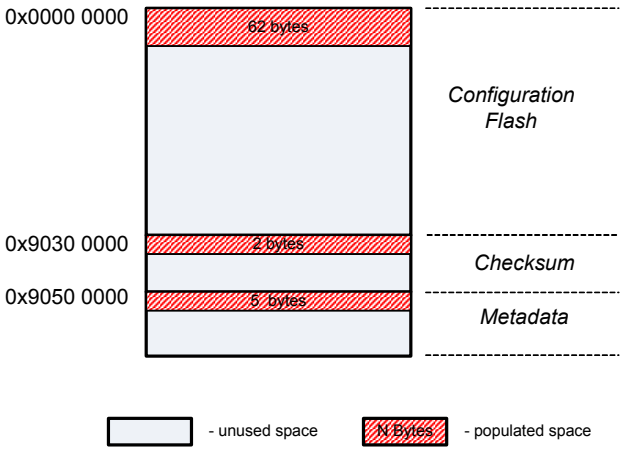
The hexadecimal (hex) file is a medium to describe the nonvolatile configuration of the project. It is the data source for the programmer.

The hex file for the CY8CMBR2xxx family follows the Intel Hex File format. Because Intel's specification is generic, it defines only some types of records that can make up the hex file. The specification allows customizing the format for essentially any possible silicon architecture. The functional meaning of the records is defined by the silicon vendor and typically varies for different chip families. See Appendix A: Intel Hex File Format on page 31 for details of the Intel Hex File format.

The CY8CMBR2xxx family defines three types of data sections in the hex file: configuration flash, checksum, and metadata. See Figure 2-2 on page 8 to determine the allocation of these sections in the address space of the Intel hex file.

The address space of the hex file does not map to the physical addresses of the CPU (other than the User's Flash, which is an unintentional coincidence). The programmer uses hex addresses (see Figure 2-2) to read sections from the hex file into its local buffer. Later, this data is programmed (translated) into corresponding addresses of the silicon.

Figure 2-2. Organization of Hex File for the CY8CMBR2xxx Family



0x0000 0000 – Configuration Flash (64 bytes). This Flash must be programmed. The first 31 bytes are for LED configuration mode, and the second 31 bytes are for device configuration mode.

0x9030 0000 – Checksum (two bytes). This is the checksum of the entire user flash section—the arithmetical sum of every byte in the user's flash. Only 2 bytes of the result are saved in this section, in big-endian format (MSB byte is first). This must be used by the programmer to check the integrity of the hex file and to verify the quality of the programming. In this context, "integrity" means that the Checksum and User Flash sections must be correlated in this file.

0x9050 0000 – Metadata (five bytes). This section contains data that is not programmed into flash. These parameters are used during programming. The fields in this section are listed in the following table.

Table 2-1. Meta Data in Hex File

Offset	Data Type	Length in Bytes
0x00	Hex file version	2 (big-endian)
0x02	I2C program address	1
0x03	I2C verify address	1
0x04	Device ID	1

- **Hex File Version**: This 2-byte field in Cypress's hex-file defines its version (or type). The version for CY8CMBR2xxx family is "0x0100". Programmer should use this field to check whether the file corresponds to CY8CMBR2xxx device or, in general cases, to select appropriate parsing algorithm if it supports some families.
- **I2C Program Address:** I2C address is to be used during programming step. Note that after the device was programmed by this file, its I2C address can be changed. During subsequent programming cycles, the "I2C Verify Address" should be used by the programmer. This field makes sense only for the first programming cycle.
- **I2C Verify Address:** I2C address to be used during verification step. Note that after the first programming cycle, the "I2C Program" and "I2C Verify" will become the same. The correct usage of "I2C Program/Verify" addresses by the programmer will be covered in the later sections.
- **Device ID:** This one-byte value defines the part number for which this HEX-file is generated. This is a value that corresponds to the content of Device ID register in the chip. For example, for CY8CMBR2110, this value is 0xA1. The programmer uses this field to check whether hex-file corresponds to target chip.

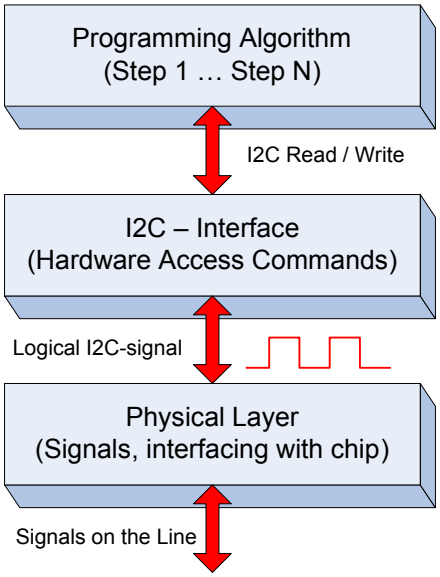
3. Communication Interface

This chapter explains the low-level details of the communication interface.

3.1 The Protocol Stack

Figure 3-1 illustrates the stack of protocols involved in the programming process. The programmer must implement both hardware and software components.

Figure 3-1. Programmer's Protocol Stack



The Programming Algorithm protocol—the top-most protocol—implements the whole programming flow in terms of atomic I2C commands. It is the most solid and fundamental part of this specification. For more information on this algorithm, see Chapter 4: Programming Algorithm on page 17.

The Serial Wire Debug (I2C) Interface and Physical Layer are the lower-layer protocols. Note that the physical layer is the complete hardware specification of the signals and interfacing pins; included are drive modes, voltage levels, resistance, and other components. Upper protocols are logical and algorithmic levels.

The purpose of the I2C interface layer is to be a bridge between pure software and hardware implementations. The Programming Algorithm protocol is implemented completely in software; its smallest building block is the I2C command. The whole programming algorithm is the meaningful flow of these blocks. The I2C interface helps to isolate the programming algorithm from hardware specifics, making the algorithm reusable. The I2C interface must transform the software representation of these commands into line signals (digital form).

3.2 I2C Interface

I2C (Inter-Integrated Circuit) is the industry standard communication interface developed by Phillips Semiconductors (now NXP Semiconductors). It is a synchronous, serial, 8-bit-oriented, bidirectional 2-wire bus that implements a master/slave relationship with as many as 128 slaves on the bus. The I2C standard defines the following working modes: Standard (up to 100 kHz), Fast (up to 400 kHz), up to 1 MHz in Fast-mode+, or up to 3.4 MHz in High Speed. The complete bus specification can be found on the official NXP website. Designers of I2C-compatible chips must use I2C-bus specification and user manual (UM10204) specification as a reference to ensure that new devices meet all specified limits.

Cypress's family of CY8CMBR2xxx devices is I2C-compatible, and these devices operate in slave mode. The master (host) uses an I2C bus to program flash or configure devices in runtime, read CapSense data, and so on.

The third-party programmer of the CY8CMBR2xxx device must implement I2C-master according to standard specification. The developer of programmer probably will use any available solution of master, which passed the test for compliance with the I2C specification. Such ICs are produced by more than 50 companies around the globe.

Note that the programmer infrequently might need to work in a multi-master environment. Consider that possibility when selecting the master's solution. In most cases, programming will be running on single-master buses.

The CY8CMBR2xxx I2C interface has the following features:

- 7-bit addressing mode (up to 128 slaves)
- Bit-rate up to 100 kHz (standard mode)
- No bus stalling - no clock stretching
- I2C buffer - 32 bytes

Developer of programmer must make sure that selected or designed solution of I2C-master supports all above features, which are a subset of I2C-specification.

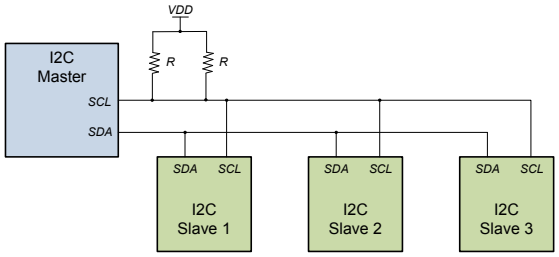
The I2C-bus defines two digital pins to communicate with master (programmer); they are sufficient for bidirectional, semi-duplex data exchange (byte granularity). These two wires are (both are bidirectional):

1. SCL (Serial Clock) - this line is used to synchronize slave with the master.
2. SDA (Serial Data) - this line is used to send data between data and slave.

Figure 3-2 shows an example of an I2C-bus with slaves.

Note: During programming of CY8CMBR2xxx device, the I2C-bus executes only transport function (sends bytes between master and slave). A complete set of lines is required from the programmer to communicate with the CY8CMBR2xxx device, as specified in section 4.3 - "Physical Layer."

Figure 3-2. I2C-bus Connection Schematic



The programming flow consists of multiple Read and Write I2C-transactions. These transactions are atomic transactions from the standpoint of this specification. They can be of different length in bytes, but both are embraced in bus's START and STOP signals. Repeated-START is not used.

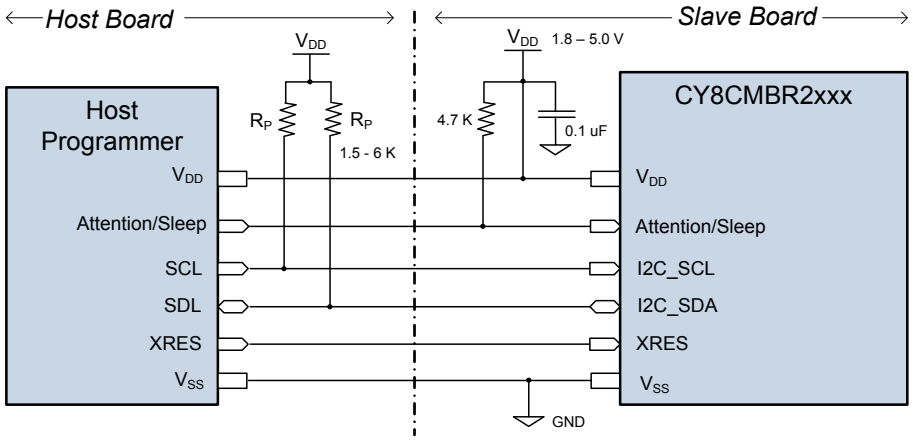
See Appendix B to understand the structure of the Read/Write transactions and their waveforms on the bus.

3.3 Physical Layer

This section describes the hardware connections between the programmer and the target device for programming. It shows the connection schematic and gives information on electrical specifications.

The external interface connection between the host programmer and the target CY8CMBR2xxx device is shown in the following figure. This figure also depicts all power supply connections required in the typical working conditions of the chip.

Figure 3-3. Connection Schematic of Programmer



Only six pins are required to communicate with the chip. Note that SCL and SDA pins are only required by the I2C protocol. Attention/Sleep connection is mandatory.

The optional HW Reset (XRES) pin may be used to reset the slave(s). This pin is used if the firmware is stuck or to clear the I2C-bus (when SCL is stuck LOW - very unlikely event). It is used to reset the part as a first step in a programming flow.

You can program a chip in Reset, No-Reset, or Power Cycle mode. The mode defines only the first step—how to acquire

the part—in the programming flow. The other steps are identical (I2C-traffic).

- Reset mode – To start programming, the host toggles the XRES line, then sends I2C commands. In this case, the power on the target board can be supplied by the host or by an external power adapter (V_{DD} line can be optional).
- No-Reset mode – In this mode host must be sure that power is supplied to target board via external adaptor or from host. Then it can generate I2C-traffic right away. XRES line is not used and V_{DD} connection can also be optional if target is powered by external source. This method has a drawback - chip is not reset before programming, this method will fail if I2C-bus or firmware is stuck.
- Power Cycle mode – To start programming, the host powers on the target and then starts sending I2C commands. The XRES line is not used.

In general, it is recommended that the programmer use all six pins and support, at least, the Reset mode programming. Power Cycle mode support is optional.

Table 3-1. Programming Mode

Mode	Necessary Pins	Unused Pins	Use Cases
Reset	V_{DD} (Optional) GND XRES SCL SDA Attention/Sleep	V_{DD} (if self-powered)	Board can be self-powered (V_{DD} is not needed). Board consumes too much current, which programmer cannot supply (V_{DD} is not needed). 6-pin case – when host supplies power and toggles XRES (this is the most popular programming method). If there are other devices on I2C-bus it is recommended to connect host's XRES to every chip. It will ensure that host can reset every slave which may stuck the bus.
No-Reset	V_{DD} (Optional) GND SCL SDA Attention/Sleep	V_{DD} (if self-powered)	Board can be self-powered (V_{DD} is not needed). Board consumes too much current, which programmer cannot supply (V_{DD} is not needed). I2C-master (host) doesn't have XRES pin. When I2C-master can not supply power, target board is self-powered.
Power Cycle	V_{DD} GND SCL SDA Attention/Sleep	XRES	When XRES pin is not available on the part's package, the only way to reset a part is the Power Cycle. It doesn't make much sense for CY8CMBR2xxx family where every package has XRES pin. Reset mode is the recommended one. Some third-party I2C-masters can use this mode if they don't implement XRES line, but can supply power (on/off) to reset a part.

Table 3-2. CY8CMBR2xxx Pin Names and Requirements

CY8CMBR2xxx Pin Name	Function	External Programmer Drive Modes
V_{DD}	Digital Power Supply Input (1.8 – 5.0 V)	Positive voltage – powered by external power supply or by programmer.
V_{SSD}	Power Supply Return	Low resistance ground connection. Connect to circuit ground.
XRES	Active high external reset input (with internal pull down).	Output - drive TTL levels (Drive mode - Strong)
SCL	I2C Clock line (up to 100 KHz)	Output - drive TTL levels (Drive mode - Open Drain Low) Input - read TTL levels in High-Z mode. In general case SCL is bidirectional to watch for clock stretching, but for CY8CMBR2xxx devices stretching is not supported. So, this line is used in unidirectional mode. The external pull up resistor (RP) must be calculated.
SDL	I2C Data line - bidirectional	Output - drive TTL levels (Drive mode - Open Drain Low) Input - read TTL levels in High-Z mode. The external pull up resistor (RP) must be calculated
Attention/Sleep	This pin is used to wake up a device if it is in deep sleep mode. It is required to pull it low by host for I2C communication to function.	Output - drive TTL levels (Drive mode - Strong)

Figure 3-3 on page 12 shows that I2C bus requires external pull-up resistors (RP). In choosing these resistors, consider supply voltage, clock speed, and bus capacitance. The typical value is in the range of 1.5 - 6 KΩ, for supply voltages 1.8 - 5.0 V. More information about calculation of RP value can be found in the I2C-bus specification (UM10204, section 7.1 - Pull-up resistor sizing).

The I2C timing specifications are described in Appendix C: Timing Specifications of the I2C Interface on page 35.

The silicon's electrical specifications are described in Appendix D: Electrical Specifications on page 37.

3.4 Hardware Access Commands

This section focuses on the low-level APIs that must be supported by the programmer of CY8CMBR2xxx devices.

The APIs must be implemented by the I2C Interface layer shown in Table 3-3. They make up software fundamental for high-level Programming Algorithm. This low-level API interface can be considered the Hardware abstraction layer, because it is hardware-independent (but its implementation is particular for concrete hardware). Theoretically, the upper layer (Programming Algorithm in Table 3-3) can be reusable for different programmers' hardware.

Table 3-3 lists the hardware access commands used by the software layer.

Table 3-3. Hardware Access Commands

Command	Parameters	Description
I2C_WriteTransfer	IN address, IN size, IN data[], OUT ackAddr, OUT ackData[]	Executes on the I2C-bus one write transaction embraced in START and STOP signals. It writes "size" bytes from array "data[]" to slave device specified by 7-bit "address". The output parameters are the acknowledgement bits of address and data bytes. ACK is logical "0" and NACK is logical "1".
I2C_ReadTransfer	IN address, IN size, OUT ackAddr, OUT data[]	Executes on the I2C-bus one read transaction embraced in START and STOP signals. It reads "size" bytes into array "data[]" from slave device specified by 7-bit "address". The output parameters are acknowledgement bit of address byte and array of read data. Note, that memory for "data[]" array must be already reserved by the API's caller.
ToggleReset	–	Generates active HIGH reset signal for target device. Programmer must have dedicated pin connected to XRES pin of the target device. See Table 3-2 on page 13. The recommended duration of active signal is ≥1 ms, see Appendix C: Timing Specifications of the I2C Interface on page 35 for details.
Power	IN state	If the programmer powers the target device, it must have this function to supply power to the device.
Attention	IN state	Sets the Attention/Sleep output line of the programmer to necessary state (LOW or HIGH).
Delay	IN delay_ms	Programmer must be able to delay programming flow for necessary time (50 - 1000 ms).

For more information on the structure and waveform of the Read/Write I2C transactions, see Appendix B.

3.5 Pseudocode

Programming flow consists of numerous I2C_Read-/Write-Transfer commands, with multiple status acknowledgement bits that must be checked every time. It would be convenient to wrap them up in new procedures. This document uses easy-to-read pseudocode to show the programming algorithm. The following two commands are used for the programming script:

```
I2C_Write (address, size, data[])
I2C_Read (address, size, OUT data[])
```

Where `address`, `size`, and `data[]` parameters have the same meaning as in the corresponding I2C_Read-/Write-Transfer commands (see Table 3-3). The upper-layer APIs will automatically check all ACKs of current transaction and will return single status via its name. These APIs will help to keep the programming script concise. The following are some usage examples:

```
BYTE[3] data = {0x07, 0x25, 0xAF}
I2C_Write(0x04, 3, data)
BYTE[10] data; //reserve 10 bytes
I2C_Read (0x04, 10, OUT data)
```

The defined Write and Read pseudo-commands must be successful if they return ACK status for every written byte. For Read transaction, this is ACK of just address byte, but for Write transaction address and all data bytes, it must be ACKed. If at least one byte is NACKed, the transaction is treated as failed. In this case, depending on the programming context, the process must be terminated or the transaction tried again.

The implementation of Write/Read pseudo-commands based on hardware access commands (Table 3-3) appears on the following page.

```
bool I2C_Write (address, size, data[]) {
       BYTE ackAddr;
       BYTE[size] ackData; //reserve space for status bytes of data
       //actually it is enough 1 bit for 1 ACK
       //This implementation uses byte to store 1 ACK bit
       I2C_WriteTransfer (address, size, data[], OUT ackAddr, out ackData );
       // Check ACKs of address and data bytes (ACK = 0x00, NACK = 0x01)
       If (ackAddr != 0x00) Return FALSE; //NACK
       For (BYTE i = 0; i < size; i++) {
          If (ackData[i] != 0x00) Return FALSE; //NACK
       }
       Return TRUE; //all ACKed
}
bool I2C_Read (address, size, OUT data){
       BYTE ackAddr; //reserve 1 bit (byte in fact) for ACK bit of address byte
       //assuming that "size" bytes for "data" is allocated by caller
       I2C_ReadTransfer (address, size, OUT ackAddr, OUT data);
       If (ackAddr != 0x00) Return FALSE; //NACK
       Return TRUE; //ACK
}
```

The programming code in Chapter 4: Programming Algorithm on page 17 will be based mostly on *Write/Read* pseudo-commands and some commands from Table 3-3.

4. Programming Algorithm

This chapter describes in detail the programming flow of the CY8CMBR2xxx device. It starts with a high-level description of the algorithm and then describes every step using pseudo-code. All programming script is made up of Hardware Access Commands ("Hardware Access Commands" on page 14), Atomic I2C_Read/Write() APIs ("Pseudocode" on page 14), and High-Level subroutines ("Subroutines used in Programming Flow" on page 18).

It is possible to write all script using "Hardware Access Commands". In that case, the script would be enormous and will have too much duplication. This approach would cause significant inconvenience in studying the script and its support.

The purpose of high-level APIs (pseudo-code) is to make the script easy in reading and eventually mapping on actual programming language.

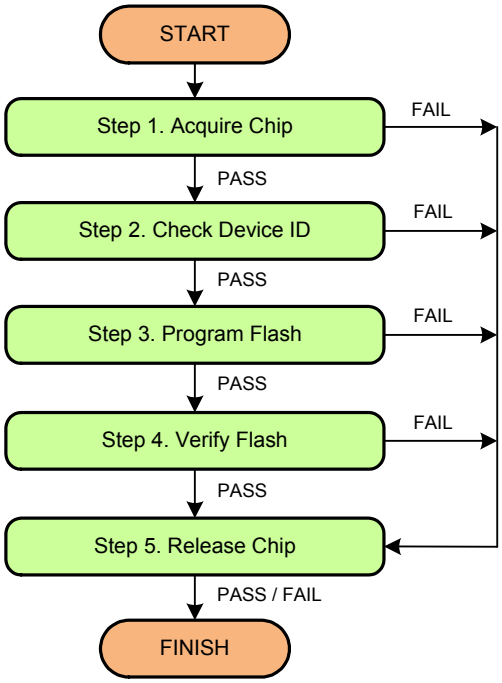
4.1 High-Level Programming Flow

Figure 4-1 shows the sequence of steps that must be executed to program the CY8CMBR2xxx device. These steps are described in detail in the following sections. All of the steps must be completed for a successful programming operation. The programmer should stop the programming flow if any step fails. Also, in pseudocode, it is assumed that the programmer checks the status of each I2C transaction (*I2C_Write, I2C_Read, WritePacket, ReadPacket*). This extra code is not shown in the programming script.

If any of these transactions fails, then programming must be aborted. To abort, execute "Step 5. Release Chip", which executes opposite actions of Step 1. It ensures that Programmer and target are left in the known state after programming is stopped (upon PASS or FAIL).

The flash programming in the CY8CMBR2xxx family is implemented using its registers via I2C-bus. The external programmer puts parameters into Flash (or registers) and requests system calls which, in turn, perform flash updates. See the *CY8CMBR2xxx Technical Reference Manual* for a detailed description of these APIs.

Figure 4-1. High-Level Programming Flow of CY8CMBR2xxx Device



4.2 Subroutines used in Programming Flow

The programming flow includes some operations that are intensively used in all steps. Eventually, the programming code looks compact, and easy to read and understand. Besides that, most of registers and frequently used constants are named and referred to from the pseudocode

Table 4-1. Constants Used in Programming Script

Constant Name	Value	Description
Device's Registers		
HOST_MODE	0x00	Host Mode Register – is used to set the current operation mode, which can be done from any mode. Also, it has bit fields responsible for certain requests/status (they depend on operating mode).
DEVICE_ID	0x1E	Device Identity Register – this is 1-byte read only register returns the unique device ID through which device can be identified. For example the ID of CY8CMBR2110 device is 0x1A. It is identical for parts in one package. This register is available in Normal Operating mode.
DEVICE_STAT	0x03	Device Status Register – is used to indicate whether the factory default configuration or the user configuration is loaded at power up. This register is available in Normal operating mode.
DEV_FEATURES	0x02	Device Features Configuration Register – is used to enable/disable some CapSense features. In context of this document only EMC field is used. This register is available in Device Configuration mode.
I2C_CFG	0x01	I2C Configuration Register – is used to set (or read) the I2C slave address. Slave range is 0x00-0x7F. This register is available in Device Configuration mode.
DATA_OFFSET	0x01	This is address of 1st register in the LED/Device configuration modes, starting from which data will be programmed in flash. They are LED_CONFIG and I2C_CFG registers correspondingly.
HOST_MODE Register's Fields		
MODE_NORMAL	0x00	Normal operating mode.
MODE_LED_CONFIG	0x01	LED configuration mode.
MODE_DEVICE_CONFIG	0x02	Device configuration mode.
F_SAVE_2_FLASH	0x08	Save to Flash bit – is used to store the current configuration into flash. Available only in Device Configuration mode.
F_CHECKSUM_MATCHED	0x10	Checksum matched bit is set if checksum sent by host matches actually calculated one by device. It is checksum of data to be programmed into flash. This bit is available in Device Configuration mode only.
DEVICE_STAT Register's Fields		
F_FACTORY_DEFAULTS_LOADED	0x40	Factory Defaults Loaded – this bit indicates whether factory defaults or the user's configuration is loaded at power up.
DEV_FEATURES Register's Fields		
F_EMC	0x01	The value of EMC bit is used during saving to flash to figure out necessary delay of program operation.

Table 4-2. Subroutines used in Programming Flow

Subroutine	Description
bool WritePacket(address, size, data[])	This subroutine wraps I2C_Write() API from "Pseudocode" on page 14. It keeps sending same I2C write request until it is ACKed. In some cases CY8CMBR2xxx device can be unresponsive on I2C bus, and thus master has to try some times. Also after every successful write, master must wait for 50 ms before it can access this device next time.
bool ReadPacket(address, size, OUT data[])	This subroutine wraps I2C_Read() API from "Pseudocode" on page 14. It keeps sending same I2C read request until it is ACKed. In some cases CY8CMBR2xxx device can be unresponsive on I2C bus, and thus master has to try some times. Also after every successful read, master must wait for 50 ms before it can access this device next time.

The implementation of these subroutines follows. It is based on pseudocode and registers defined in “Hardware Access Commands” on page 14 and “Pseudocode” on page 14. It uses constants defined in this chapter.

The pseudocode is similar to C-style notation.

```
// "WritePacket" Subroutine
bool WritePacket (address, size, data[])
{
       bool ack;
       for (i = 0; i < 20; i++)
       {
          ack = I2C_Write(address, size, data[]);
          if (ack) // ACK
          {
              Delay(50); // 50 ms delay after ACKed write
              return TRUE;
          }
       }
       return FALSE; // NACK
}

// "ReadPacket" Subroutine
bool ReadPacket (address, size, OUT data[])
{
       bool ack;
       for (i = 0; i < 20; i++)
       {
          ack = I2C_Read(address, size, OUT data[]);
          if (ack) // ACK
          {
              Delay(50); // 50 ms delay after ACKed read
              return TRUE;
          }
       }
       return FALSE; // NACK
}
```

4.3 Step 1 – Acquire Chip

The first step in programming flow is to ensure that the device is detected on the bus and is ready for programming. The acquisition algorithm is a bit tricky for the CY8CMBR2xxx device. Besides initialization, determine what I2C-address from hex-file to use: Program or Verify address (see “Nonvolatile Subsystem” on page 7). This address is programmable in CY8CMBR2xxx devices, and it can be changed after successful programming cycle. Note, that Program and Verify addresses are configurable in the EZ-Click GUI. If you need to change them update your project and regenerate hex-file.

The following programming scenarios are possible:

- Target is just from factory and will be programmed first time (and probably the last time) for end design. It is mass production programming. In this case hex-file will have correct “I2C Program Address” that match actual address of target chip. Cypress will ship these devices with default factory configuration where I2C address is 0x04.
- Target is programmed second and more times using hex-files with same Program/Verify addresses. This is prototyping scenario or flash upgrade in field. In this case the device will boot up with Verify address (programmed last time). So, it would be necessary to use this address for both operations Program and Verify.

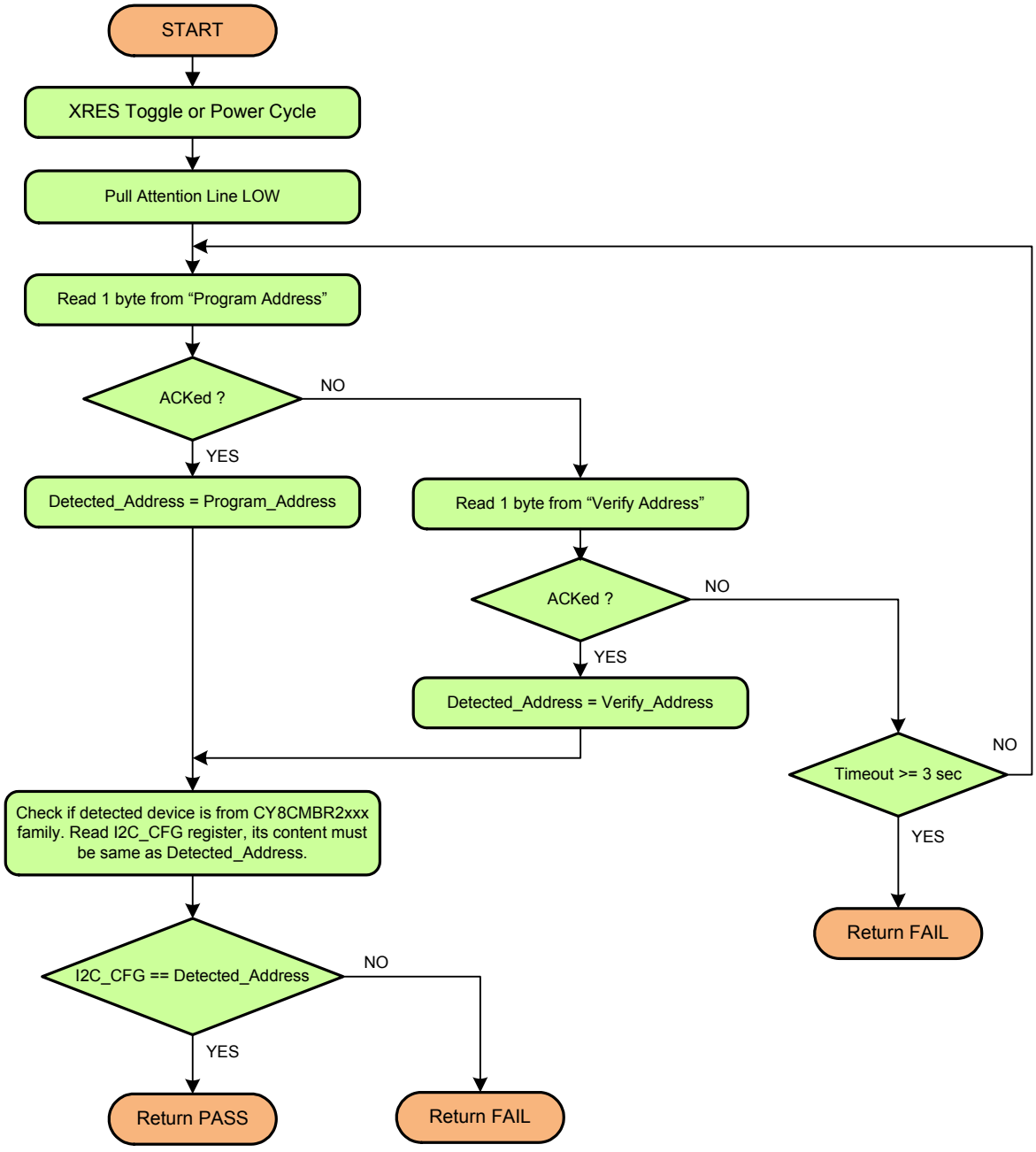
In general, it is required for hex-file to reflect correct Program and Verify addresses of the target chip. The end design must avoid following ambiguity scenarios on the bus:

- When two slaves have same addresses as corresponding fields in hex file (I2C Program/Verify addresses). In this case, the target device may not be detected correctly (especially if both are from CY8CMBR2xxx family).
- After programming step a new target's address will conflict with other device on the bus. Theoretically this case should fail during verification step.

Acquire step programmer determines correct I2C address to be used during next two steps: “Check Device ID” and “Program Flash”. In “Verify Flash” step the Verify address from hex-file is used always.

Figure 4-2 on page 21 shows the acquire algorithm. It is assumed that programmer starts acquisition from one of three modes described in “Physical Layer” on page 12.

Figure 4-2. Flow Chart of Acquisition Sequence



Pseudocode – Step 1. Acquire Chip

```
//-------------------------------------------------------------------------------
// Note, that this step requires following data from hex-file:
// - I2C Program Address (0x90500002 - offset in hex)
// - I2C Verify Address  (0x90500003 - offset in hex)
// Programmer should implement below APIs:
// 1) HEX_GetProgramAddr() and  2) HEX_GetVerifyAddr()

// Reset Target depending on acquire mode - Reset or Power Cycle
If (AcquireMode == "Reset") ToggleReset(); // Toggle XRES pin, target must be powered.
Else If (AcquireMode == "Power Cycle") Power(ON);// Supply power to target.

// Pull Attention/Sleep line LOW
Attention(LOW) ;

// Loop to find out correct I2C device from hex-file
Program_Address = HEX_GetProgramAddr();
Verify_Address  = HEX_GetVerifyAddr();

Do
{
       ack = ReadPacket(Program_Address, 1, out data);
       If (ack == ACK)
{
              Detected_Address = Program_Address; // to be used in Steps 2,3
              break;
}

ack = ReadPacket(Verify_Address, 1, out data);
       If (ack == ACK)
       {
              Detected_Address = Verify_Address; // to be used in Steps 2,3
              break;
       }

}
While (time_elapsed < 3 sec);

If (time_elapsed >= 3 sec) Return FAIL;

//Check if detected device belongs to CY8CMBR2xxx family
//Read I2C_CFG register and compare it with Detected_Address. They must match.
data[0] = HOST_MODE;
data[1] = MODE_DEVICE_CONFIG;
WritePacket (Detected_Address, 2, data);

ReadPacket (Detected_Address, I2C_CFG + 1, out data);

If (data[I2C_CFG] != Detected_Address) Return FAIL;

Return PASS;
//-------------------------------------------------------------------------------
```

4.4 Step 2 – Check Silicon ID

This step is required to verify that the acquired device corresponds to the hex file. It reads the ID from the hex file and compares it to the ID obtained from the target.

Pseudocode – Step 2. Check Silicon ID

```
//-------------------------------------------------------------------------------
// Read "Device ID" from Hex-file - 1 byte from address 0x9050 0004.
// HEX_ReadDeviceID() must be implemented.
// "Detected_Address" is taken from Step 1.
HexID = HEX_ReadDeviceID();

// Read "Device ID" register (in Normal Operating mode)
data[0] = HOST_MODE;
data[1] = MODE_NORMAL;
WritePacket (Detected_Address, 2, data);

ReadPacket (Detected_Address, DEVICE_ID + 1, out data);

If (data[DEVICE_ID] != HexID) Return FAIL;

Return PASS;
//-------------------------------------------------------------------------------
```

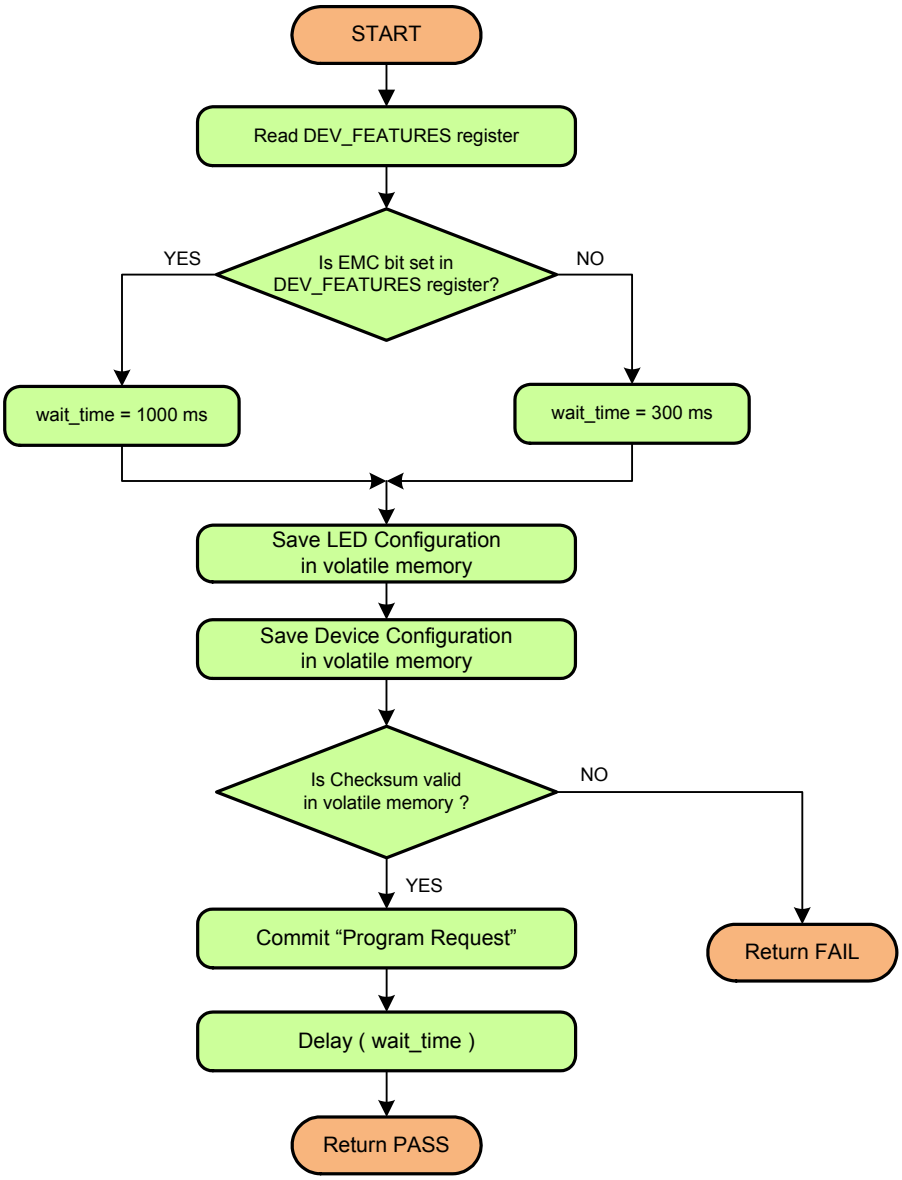
4.5 Step 3 – Program Flash

Programming of flash memory of CY8CMBR2xxx is straightforward. Load configuration data (62 bytes) into volatile memory and commit program. This request programs whole flash and generates software reset to reload new configuration. This is a time-consuming operation; the programmer must wait until this request is completed. The wait time depends on EMC bit in current configuration and can range from 1000 to 300 ms. Only after this delay can the programmer move to “Verify Step”.

The source data is extracted from the hex-file starting from address 0x00000000 (see Figure 2-2 on page 8). Note, that flash size of the acquired silicon must be equal to size of configuration data in the hex-file. This was ensured in “Step 2 – Check Silicon ID” on page 23 by comparing Device IDs of the hex and the target.

The following figure illustrates this programming algorithm.

Figure 4-3. Algorithm of “Program Flash” Step



Pseudocode – Step 3. Program Flash

```
//-------------------------------------------------------------------------------
// Configuration data must be extracted from hex-file (address 0x0000 0000). For that:
// HEX_ReadData() API must be implemented.
// "Detected_Address" is taken from Step 1.

//1. Find delay for Program/Reload operations
data[0] = HOST_MODE;
data[1] = MODE_DEVICE_CONFIG;
WritePacket ( Detected_Address, 2, data);
ReadPacket (Detected_Address, DEV_FEATURES + 1, data)

if ((data[DEV_FEATURES] & F_EMC) == F_EMC)
       wait_delay = 1000; //ms
else
wait_delay = 300;  //ms

//2. Save LED Configuration (31 bytes) in volatile memory
data[0] = HOST_MODE;
data[1] = MODE_LED_CONFIG;
WritePacket ( Detected_Address, 2, data);
//Extract 31 bytes from hex-file from offset 0x00000000
Data_LED = HEX_ReadData( 0, 31 );
//Prepare I2C buffer and write it into volatile memory
data[0] = DATA_OFFSET;
for (i = 0; i < 31; i ++)
{
       data[i+1] = Data_LED[i]
{
WritePacket ( Detected_Address, 32, data ) ;

//3. Save Device Configuration (31 bytes) in volatile memory
data[0] = HOST_MODE;
data[1] = MODE_DEVICE_CONFIG;
WritePacket ( Detected_Address, 2, data);
//Extract 31 bytes from hex-file from offset 0x00000020
Data_Device = HEX_ReadData( 31, 31 );
//Prepare I2C buffer and write it into volatile memory
data[0] = DATA_OFFSET;
for (i = 0; i < 31; i ++)
{
       data[i+1] = Data_Device[i]
{
WritePacket ( Detected_Address, 32, data ) ;

//4. Find if checksum of loaded data is correct
ReadPacket (Detected_Address, HOST_MODE + 1, data)

if ((data[HOST_MODE] & F_CHECKSUM_MATCHED) == 0x00) return FAIL;

//5. Commit Program request
data[0] = HOST_MODE;
data[1] = MODE_DEVICE_CONFIG | F_SAVE_2_FLASH;
WritePacket ( Detected_Address, 2, data);

//6. Wait for necessary time until programming and reloading is completed
Delay ( wait_delay );

return PASS;
```

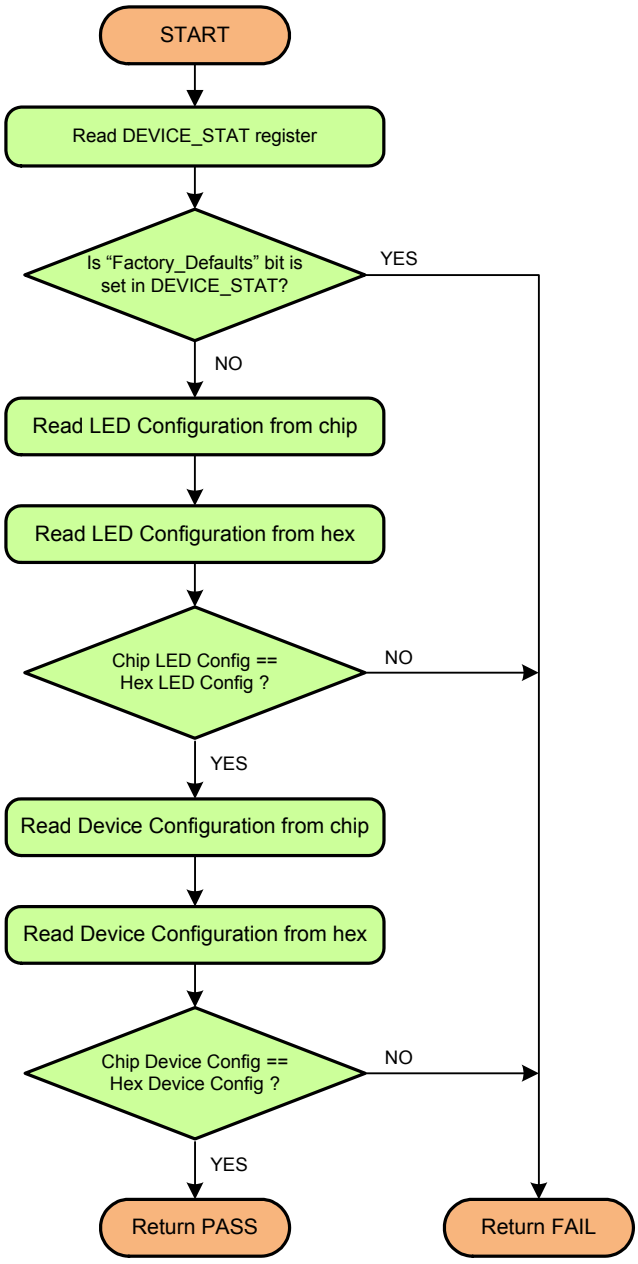
4.6 Step 4 – Verify Flash

This is mandatory for the programmer, because no other method ensures that the written data is correct (for example, checksum). Checksum cannot guarantee reliably that all data in flash is identical to hex-content, so each flash byte must be checked separately.

Verification process starts from checking the FACTORY_DEFAULTS_LOADED bit, which ensures that data from custom flash is loaded. After that, the programmer reads LED and Device configurations from chip and compares it against corresponding hex data. If any difference is found, the programmer must stop and return fail.

The programmer reads new data not from flash directly, but from the same volatile buffers which were used during programming (see Figure 2-2 on page 8). New flash data was automatically loaded there in the end of programming step. This data must be identical to flash's content, since “nobody” tried to change it between Program and Verify steps.

Figure 4-4. Algorithm of “Verify Flash” Step



Pseudocode – Step 4. Verify Flash.

```
//-------------------------------------------------------------------------
// Configuration data must be extracted from hex-file (address 0x0000 0000).
// The I2C address for verification is extracted from hex-file (address 0x9050 0003)
// 2 HEX APIs are used in this step: HEX_ReadData() and HEX_GetVerifyAddr()

Verify_Address = HEX_GetVerifyAddr();

//1. Check if factory defaults or configuration from flash is loaded
data[0] = HOST_MODE;
data[1] = MODE_NORMAL;
WritePacket ( Verify_Address, 2, data);

ReadPacket ( Verify_Address, DEVICE_STAT + 1, data);

if ((data[DEVICE_STAT] & F_FACTORY_DEFAULTS_LOADED) == F_FACTORY_DEFAULTS_LOADED)
       return FAIL;

//2. Verify LED Configuration
data[0] = HOST_MODE;
data[1] = MODE_LED_CONFIG;
WritePacket ( Verify_Address, 2, data);

ReadPacket ( Verify_Address, DATA_OFFSET + 31, out chip_Data);

//Extract 31 bytes from hex-file from offset 0x00000000
hex_Data = HEX_ReadData( 0, 31 );

//Compare them
for (i = 0; i < 31; i++)
{
       if (chip_Data[DATA_OFFSET + i] != hex_Data[i]) return FAIL;
}

//3. Verify Device Configuration
data[0] = HOST_MODE;
data[1] = MODE_DEVICE_CONFIG;
WritePacket ( Verify_Address, 2, data);

ReadPacket ( Verify_Address, DATA_OFFSET + 31, out chip_Data);

//Extract 31 bytes from hex-file from offset 0x00000020
hex_Data = HEX_ReadData( 31, 31 );

//Compare them
for (i = 0; i < 31; i++)
{
       if (chip_Data[DATA_OFFSET + i] != hex_Data[i]) return FAIL;
}

return PASS;
```

4.7 Step 5 – Release Chip

This step, which is the opposite of “Acquire Chip,” releases the chip from the programmer. The programmer executes final actions, such as power down or reset, disconnecting from I2C bus (putting lines into high-Z state), and so on.

After this step, the programmer can be disconnected from the chip. For example, on an automated pipeline, the chip in the socket is replaced by the next part. So, the programmer can start again from “Step 1 – Acquire Chip” on page 20.

Note that in a prototyping environment, when the new functionality must be checked right away, there is no need to put the device into sleep mode. Just keep Attention/Sleep line HIGH and toggle reset line (or even no toggling needed, since new configuration is already loaded and started in the end of “Step 3 – Program Flash” on page 24).

It is recommended to call this step always in the end of programming independently of the status of the previous operations. This step is optional and does not generate any I2C-traffic, but it is necessary to leave the programmer and target in the known state in the end of programming.

Pseudocode – Step 5. Release Chip

```
//-------------------------------------------------------------------------------
// This step depends on the power source. Whether target powered by Programmer or
// external source.

// 1.Move target into sleep state
Attention (HIGH);

// 2.Power Off or Reset device
Power (OFF) // if powered by Programmer

// or ToggleReset() if target uses external power supply.
// if XRES pin not implemented by Programmer, just return (do nothing).

return PASS / FAIL; // this method should return result of previous Steps,
                    // which actually executed real "programming"

//-------------------------------------------------------------------------------
```

Appendix A. Intel Hex File Format

Intel hex file records are a text representation of hexadecimal-coded binary data. Only ASCII characters are used, so the format is portable across most computer platforms.

Each line (record) of Intel hex file consists of six parts as shown in the following figure.

Figure A-1. Hex File Record Structure

Start code (Colon character)	Byte count (1 byte)	Address (2 bytes)	Record type (1 byte)	Data (N bytes)	Checksum (1 byte)

1. **Start code**, one character - an ASCII colon ':'
2. **Byte count**, two hex digits (1 byte) - specifies the number of bytes in the data field.
3. **Address**, four hex digits (2 bytes) - a 16-bit address of the beginning of the memory position for the data.
4. **Record type**, two hex digits (00 to 05) - defines the type of the data field. The record types used in the hex file generated by Cypress are as follows.
   - 00 - Data record, which contains data and 16-bit address.
   - 01 - End of file record, which is a file termination record and has no data. This must be the last line of the file; only one is allowed for every file.
   - 04 - Extended linear address record, which allows full 32-bit addressing. The address field is 0000, the byte count is 02. The two data bytes represent the upper 16 bits of the 32 bit address, when combined with the lower 16-bit address of the 00 type record.
5. **Data**, a sequence of 'n' bytes of the data, represented by 2n hex digits.
6. **Checksum**, two hex digits (1 byte), which is the least significant byte of the two's complement of the sum of the values of all fields except fields 1 and 6 (Start code ':' byte and two hex digits of the Checksum).

Examples for the different record types used in the hex file generated for CY8CMBR2xxx device are as follows.

Consider that these three records are placed in consecutive lines of hex file (Chip-Level Protection and End of Hex File).

:0200000490600A

:0100000002FD

:00000001ff

For the sake of readability, "Record type" is highlighted in red and 32-bit address of the chip-level protection is in blue.

The first record (*:0200000490600A*) is an extended linear address record as indicated by the value in the Record Type field (04). The address field is 0000, the byte count is 02. This means that there are two data bytes in this record. These data bytes (9060) specify the upper 16-bits address of the 32-bit address of data bytes. In this case, all the data records that follow this record are assumed to have their upper 16-bit address as 0x9060 (in other words, the base address is 0x90600000). 0A is the checksum byte for this record:

0x0A = 0x100 – (0x02+0x00+0x00+0x04+0x90+0x60).

The next record (:0100000002FD) is a data record, as indicated by the value in the Record Type field (00). The byte count is 01, meaning there is only one data byte in this record (02). The 32-bit starting address for these data bytes is at address 90600000. The upper 16-bit address (9060) is derived from the extended linear address record in the first line; the lower 16-bit address is specified in the address field of this record as 0000. FD is the checksum byte for this record.

The last record (:00000001FF) is the end of file record, as indicated by the value in the Record Type field (01). This is the last record of the hex file.

Note The data records of the following multi-bytes region in hex file are in big-endian format (MSB byte in lower address): Checksum data at address 0x9030 0000, Meta data at address 0x9050 0000. The data records of the rest of the multi-byte regions in hex file are all in little-endian format (LSB byte in lower address).

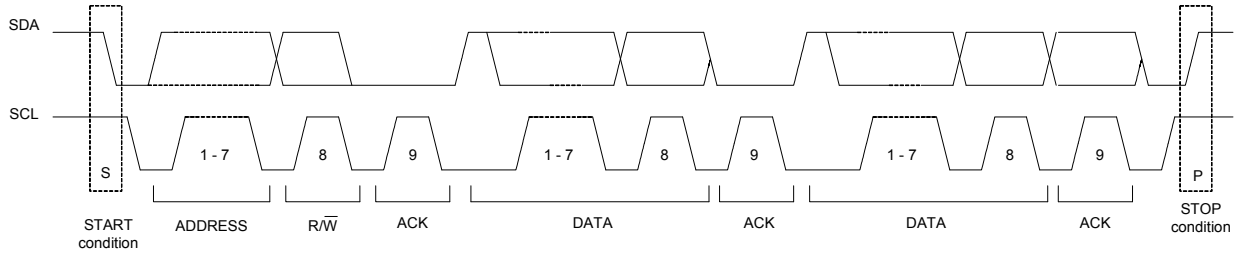
Appendix B. I2C Protocol - Packets and Signals

I2C-interface is a packet based serial transaction protocol and at the pin level uses one bi-directional data line (SDA) and a one clock connection (SCL). Generation of clock signal on I2C-bus is always responsibility of master device(s). Bus clock signals from master can only be altered when they are stretched by a slow slave device holding down the clock line or by another master when arbitration occurs. Clock stretching is not a case for CY8CMBR21xxx devices and multi-master environment is not recommended for programming of this device. A complete data transfer on I2C-bus (one packet) consists of five phases:

- **Start Condition** - this signal initiates packet transfer. It is HIGH to LOW transition on SDA line while SCL is HIGH. The bus is considered to be busy after the START condition. So no any other master will try to access bus while it is busy. The bus is considered to be free again, once STOP condition is generated.
- **Address** - the 7-bit address is sent by master to establish connection with necessary slave device.
- **R/W Bit** - using this bit master informs slave about type of transaction - Read or Write.
- **Data Block** - This is actual data transferred between master and slave. It must be at least 1-byte long, and the number of bytes per transfer is unrestricted. The granularity of the data is 8-bits and it is transferred from master to slave (Write) or from slave to master (Read) depending on the R/W bit.
- **Stop Condition** - this signal ends the packet transfer. It is a LOW to HIGH transition on the SDA line while SCL is HIGH.

The timing diagrams of the I2C-transfer are shown in the following figure. This diagram is common for Read and Write transfer.

Figure B-1. Diagrams of Generic I2C-packets

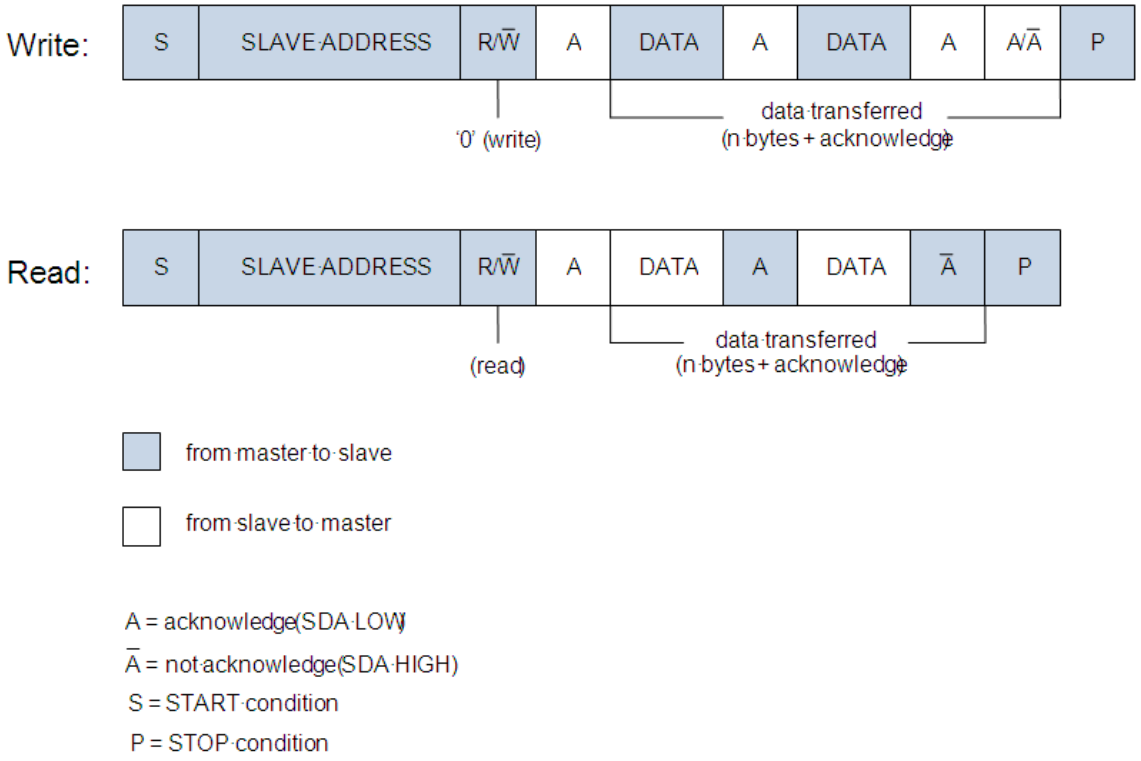


The I2C packet is transmitted in following sequence:

1. The START condition moves bus into the busy state.
2. 7-bit slave address is sent, which is received by all slaves. Once this phase completed, only addressed slave will talk to master, all other will be waiting for STOP condition.
3. R/W bit defines direction of the transaction: LOW - Write to slave, HIGH - Read from slave.
4. ACK bit is sent by slave device signals master that requested address is present on bus and is ready for communication. When SDA remains HIGH during this 9th bit clock pulse, this is defined as Not Acknowledge signal. The master can then generate either a STOP condition to abort the transfer, or a repeated START to start a new transfer. When SDA remains stable LOW during HIGH period of clock - this is Acknowledge signal.
5. Data Slot - consists necessary number of 9-bit data chunks:
   - 8-bit - Data Byte, for Write transaction is sent by master, for Read is sent by slave.
   - 1-bit - ACK signal, for Write transaction is signaled by slave (meaning that it is ready for next byte to receive), for Read transaction master signals by ACK slave that it is ready for next byte.
6. The STOP condition ends transaction and frees the bus.

Note that you can use the ACK slot for clock stretching--the slave pauses the transaction by holding SCL line LOW. The transaction cannot continue until the line is released in HIGH again. Clock stretching is optional; most slave devices, including the CY8CMBR2xxx, do not implement this feature.

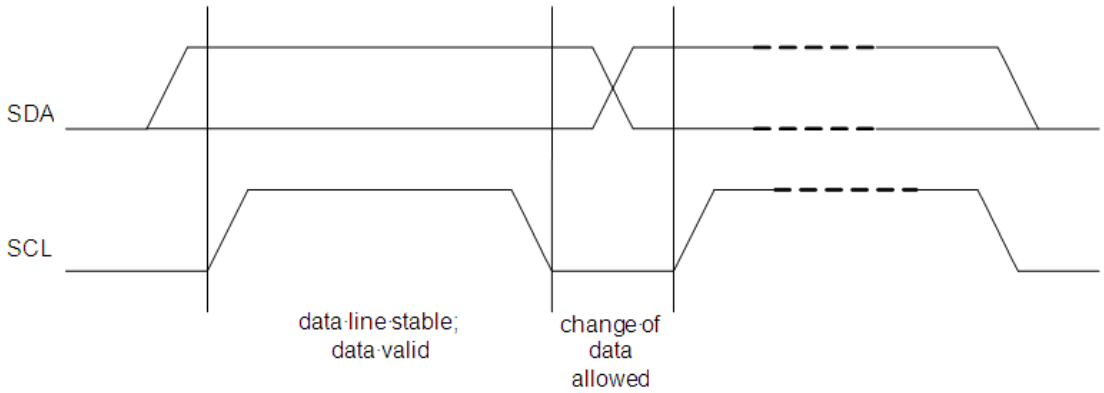
Figure B-2. Format of Read/Write I2C-packets



B.1 Data Validity

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can change only when the clock signal on the SCL line is LOW. One clock pulse is generated for each data bit transferred.

Figure B-3. Bit Transfer on I2C-bus



Appendix C. Timing Specifications of the I2C Interface

Figure C-1. Definition of I2C bus's Timing

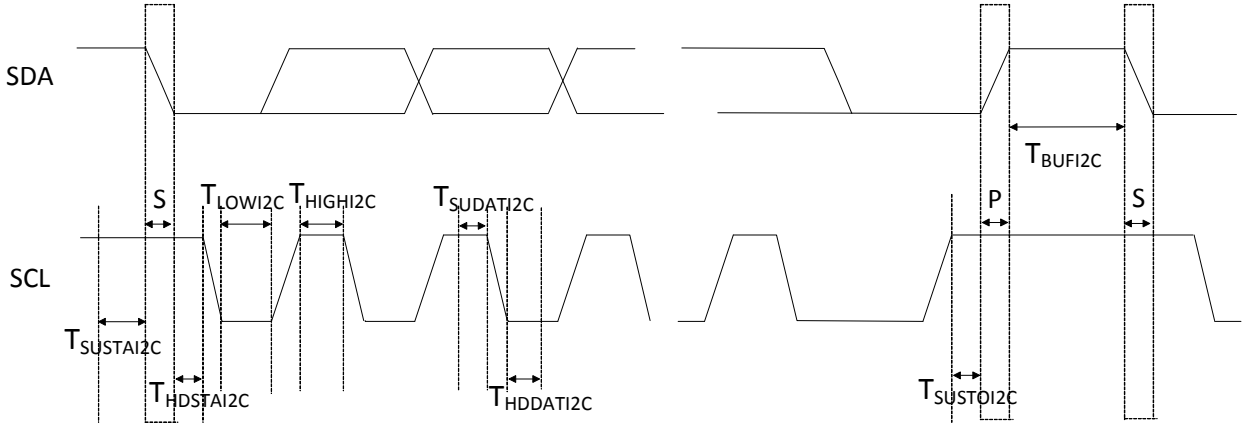


Table C-1. I2C Timing Specifications

Symbol	Description	Min	Max	Unit
F_{SCLI2C}	SCL Clock Frequency	0	100	kHz
$T_{SUSTAI2C}$	Setup Time for a START condition	4.7	–	µs
$T_{HDSTAI2C}$	Hold Time for a START condition. After this period, the first clock pulse is generated	4.0	–	µs
T_{LOWI2C}	LOW period of the SCL clock	4.7	–	µs
$T_{HIGHI2C}$	HIGH period of the SCL clock	4.0	–	µs
$T_{HDDATI2C}$	Data Hold Time	0	–	µs
$T_{SUDATI2C}$	Data Setup Time	250	–	ns
$T_{SUSTOI2C}$	Setup Time for a STOP condition	4.0	–	µs
T_{BUFI2C}	Bus Free Time between a STOP and START condition	4.7	–	µs

Appendix D. Electrical Specifications

The critical electrical specifications are captured in the following tables. For more information, refer to the Technical Reference Manual for the CY8CMBR2xxx family.

Table D-1. Operating Temperature

Parameter	Description	Min	Typ	Max	Units	Notes
T_A	Ambient temperature	-40	–	+85	°C	
T_C	Commercial Temperature	0		+70	°C	
T_J	Operational Die Temperature	-40	–	+100	°C	

Table D-2. DC Chip Level Specifications

Parameter	Description	Min	Typ	Max	Units	Notes
V_{DD}	Supply voltage	1.71	–	5.5	V	After power down, ensure that V_{DD} falls below 100 mV before powering back up.
I_{DD}	Supply current	–	2.88	4.0	mA	Conditions are V_{DD} = 3.0 V, T_A = 25 °C
I_{DA}	Active current	–	2.88	4.0	mA	Conditions are V_{DD} = 3.0 V, T_A = 25 °C, continuous sensor scan
I_{DS}	Deep Sleep current	–	0.1	1.1	µA	Conditions are V_{DD} = 3.0 V, T_A = 25 °C

The following table lists guaranteed maximum and minimum specifications for the voltage and temperature ranges: 3.0 V to 5.5 V and -40 °C ≤ TA ≤ 85°C, 2.4 V to 3.0 V and -40 °C ≤ TA ≤ 85 °C, 1.71 V to 2.4 V and -40 °C ≤ TA ≤ 85 °C, respectively. Typical parameters apply to 5 V and 3.3 V at 25 °C and are for design guidance only.

Table D-3. DC I2C Specifications

Parameter	Description	Min	Typ	Max	Units	Notes
V_{ILI2C}	Input Low Level	–	–	0.25 x V_{DD}	V	3.1 V ≤ V_{DD} ≤ 5.5 V
		–	–	0.25 x V_{DD}	V	2.5 V ≤ V_{DD} ≤ 3.0 V
		–	–	0.25 x V_{DD}	V	1.71 V ≤ V_{DD} ≤ 2.4 V
V_{IHI2C}	Input High Level	0.65 x V_{DD}	–	–	V	1.71 V ≤ V_{DD} ≤ 5.5 V

Table D-4. AC Chip Level Specifications

Parameter	Description	Min	Max	Units	Notes
SR_{POWER_UP}	Power Supply Slew Rate	-	250	V/ms	V_{DD} slew rate during power up.
T_{XRST}	External Reset Pulse Width at Power Up	1		ms	Applicable after device power supply is active
T_{XRST2}	External Reset Pulse Width after Power Up	10		µs	Applicable after device V_{DD} has reached max value