Programming manual

SPC11x8/SPD11x8 Programming Specifications

Revision 3 – May 2020

Introduction

This programming manual provides the necessary information on programming the flash memory of the SPC11x8/SPD11x8 microcontroller. It describes the communication protocol required for access by an external programmer, explains the programming algorithm and gives electrical specifications of the physical connection.

Contents

List of tables

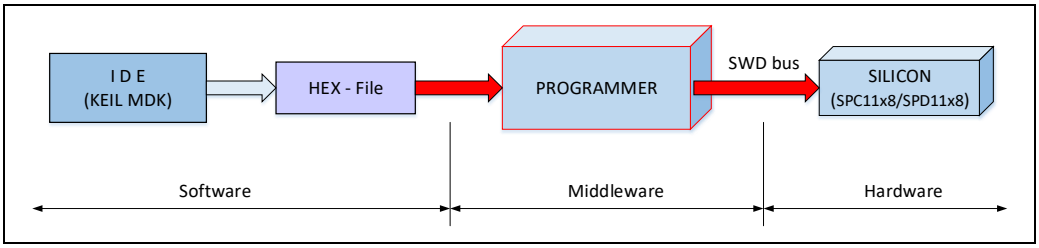
List of figures

1 Introduction

1.1 Programmer

A programmer is a hardware-software system that stores a binary program (hexadecimal file) in the silicon's program 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 view of the development environment is illustrated in Figure 1.

Figure 1. Programmer in Development Environment



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

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

- Parse the hex file and extracts necessary information
- Interfaces with the silicon as a serial wire debug (SWD) master
- Implements the programming algorithm by translating the hex data into SWD signals

This document does not include the specific implementation of the programmer. It focuses mainly on data flow, algorithms, and physical interfacing. Specially, 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 chip

1.2 Introduction to SPC11X8/SPD11X8

The SPC11X8/SPD11X8 integrates a full-feature ARM Cortex-M4 core that can runs up to 200 MHz, is therefore compatible with all ARM tools and software. Customers can program and debug the chip via ARM debug interface. The ARM SWJ-DP interface is embedded in SPC11X8/SPD11X8. The SWJ-DP is a combined JTAG-DP and SW-DP that enables a debug probe to connect to the target using either the SWD protocol or JTAG. And a specific sequence on the SWDIO pin is used to switch between JTAG-DP and SW-DP. In this document, we focus on the programming specifications based on SWD protocol.

The nonvolatile subsystem of the SPC11X8/SPD11X8 consists of a flash memory up to 128KB for storing programs and the chip's configuration data. The device can be programmed after it is installed in the system by way of the SWD interface (in-system programming). The programming frequency ranges from 100 kHz to 10 MHz.

This document focuses on the specific programming operations without referencing the silicon architecture. Many important topics are detailed in the Appendices.

2 Required Data

2.1 Hex File Origin

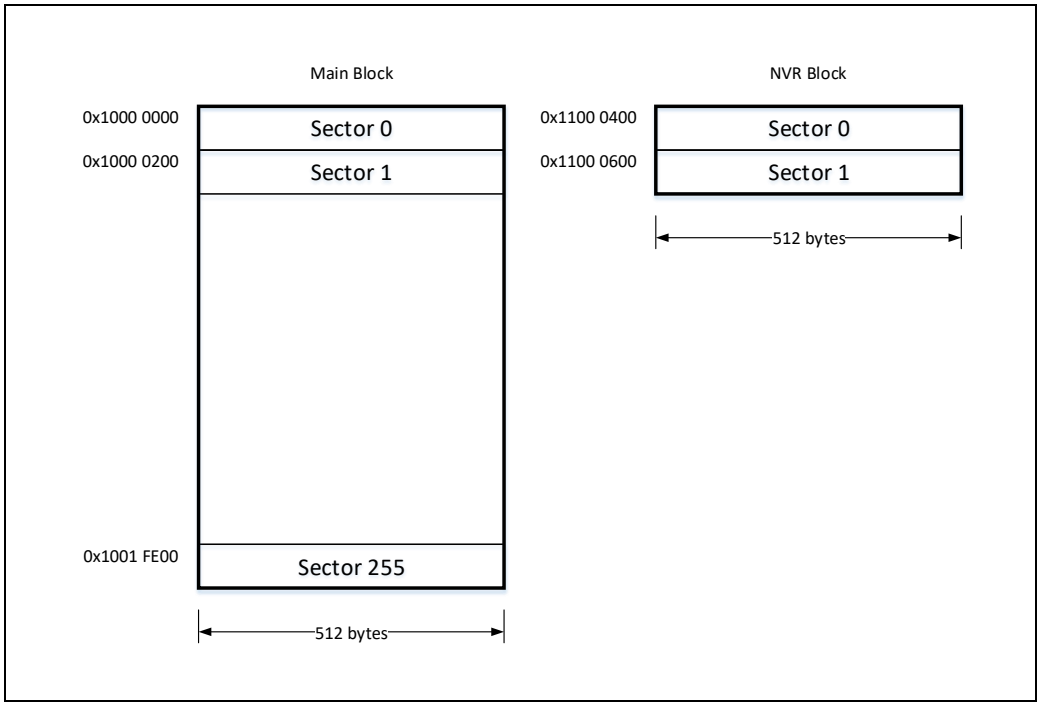
Customers will use KEIL MDK to develop their projects. After development is completed, the customer's code is saved in the hex file.

2.2 Flash Memory System

The flash memory is organized into one main block of up to 128KB. There can be up to 256 sectors in the main block. Each sector consists of two pages and each page consists of 256 bytes. The programming granularity is one page at a time.

In addition to the main block, the flash contains two NVR sectors. One is for OTP flash (Sector 0) and another stores chip-level configuration data (Sector 1). Figure 2 shows the flash organization and how it maps to the CPU's memory space.

Figure 2. Flash Memory



The possible capacity of flash main block in the SPC11x8/SPD11x8 can be 32, 64, 96 or 128. For example, a chip with 32KB contains one flash main block with 64 sectors, and a chip with 128KB has one flash main block with 256 sectors.

The flash memory is mapped directly to the CPU's address space starting from 0x10000000. Therefore, the firmware or external programmer can read its content directly from the given address.

2.3 Organization of the Hex File

The hex file is the data source for the programmer. The hex file for the SPC11X8/SPD11X8 device follows the Intel Hex File format.

The SPC11X8/SPD11X8 device defines three types of data sections in the hex file: user flash, OTP flash and configuration data. See Figure 3 to determine the allocation of these sections in the address space of the Intel hex file.

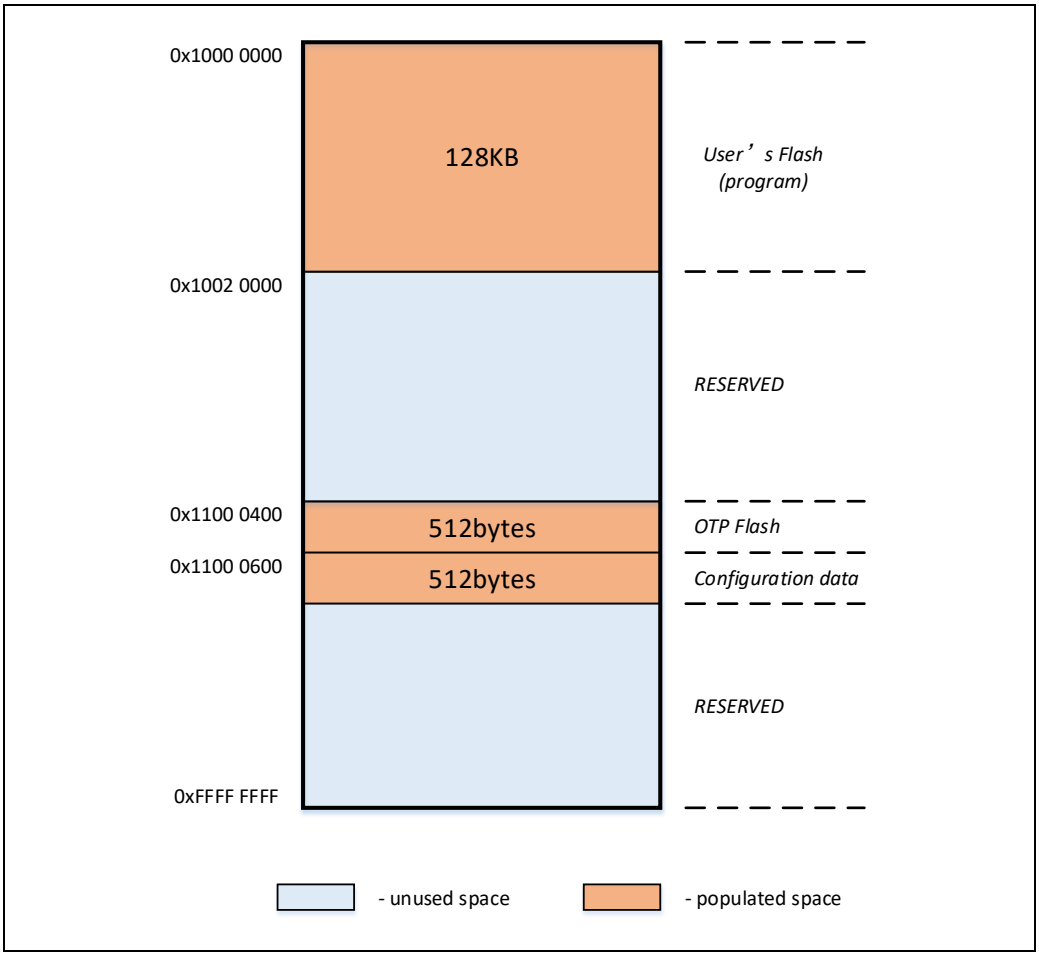
The address space of the hex file maps on to the physical addresses of the CPU. The programmer uses hex addresses (see Figure 3) to read sections from the hex file into its local buffer. Later, this data is programmed into the corresponding addresses of the device.

0x1000 0000 – User's Flash (128 KB max). This is the user's program (code) that must be programmed. The programmer can either read all of this section at once or gradually by 256-byte blocks. The programming of the flash is carried out on the page on the basis of 256 bytes for each request.

0x1100 0400 – OTP Flash (512 bytes max). This section contains data that can only be programmed once. The programmer can either read all of this section at once or gradually by 256-byte blocks.

0x1100 0600 – Configuration Data (512 bytes max). This section contains data that is used to configure the target chip security feature. The programmer can either read all of this section at once or gradually by 256-byte blocks.

Figure 3. Organization of Hex File for the SPC11X8/SPD11X8



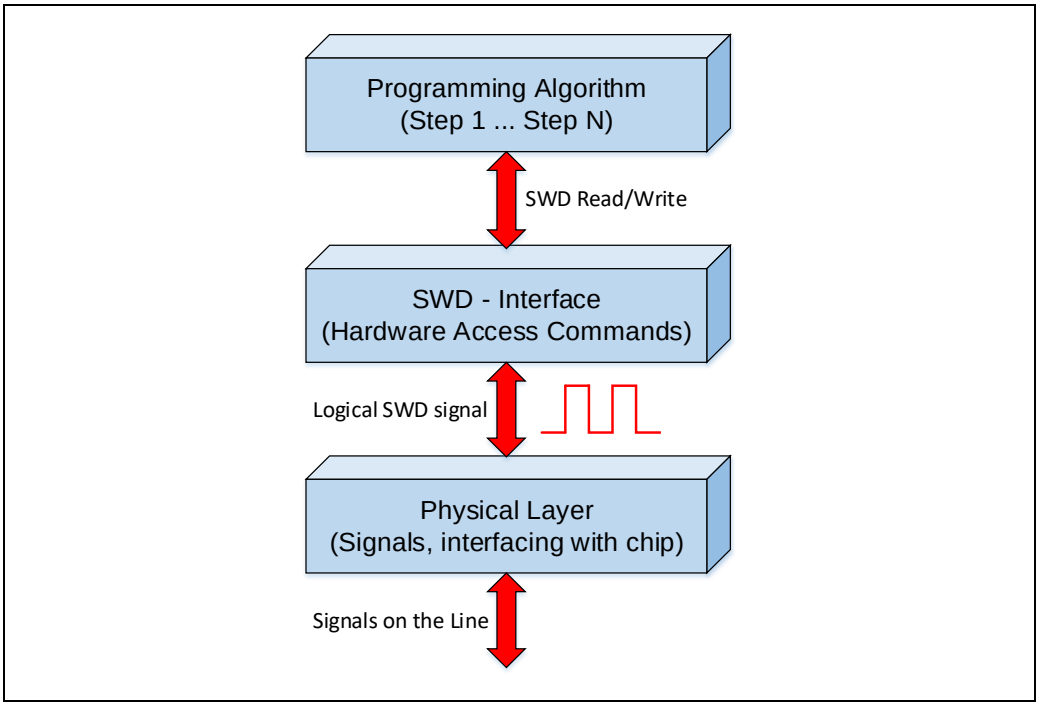
3 Communication Interface

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

3.1 The Protocol Stack

Figure 4 illustrates the stack of protocol involved in the programming process. The programmer must implement both hardware and software components.

Figure 4. Programmer's Protocol Stack



The Programming Algorithm protocol implements the whole programming flow in terms of atomic SWD commands. It is the most solid and fundamental part of this specification.

The SWD 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, and includes drive modes, voltage levels, resistance, and other components. Upper protocols are logical and algorithmic levels.

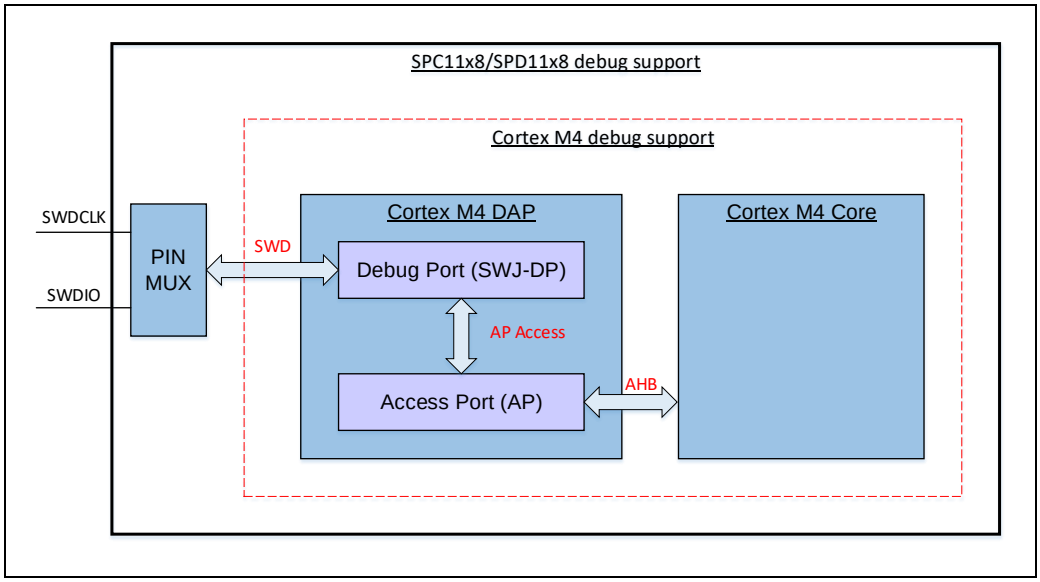
The purpose of the SWD interface layer is to be a bridge between pure software and hardware implementations. The "Programming Algorithms" protocol is implemented completely in software; its smallest building block is the SWD command. The whole programming algorithm is the meaningful flow of these blocks.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

3.2 SWD Interface

The SWD interface uses the Serial Wire Debug protocol developed by ARM. The SPC11X8/SPD11X8 integrates the standard Cortex M4 DAP block provided by ARM. Therefore, it complies with the ARM specification, ARM Debug Interface v5 Architecture Specification.

Figure 5 shows the SPC11X8/SPD11X8 debug support. It includes the debug interface, CPU subsystem. The standard ARM modules are outlined in red.

Figure 5. SPC11X8/SPD11X8 Debug Support



The SWD interface (ARM) defines just two digital pins to communicate with the external programmer/debugger. The SWDCLK and SWDIO pins are sufficient for bidirectional, semi-duplex data exchange.

Only three types of SWD commands can appear on the bus: Read, Write, and Line Reset. The Line Reset command is used only once during programming to establish a connection with the device. Read and Write make up rest of the programming flow.

Before programming the target chip flash, the external programmer should transfer the flash programming algorithms (Provided by SPINTROL) to the target chip through SWD interface. And then, the typical operation of the programmer is to transfer all necessary parameters to the target chip SRAM and request target chip to run the flash programming algorithms. This task is performed only by the SWD Read and Write command.

3.3 Hardware Access Commands

The Cortex-M4 DAP module, shown in Figure 5, supports three types of transactions: Read,

Write, and Line Reset. All of them are defined in the ARM specification. The APIs must be implemented by the SWD Interface layer shown in Figure 4. In addition, the upper protocol, Programming Algorithm, requires two extra commands to manipulate the hardware: Power (state) and ToggleReset(). Table 1 lists the hardware access commands used by the software layer.

Table 1. Hardware Access Command

Command	Parameters	Description
SWD_LineReset		Standard ARM command to reset the debug port (DAP): 1. Send at least 50 SWCLK cycles with SWDIO HIGH. 2. Send the 16-bit sequence on SWDIO = 0111100111100111 (MSB transmitted first). 3. Send at least 50 SWCLK cycles with SWDIO HIGH. This sequence will make the SWD interface of the target chip enter line reset state. It is the first transaction in programming flow.
SWD_IdleCycle	IN count32	Sends numbers of SWCLK cycles with SWDIO HIGH.
SWD_Write	IN APnDP, IN addr, IN data32, OUT ack	Sends 32-bit data to the specified register of the DAP. The register is defined by “APnDP” (1 bit) and “addr” (2 bits) parameters. DAP returns 3-bit status in “ack”.
SWD_Read	IN APnDP, IN addr, OUT data32, OUT ack, OUT parity	Reads 32-bit data from the specified register of the DAP. The register is defined by “APnDP” (1-bit) and “addr” (2 bit) parameters. DAP returns 32-bit data, status, and parity bit of read 32-bit word.
ToggleReset		Generates reset signal for target device. The programmer must have a dedicated pin connected to XRSTn pin of the target device.
Power	IN state	If the programmer powers the target device, it must have this function to supply power to the device.

For information on the structure of the Read/Write SWD packet and its waveform on the bus, please see ARM Debug Interface v5 Architecture Specification.

The *SWD_Read/Write* commands allow accessing registers of the Cortex-M4 DAP module from Figure 5. The DAP functionally is split into two control units:

- Debug Port (DP) – responsible for the physical connection to the programmer/debugger.
- Access Port (AP) – provides the interface between the DAP module and one or more debug components (such as Cortex-M4 CPU).

The external programmer can access registers of these access ports using the following bits in the SWD packet:

- APnDP - select access port (0 - DP, 1 - AP).
- ADDR - 2-bit field addressing a register in the selected access port.

The *SWD_Read/Write* commands are used to access these registers. They are the smallest transactions that can appear on the SWD bus. Table 2 shows the DAP registers that are used during programming.

Table 2. DAP Registers (in ARM notation)

Register	APnDP	Address	Access	Full Name
IDCODE	0	2'b00	R	Identification Code Register
CTRL/STAT	0	2'b01	R/W	Control/Status Register
SELECT	0	2'b10	W	AP Select Register
CSW	1	2'b00	R/W	Control Status/Word Register
TAR	1	2'b01	R/W	Transfer Address Register
DRW	1	2'b11	R/W	Data Read/Write Register

For more information about these registers, see the ARM Debug Interface v5 Architecture Specification.

3.4 Pseudocode

This document uses an easy-to-read pseudocode to show the programming algorithm. The following two commands are used for the programming script:

```
Write_DAP ( Register, Data32)
Read_DAP ( Register, OUT Data32)
```

Where Register is an AP/DP register defined by APnDP and Address bits (see Table 2). The pseudo-commands correspond to Read or Write SWD transactions. The following are some usage examples:

```
Write_DAP( TAR, 0x20000000)
Write_DAP( DRW, 0x12345678)
Read_DAP ( IDCODE, OUT swd_id)
```

The “Register” parameter technically can be represented as the structure in C:

```
struct DAP_Register
{
BYTE APnDP; // 1-bit field
BYTE Addr;  // 2-bit field
};
```

Then, DAP registers will be defined as:

```
DAP_Register   TAR    = { 1, 1 },
               DRW    = { 1, 3 },
               IDCODE = { 0, 0 };
```

The defined *Write* and *Read* pseudo-commands must be successful if both return the ACK status of the SWD transaction. For the Read transaction, the parity bit must be taken into account (correspond to read data32 value). If the status of the transaction, the parity bit, or both is bad, the transaction must be considered to have failed. In this case - depending on the programming context - programming must terminate or the transaction must be tried again.

The implementation of Write/Read pseudo-commands based on hardware access commands *SWD_Read/Write* (in Table 1) are as follows.

```
SWD_Status Write_DAP ( Register, data32 ){
    SWD_Write ( Register.APnDP, Register.Addr, data32, OUT ack);
    Return ack;
}

SWD_Status Read_DAP ( Register, OUT data32){
    SWD_Read ( Register.APnDP, Register.Addr, OUT data32, OUT ack,
    OUT parity);
    If (ack == 3’b001){ //ACK, then check also the parity bit
        Parity_data32 = 0x00;
       For (i=0; i<32; i++) Parity_data32 ^= ((data32 >> i) & 0x01);
       If (Parity_data32 != parity) ack = 3’b111; //NACK
    }
    Return ack;
}
```

The programming code in Chapter 4: Programming Algorithm will be mostly based on

Write/Read pseudo-commands and some commands from Table 1.

3.5 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 SPC11X8/SPD1178 device is shown in Figure 6. The external interface connection between the host programmer and the target SPC11X8/SPD1178 device is shown in Figure 7.

Figure 6. Connection Schematic of Programmer for SPC11X8/SPD1178

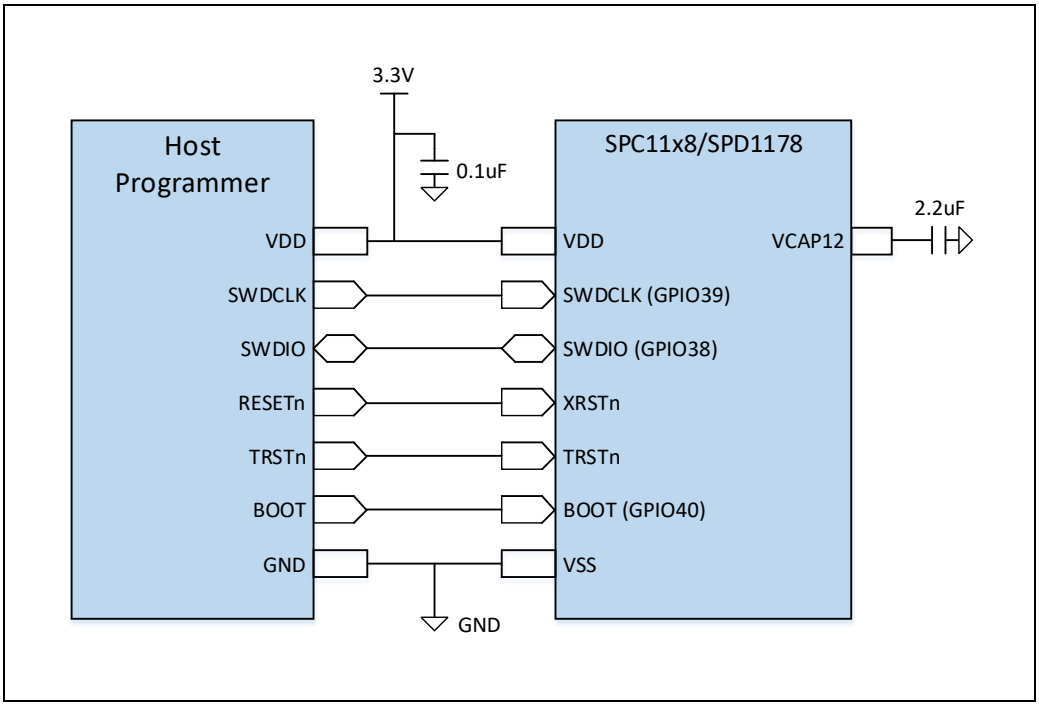
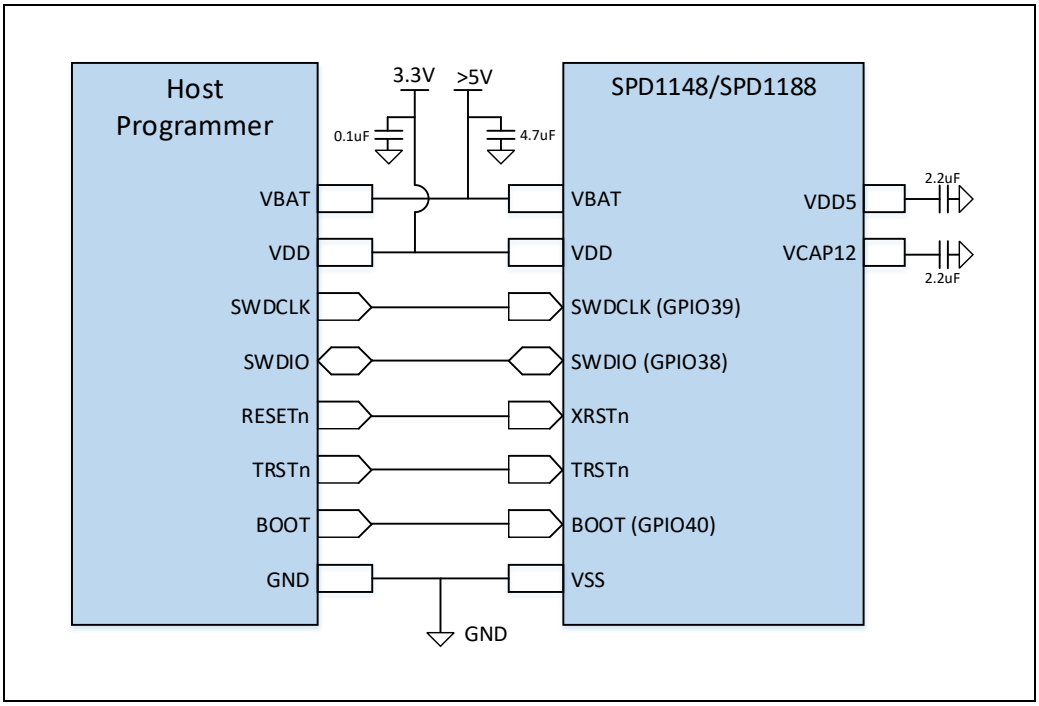


Figure 7. Connection Schematic of Programmer for SPD1148/SPD1188



Only seven or eight pins are required to communicate with the chip. Note that the SWDCLK and SWDIO pins are only required by the SWD protocol. The additional pins (RESETn, TRSTn and BOOT) are required by this device and are not related to the ARM standard. The RESETn pin is used to reset the target as a first step in a programming flow; the TRSTn pin and BOOT pin are used to active debug test. To start programming, the host toggles the RESETn line and then sends SWD commands (see Table 1). The power on the target board can be supplied by the host or by an external power adapter (VDD or VBAT line can be optional).

Table 3. SPC11X8/SPD11X8 Pin Names and Requirements

SPC11X8/SPD11X8 Pin Name	Function	External Programmer Drive Modes
VBAT(1)	Chip Main Power Supply Input (> 5V)	Positive voltage – powered by external power supply or by programmer.
VDD	Digital Power Supply Input (3.3V)	Positive voltage – powered by external power supply or by programmer.
VSS	Power Supply Return	Low resistance ground connection. Connect to circuit ground.
XRSTn	External active low reset input	Output - CMOS levels for SPC11x8/SPD1178; **TTL levels for SPD1148 and SPD1188**
TRSTn	Debug test reset pin, active high	Output - CMOS levels, should keep HIGH
BOOT	Boot pin	Output - CMOS levels, should keep HIGH
SWDCLK	SWD clock input	Output - CMOS levels
SWDIO	SWD data line	Input / Output - drive CMOS levels
AVDD	Analog Power Supply Input (3.3V)	This power can be supplied from V_{DD} source
AVSS	Power Supply Return	-

(1) VBAT power supply input is only need for SPD1148 and SPD1188.

4 Programming Algorithm

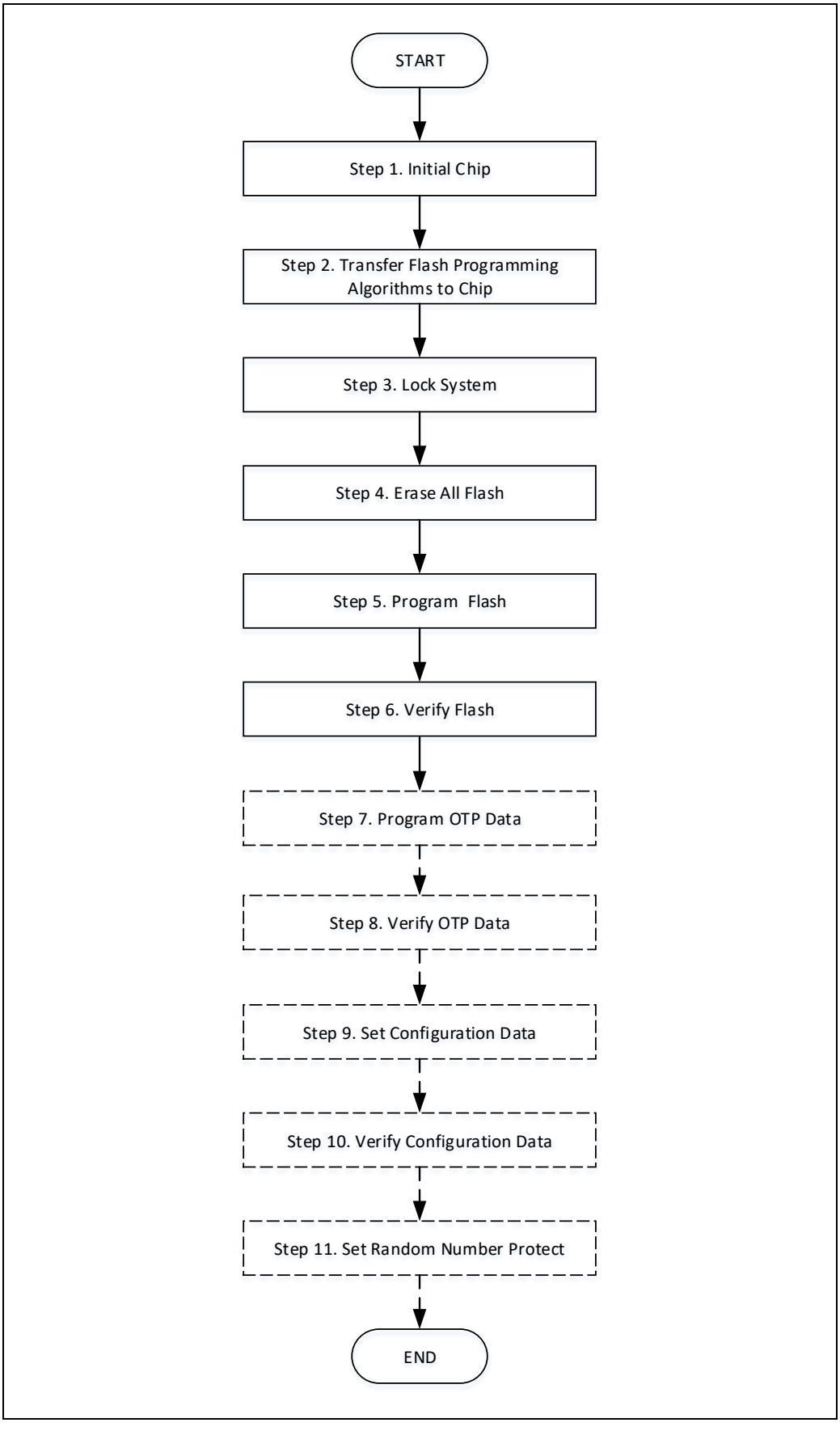
This chapter describes the programming flow of the SPC11X8/SPD11X8 device. It starts with a high-level description of the algorithm and then describes every step using a pseudocode. All code is based on upper-level subroutines made up of atomic SWD instruction (see "Pseudocode" in section 3.4). These subroutines are defined in the next section. The ToggleReset() and Power() commands are also used (from Table 1).

4.1 High-Level Programming Flow

Figure 8 shows the sequence of steps that must be executed to program the SPC11X8/SPD11X8 device. **If no OTP data or Configuration data to be programmed, Step 7 to Step 10 are not necessary. Step 11 is also not necessary if not setting Random Number protect.** These steps are described in detail in following sections. All of the steps in this programming flow must be completed successfully 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 SWD transaction (Write_DAP, Read_DAP, WriteIO, ReadIO). This extra code is not shown in the programming script.

The flash programming is implemented by SPINTROL. And the programmer should transfer the flash programming algorithms to target chip SRAM first. Then, the external programmer puts parameters into SRAM (or registers) and requests the target chip to run the flash programming algorithms, which perform flash updates.

Figure 8. High-Level Programming Flow of SPC11X8/SPD11X8



4.2 Sub-routines used in Programming Flow

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

Table 4. Constants Used in Programming Script

Constant Name	Value (Address)	Description
Cortex-M4 Registers		
DCRDR	0xE000EDF8	Debug Core Register Data Register – used to hold register read result or to write data to the selected register
DCRSR	0xE000EDF4	Debug Core Register Selector Register – used to select the CPU core register
DHCSR	0xE000EDF0	Debug Halting Control and Status Register – used to halt or run CPU core
Chip Registers		
WDT0REGKEY	0x40001018	Watchdog Timer 0 Lock Register
WDT0CTL	0x40001008	Watchdog Timer 0 Control Register
WDT1REGKEY	0x40002018	Watchdog Timer 1 Lock Register
WDT1CTL	0x40002008	Watchdog Timer 1 Control Register
Chip SRAM		
SRAM_BASE	0x20000000	SRAM base address for the SPC11X8/SPD11X8 flash programming algorithm
S_CMD	0x20003010	4-byte, stored the programming command
S_STATUS	0x20003018	4-byte, finish status of programming
S_RESULT	0x20003020	4-byte, result of programming
S_ADDRESS	0x20003028	4-byte, stored the start address of programming
S_SIZE	0x20003030	4-byte, stored the data size (in bytes)
S_DATA	0x20003040	256-byte, stored the byte data to be programmed

Table 5. Programming Commands

Command	Value	Description
S_CMD_ERASE	0xF120	Erase target chip flash memory
S_CMD_PROG	0xF130	Program target chip flash page
S_CMD_LOCK	0xF140	Lock target chip system parameters
S_CMD_SET_CONFIG	0xF150	Set target chip configuration data
S_CMD_BLANK_CHECK	0xF180	Flash blank check
S_CMD_VERIFY	0xF190	Verify data in target chip Flash memory
S_CMD_PROG_OTP	0xF1A0	Program target chip OTP flash page
S_CMD_SET_RDN	0xF1B0	Set target chip random number protect
S_CMD_NONE	0x0	Dummy command

Table 6. Programming Status

Status	Value	Description
S_STATUS_DONE	0x05FA	Programming command has been executed
S_STATUS_INIT	0x0	Initial value

Table 7. Programming Result

Result	Value	Description
S_RESULT_SUCCESS	0x1111	Programming command executes successfully
S_RESULT_FAIL	0x2222	Programming command executes failed
S_RESULT_INIT	0x0	Initial value

Table 8. Sub-routines used in Programming Flow

Sub-routine	Description
bool WriteIO(addr32, data32)	Writes 32-bit data into specified address of CPU address space. Returns “true” if all SWD transactions succeeded (ACKed).
bool ReadIO(addr32, OUT data 32)	Reads 32-bit date from specified address of CPU address space. Note that actual size of read data (8, 16, 32 bits) depends on setting in CSW register of DAP. By default all accesses are 32 bits long. Returns “true” if all SWD transactions succeeded (ACKed).
bool PollCmdStatus()	Waits until programming command is completed and then checks its result. Timeout is 1 sec. Returns “true” (success) if command is completed and its result is S_RESULT_SUCCESS; otherwise, false.
bool HaltCpuCore()	Halt the target chip processor core
bool RunCpuCore()	Free run the target chip processor core

The implementation of these subroutines follows. It is based on pseudocode and registers defined in “Hardware Access Commands” in Section 3.3 and “Pseudocode” in Section 3.4. It uses constants defined in this chapter.

The pseudocode is similar to C-style notation.

```
// “WriteIO” Subroutine
bool WriteIO ( addr32, data32 )
{
ack1 = Write_DAP (TAR, addr32);
ack2 = Write_DAP (DRW, data32);
return (ack1 == 3’b001) && (ack2 == 3b’001);
}
```

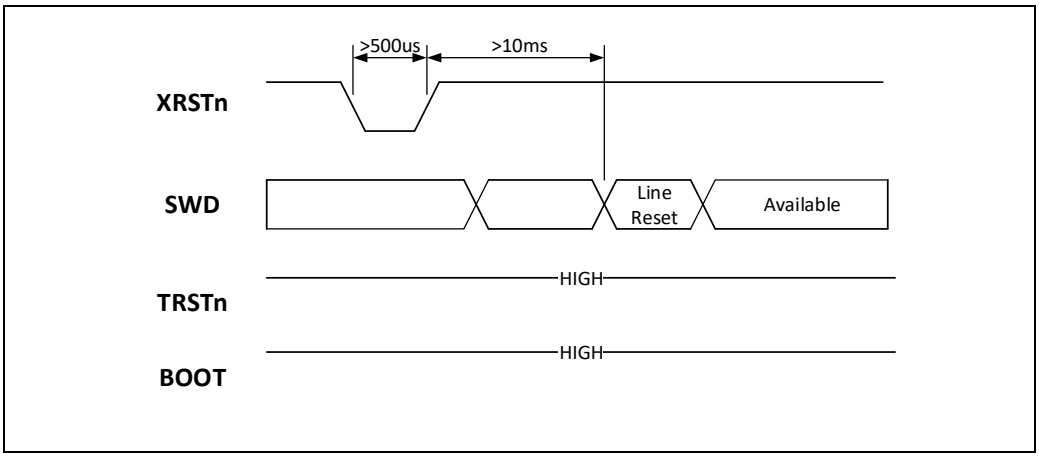
```
// “ReadIO” Subroutine
bool ReadIO ( addr32, OUT data32 )
{
ack1 = Write_DAP (TAR, addr32);
ack2 = Read_DAP (DRW, OUT data32);
ack3 = Read_DAP (DRW, OUT data32);
return (ack1 == 3’b001) && (ack2 == 3b’001) && (ack3 == 3b’001);
}

// “PollCmdStatus” Subroutine
bool PollCmdStatus()
{
do{
delay_ms(2);
ReadIO (S_STATUS, OUT status);
}while ((status != S_STATUS_DONE) && (time_elapsed < 5 sec));
if (time_elapsed >= 5 sec ) return false; // timeout
ReadIO (S_RESULT, OUT statusCode);
if (statusCode != S_RESULT_SUCCESS);
return false; // Programming command failed
else
return true; // Programming command succeeded
}

// “HaltCpuCore” Subroutine
bool HaltCpuCore()
{
// Halt processor core
wdata32 = (0xA05FU << 16) | 0x03U;
WriteIO (DHCSR, wdata32);

// Check
ReadIO (DHCSR, OUT rdata32);
If ((rdata32 & (0x1 << 17)) == 0) return false; // CPU not halt
Else return true; // CPU halted
}

// “RunCpuCore” Subroutine
bool RunCpuCore()
{
// Set Main Stack Pointer (MSP)
// MSP_VALUE is the first word data (byte 0 ~ 3, little-endian)
// of flash programming algorithm provided by SPINTROL
```

```
wdata32 = MSP_VALUE;
WriteIO (DCRDR, wdata32);

wdata32 = (1U << 16) | 0x11U; // Select MSP register
WriteIO (DCRSR, wdata32);

// Wait writing MSP register finished
do{
ReadIO (DHCSR, OUT rdata32);
}while ((rdata32 & 0x10000) == 0) && (time_elapsed < 1 sec));

if (time_elapsed >= 1 sec ) return false; // timeout

// Set Program Counter (PC)
// PC_VALUE is the second word data (byte 4 ~ 7, little-endian)
// of flash programming algorithm provided by SPINTROL
wdata32 = PC_VALUE;
WriteIO (DCRDR, wdata32);

wdata32 = (1U << 16) | 0x0FU; // Select PC register
WriteIO (DCRSR, wdata32);

// Wait writing PC register finished
do{
ReadIO (DHCSR, OUT rdata32);
}while ((rdata32 & 0x10000) == 0) && (time_elapsed < 1 sec));

if (time_elapsed >= 1 sec ) return false; // timeout

// Run processor
wdata32 = (0xA05FU << 16) | 0x01U;
return WriteIO (DHCSR, wdata32);
}
```

4.3 Step 1 – Initial Chip

The first step in the programming of the SPC11X8/SPD11X8 device is to reset it. After that, the external programmer sends a special sequence on the SWD port to line reset target chip SWD interface. Then, the target chip SWD interface is available for the programmer. The timing requirements of these process are shown in Figure 9.

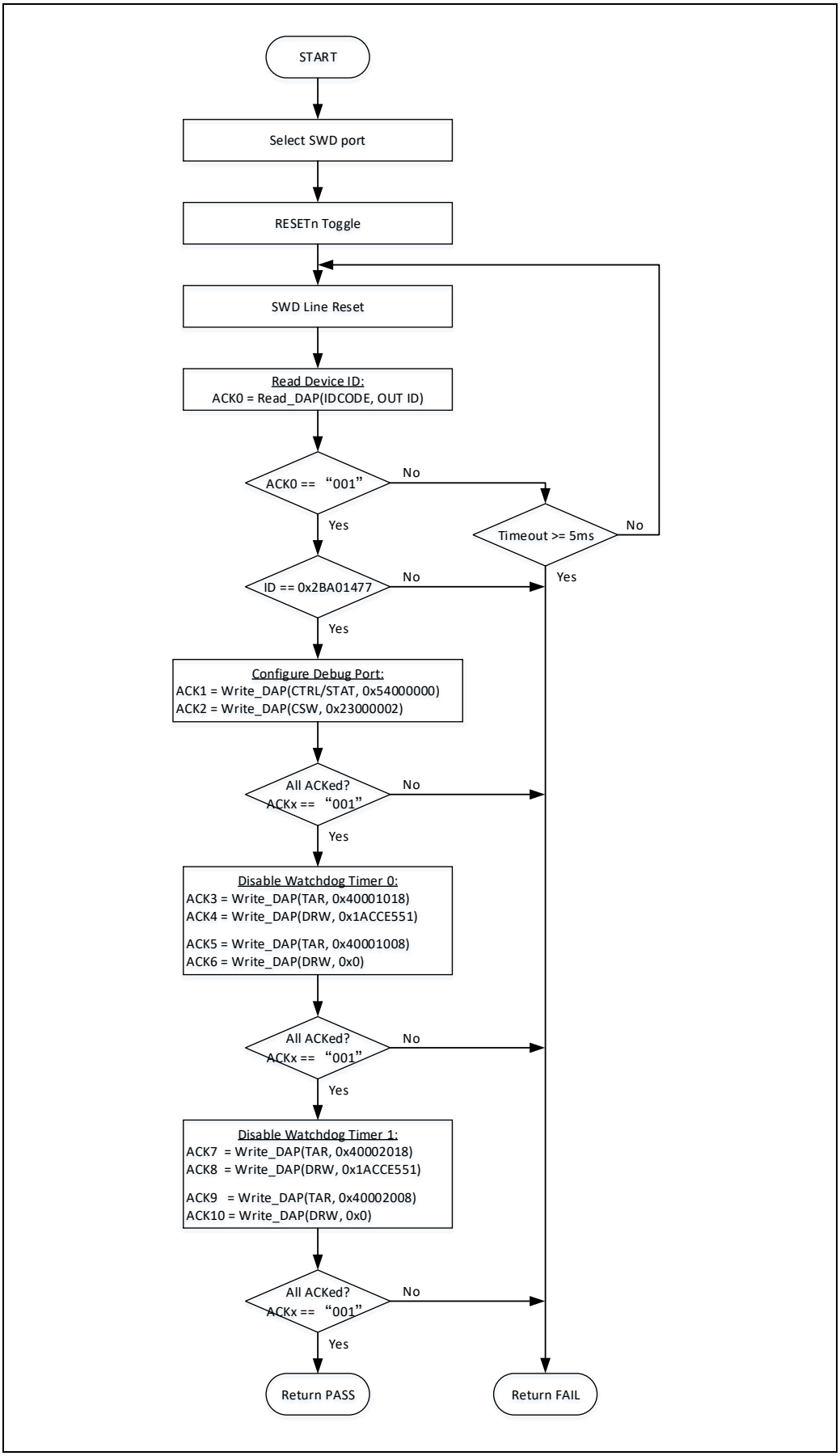
Figure 9. Timing Requirements of Initial Chip



The TRSTn pin and the BOOT pin should keep HIGH in the whole programming process. Otherwise, the chip will not work normal.

Figure 10 shows the implementation of initial chip procedure.

Figure 10. Flow Chart of Initial Chip Sequence



Pseudocode – Step 1. Acquire Chip

```
//-------------------------------------------------------------
// Reset Target chip
ToggleRESETn(); // Toggle RESETn pin, target must be powered.

//Execute ARM's connection sequence – acquire SWD-port.
Do
{
SWD_LineReset();

SWD_IdleCycle(5);
ack = Read_DAP ( IDCODE, out ID);
}While ((ack != 3b'001) && time_elapsed < 5 ms);

If (time_elapsed >= 5 ms) Return FAIL;
If (ID != 0x2BA01477) Return FAIL; //SWD ID of Cortex-M4 CPU.

//Initialize Debug Port
Write_DAP (CTRL/STAT, 0x54000000);
Write_DAP (CSW, 0x23000002);  // Access size = 32-bit

// Disable Watchdog Timer 0
ack = WriteIO (WDT0REGKEY, 0x1ACCE551);
if(ack != 3b'001) Return FAIL;
ack = WriteIO (WDT0CTL, 0x0);
if(ack != 3b'001) Return FAIL;

// Disable Watchdog Timer 1
ack = WriteIO (WDT1REGKEY, 0x1ACCE551);
if(ack != 3b'001) Return FAIL;
Return WriteIO (WDT1CTL, 0x0);

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

4.4 Step 2 – Transfer Flash Programming Algorithm

This step is used to transfer Flash Programming Algorithm to target chip SRAM area with address beginning at 0x20000000.

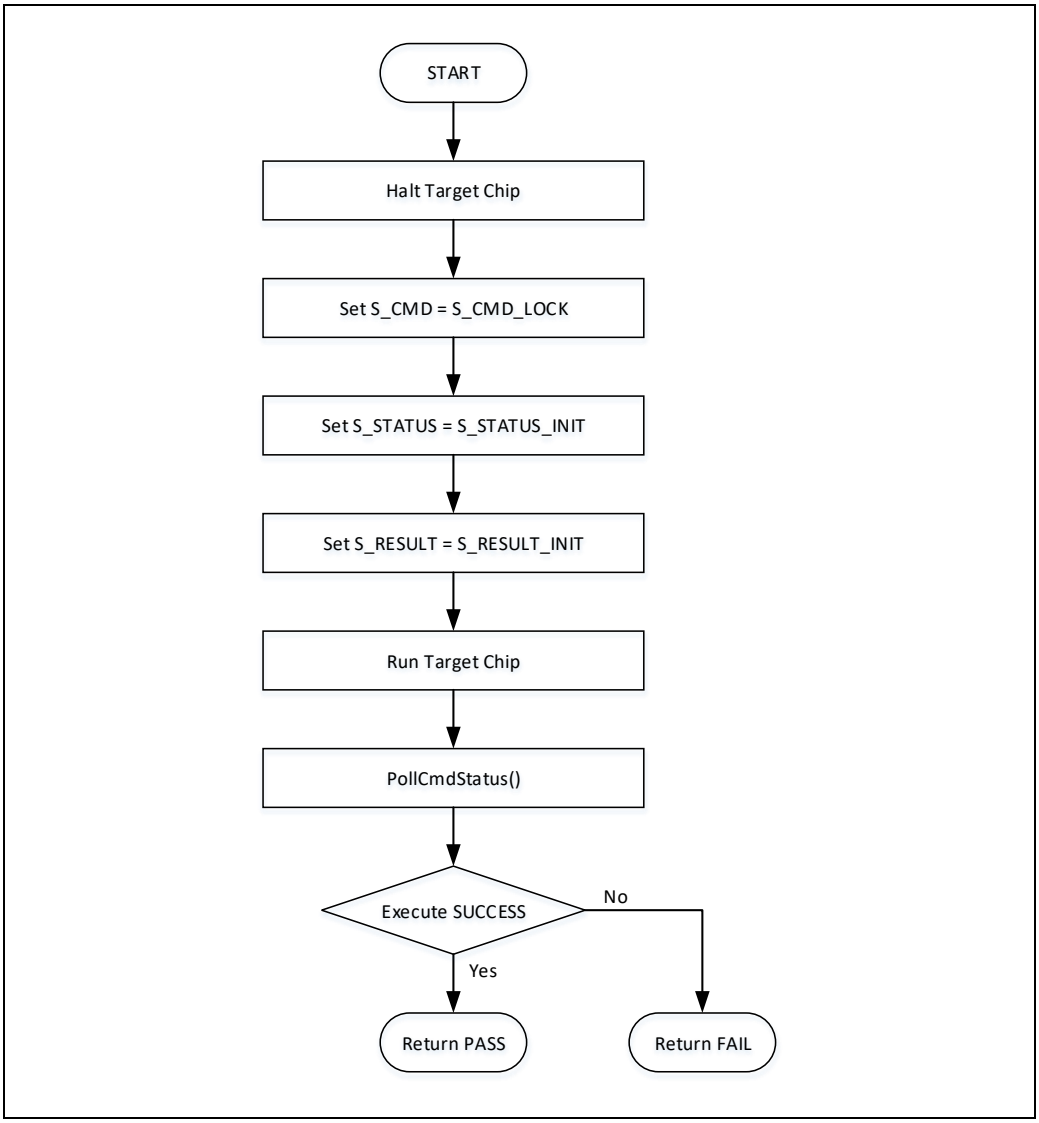
Pseudocode – Step 2. Transfer Flash Programming Algorithm

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

// Halt target chip processor core
HaltCpuCore();

// Transfer flash programming algorithm data
// length - flash programming algorithm data length in byte
// Array[] – byte array stored flash programming algorithm data
For(i = 0 ; i < length; i += 4)
{
data32 = (Array [i] << 0) + (Array [i + 1] << 8) +
          Array [i + 2] << 16) + (Array [i + 3] << 24);
WriteIO (0x20000000 + i, data32);
}

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

4.5 Step 3 – Lock System

This step locks the system parameters in the target chip. Figure 11 shows the implementation of lock system procedure.

Figure 11. Flow chart of Lock System



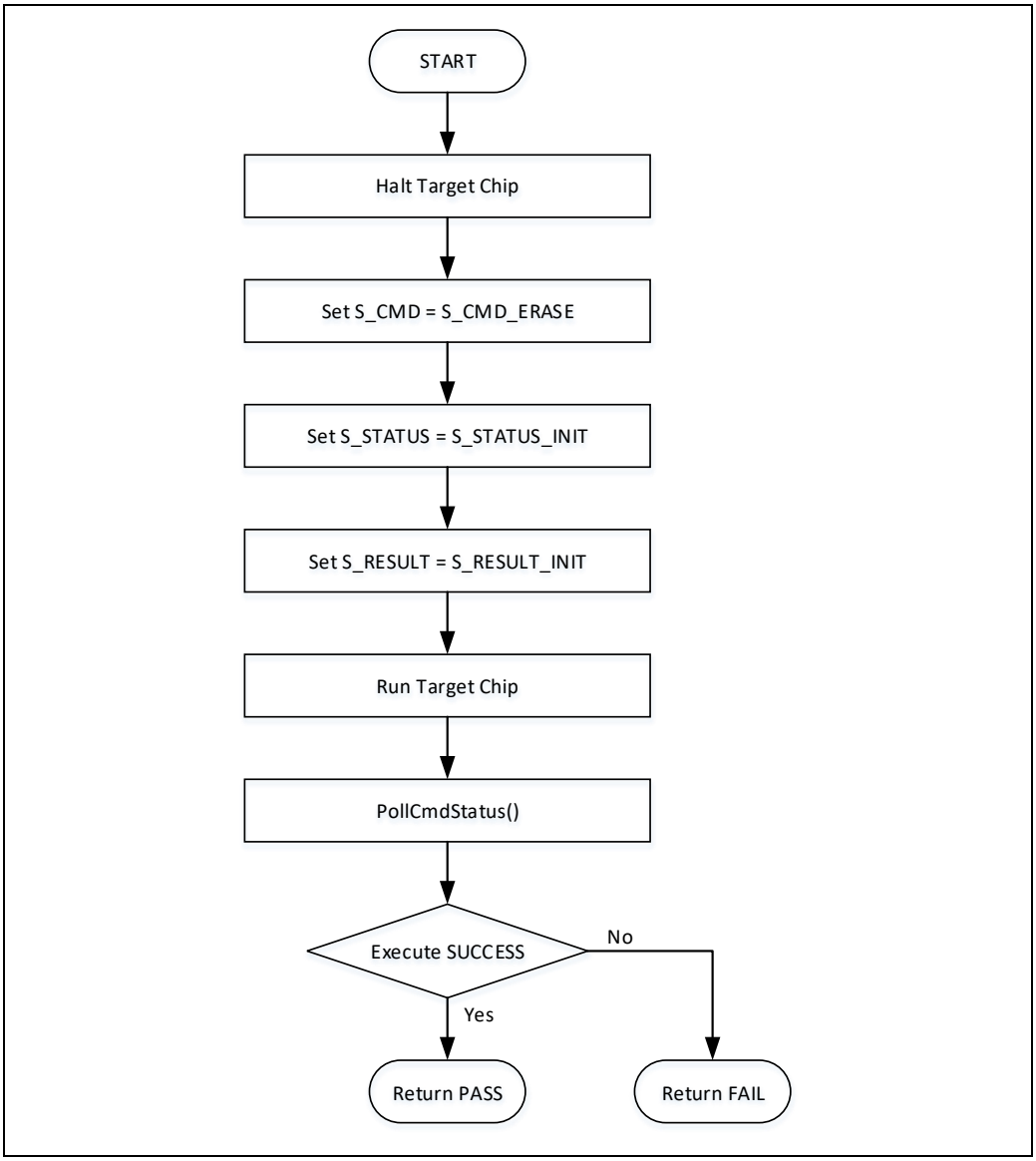
Pseudocode – Step 3. Lock System

```
//-------------------------------------------------------------
// Halt target chip processor core
HaltCpuCore();

// Set CMD
WriteIO (S_CMD, S_CMD_LOCK);

// Init Status
WriteIO (S_STATUS, S_STATUS_INIT);

// Init Result
WriteIO (S_RESULT, S_RESULT_INIT);

// Run target chip processor core
RunCpuCore();

// Wait until command executing finished
Return PollCmdStatus();

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

4.6 Step 4 – Erase All Flash

This step erases the whole flash chip of the target chip. After erasing the flash memory, it is recommended to check whether the target flash area data are all 0xFF (Blank Check). Figure 12 shows the process of erasing flash memory.

Figure 12. Flow chart of Erase All Flash



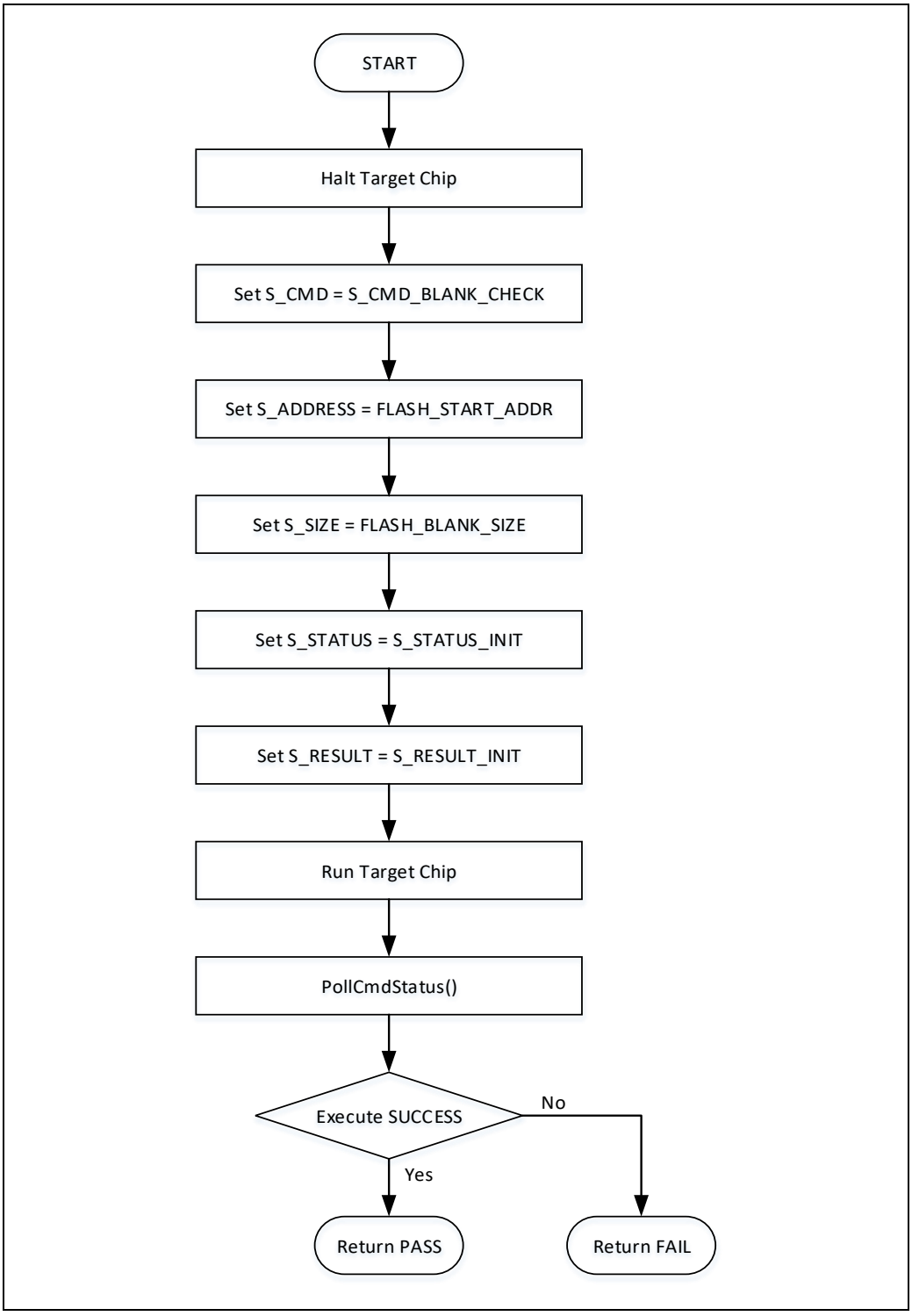
Pseudocode – Step 4. Erase All Flash

```
//-------------------------------------------------------------
// Halt target chip processor core
HaltCpuCore();

// Set CMD
WriteIO (S_CMD, S_CMD_ERASE);

// Init Status
WriteIO (S_STATUS, S_STATUS_INIT);

// Init Result
WriteIO (S_RESULT, S_RESULT_INIT);

// Run target chip processor core
RunCpuCore();

// Wait until command executing finished
Return PollCmdStatus();

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

Flash Blank Check

Figure 13 shows the process of Flash memory blank check. The FLASH_START_ADDR should be 256-byte aligned and the FLASH_BLANK_SIZE must be multiple of 256.

Figure 13. Flow chart of Flash Blank Check



Pseudocode – Flash Blank Check

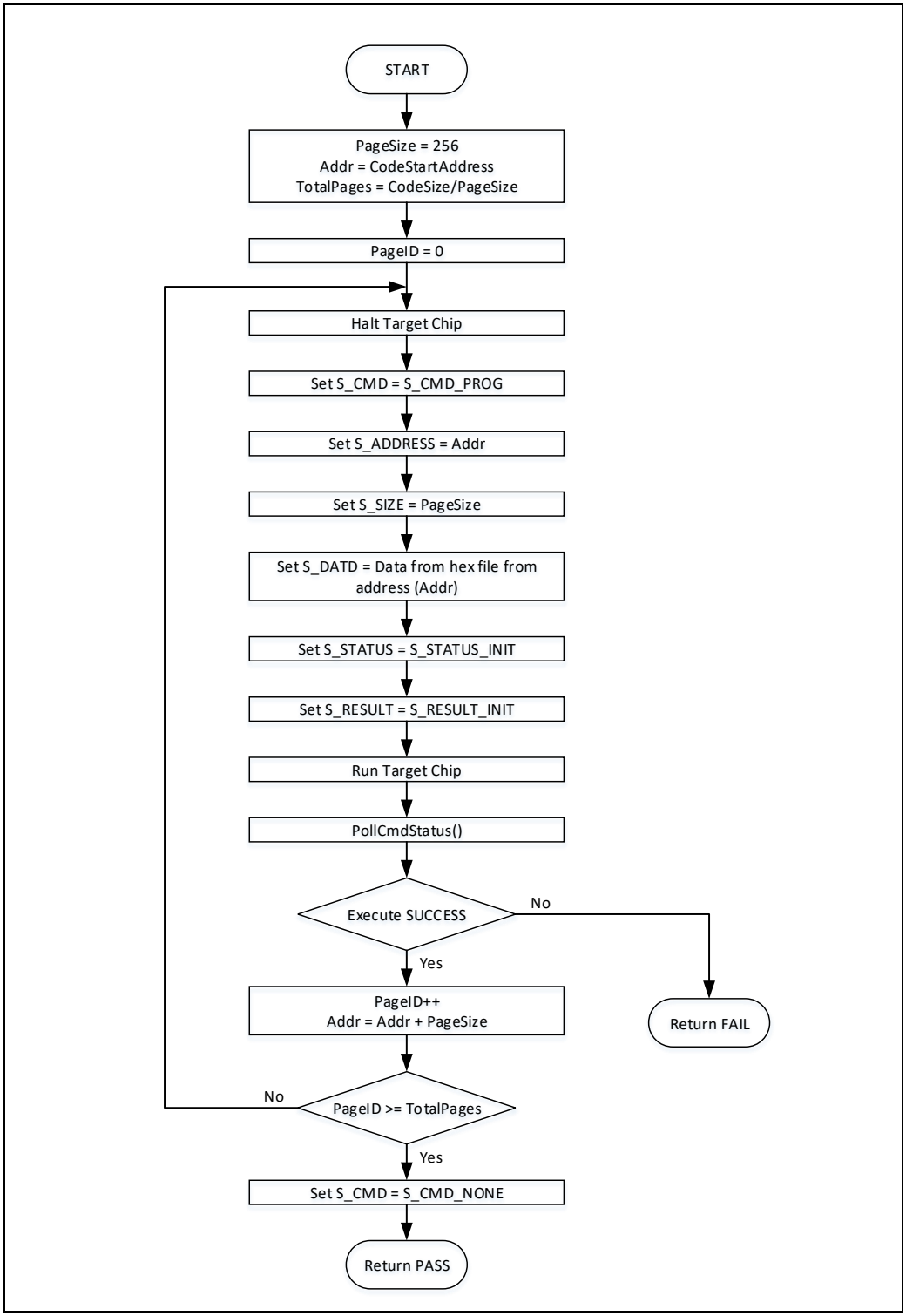
```
//-------------------------------------------------------------
// Halt target chip processor core
HaltCpuCore();

// Set CMD
WriteIO (S_CMD, S_CMD_BLANK_CHECK);

// Set Flash Blank Check start address
// for example 0x10000000 for Flash Main Block
//             0x11000400 for OTP Flash
//             0x11000600 for Configuration Data
WriteIO (S_ADDRESS, FLASH_START_ADDR);

// Set Flash Size (in bytes) for Blank Check
WriteIO (S_SIZE, FLASH_BLANK_SIZE);

// Init Status
WriteIO (S_STATUS, S_STATUS_INIT);

// Init Result
WriteIO (S_RESULT, S_RESULT_INIT);

// Run target chip processor core
RunCpuCore();

// Wait until command executing finished
Return PollCmdStatus();

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

For example, if S_ADDRESS = 0x10000000, S_SIZE = 0x10000, the flash area 0x10000000 ~ 0x1000ffff will be blank-checking; if S_ADDRESS = 0x11000400, S_SIZE = 0x200, the OTP flash area 0x11000400 ~ 0x100005ff will be blank-checking; if S_ADDRESS = 0x11000600, S_SIZE = 0x200, the Configuration Data flash area 0x11000600 ~ 0x100007ff will be blank-checking.

4.7 Step 5 – Program Flash

Flash memory of SPC11X8/SPD11X8 is programmed in pages. Each page is 256 bytes long. The programmer must serially program each page one by one. The user code start address and size can be extracted from the user hex file. The user code size must be multiple of 256.

Figure 14. Flow chart of Program Flash



Pseudocode – Step 5. Program Flash

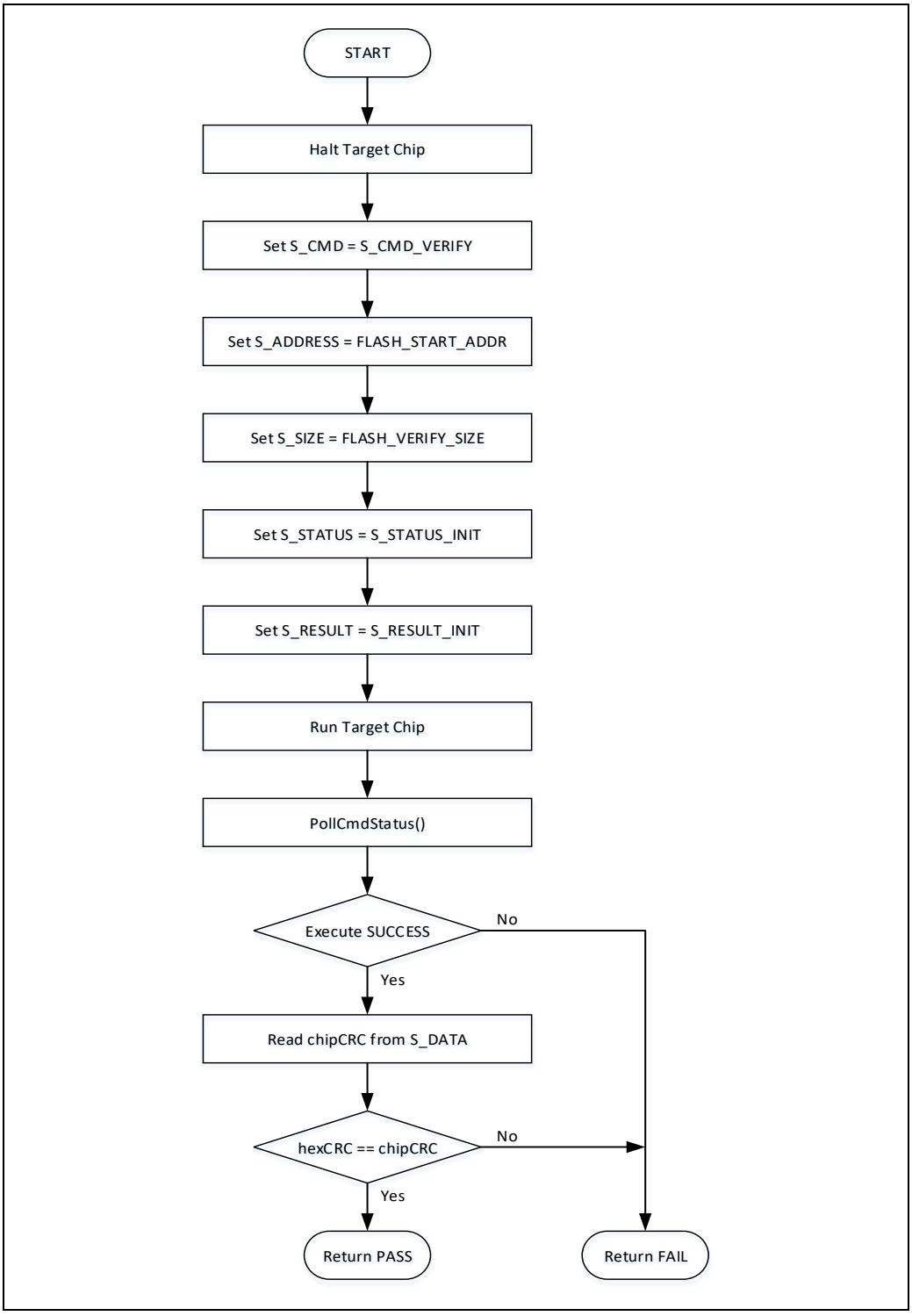
```
//-------------------------------------------------------
// User code start address and code size can be calculated from
// user code hex file
PageSize = 256;
TotalPages = CodeSize/PageSize;
Addr = CodeStartAddress;

// Program user code page by page
For(PageID = 0; PageID < TotalPages; PageID++)
{
    // Halt target chip processor core
    HaltCpuCore();

    // Set CMD
    WriteIO (S_CMD, S_CMD_PROG);

    // Set address to be programmed
    WriteIO (S_ADDRESS, Addr);

    // Set data size to be programmed
    WriteIO (S_SIZE, PageSize);


    //Extract 256-byte data from the hex-file
    //from address: Addr into buffer - “Data”.
    //HEX_ReadData() must be implemented by Programmer.
    Data = HEX_ReadData( Addr, PageSize);

    // Set data to be programmed
    For(i = 0; i < PageSize; i += 4)
    {
    data32 = (Data[i] << 0) + (Data[i + 1] << 8) +
              Data[i + 2] << 16) + (Data[i + 3] << 24);
    WriteIO (S_DATA + i, data32);
    }

    // Init Status
    WriteIO (S_STATUS, S_STATUS_INIT);

    // Init Result
    WriteIO (S_RESULT, S_RESULT_INIT);
```

```
    // Run target chip processor core
    RunCpuCore();

    // Wait until command executing finished
    Status = PollCmdStatus();
    If(Status == FALSE) return FAIL;

    Addr += PageSize;
}

// Set CMD
WriteIO (S_CMD, S_CMD_NONE);

Return PASS;

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

4.8 Step 6 – Verify Flash

This step is used to guarantee that the user code is written without any errors. Figure 15 shows the process of Verify Flash. FLASH_START_ADDR should be 256-byte aligned and the FLASH_VERIFY_SIZE must be multiple of 256.

Figure 15. Flow chart of Verify Flash



The programmer reads the CRC value of the code data programmed to the target chip flash and then compares it with the CRC value of the user hex data. If not equal, the programmer must stop and return a failure.

In this chapter, the CRC standard is CRC-32-IEEE802.3 and the polynomial is:

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$$

The corresponding C code is as follows and the programmer can use it to calculate the CRC value of user hex data.

```
uint do_crc32(byte[] message, uint len)
{
  int i, j;
  uint crc_reg = 0;
  uint current;

  for (i = 0; i < len; i++)
  {
     current = message[i];
     for (j = 0; j < 8; j++)
     {
         if (((crc_reg ^ current) & 0x0001) != 0)
         {
             crc_reg = (crc_reg >> 1) ^ 0xEDB88320;
         }
         else
         {
             crc_reg >>= 1;
         }
         current >>= 1;
     }
  }
  return crc_reg;
}
```

Pseudocode – Step 6. Verify Flash

```
//-------------------------------------------------------------
//1. Calculate the CRC value of the code data in hex-file
//HEX_CalculateCRC() must be implemented by Programmer.
hexCRC = HEX_CalculateCRC();

//2. Verify data in the target chip

// Halt target chip processor core
HaltCpuCore();

// Set CMD
WriteIO (S_CMD, S_CMD_VERIFY);

// Set address to verify from
WriteIO (S_ADDRESS, FLASH_START_ADDR);

// Set data size to verify
WriteIO (S_SIZE, FLASH_VERIFY_SIZE);

// Init Status
WriteIO (S_STATUS, S_STATUS_INIT);

// Init Result
WriteIO (S_RESULT, S_RESULT_INIT);

// Run target chip processor core
RunCpuCore();

// Wait until command executing finished
Status = PollCmdStatus();
If(Status == FALSE) return FAIL;

// Read CRC value in the target chip (stored in S_DATA)
ReadIO (S_DATA, out data32);

chipCRC = data32;

//3. Compare them
If (chipCRC != hexCRC) return FAIL;

// Set CMD
WriteIO (S_CMD, S_CMD_NONE);
```

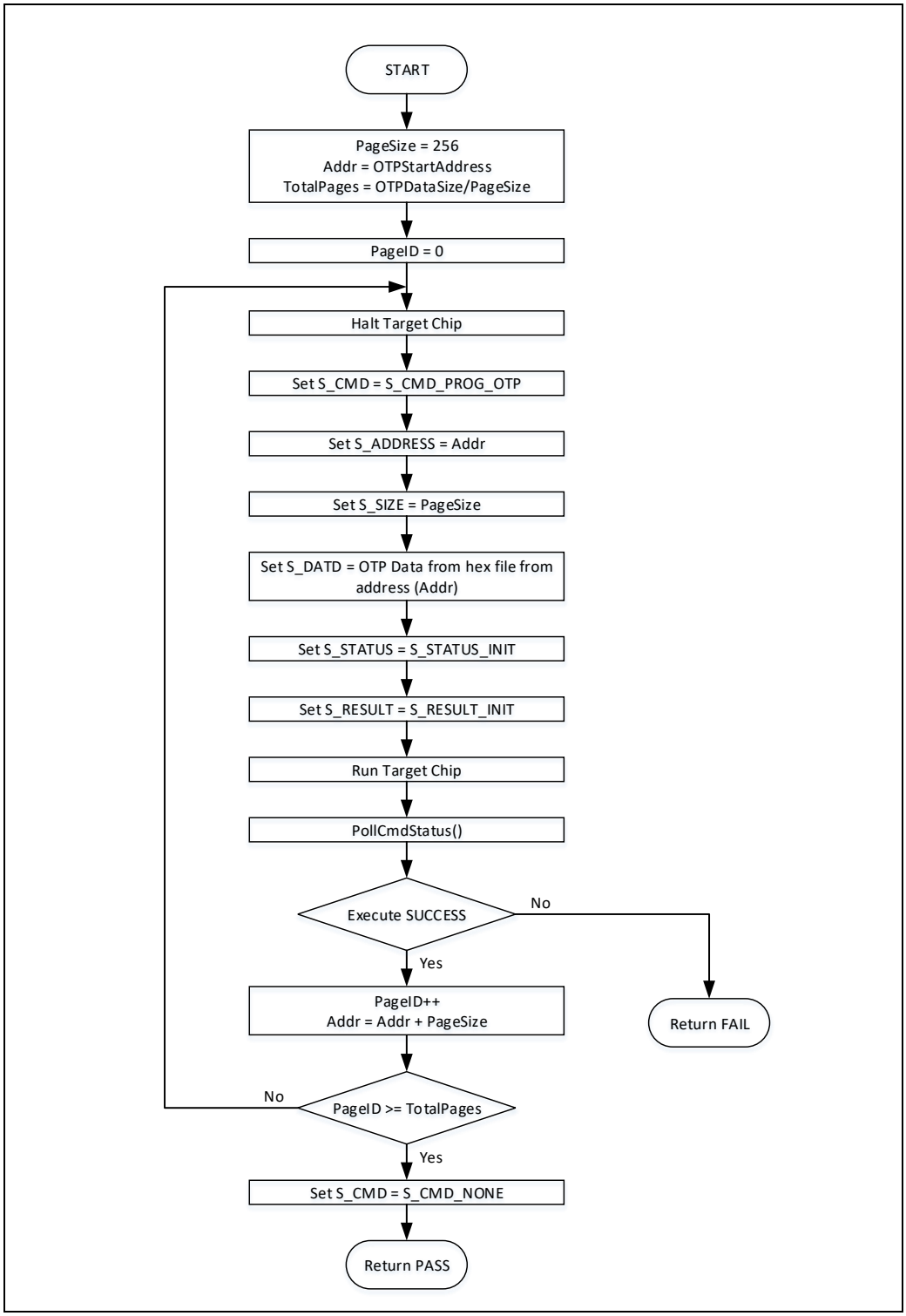
```
Return PASS;

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

For example, if S_ADDRESS = 0x10000000, S_SIZE = 0x10000, the CRC value of the data in flash area 0x10000000 ~ 0x1000ffff will be calculated and verified.

4.9 Step 7 – Program OTP Flash

This step is used to transfer OTP data to target chip. The OTP data start address and size can be extracted from the user hex file. Figure 16 shows the implementation of Program OTP Flash procedure.

Figure 16. Flow chart of Program OTP Flash



Pseudocode – Step 7. Program OTP Flash

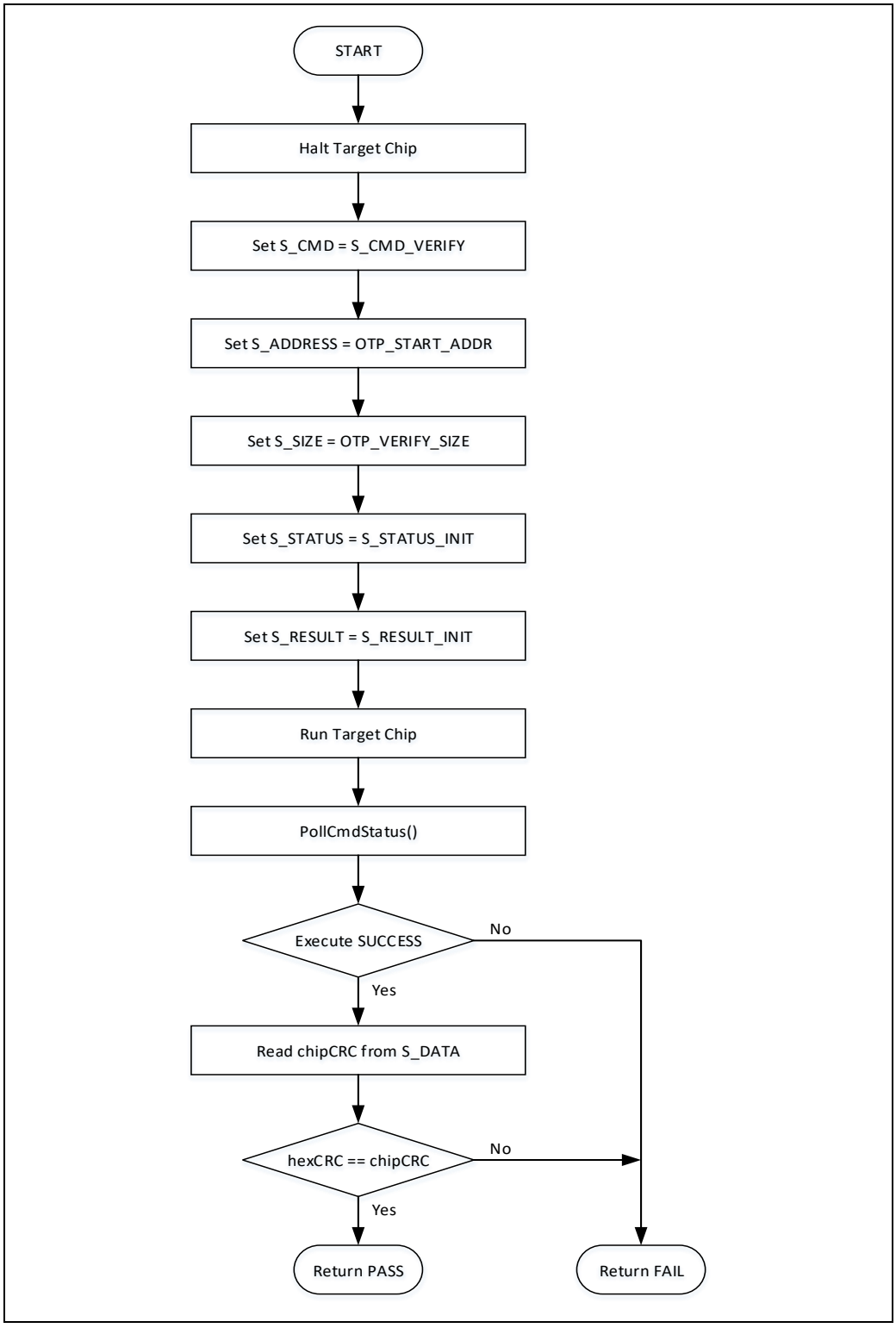
```
//-------------------------------------------------------
// OTP data start address and data size can be calculated from
// user hex file
// OTP data is located at address from 0x11000400
// Note: If 'OTPDataSize' is not multiple of 256, the last page
// size may be not 256, please set 'PageSize' as the actual size
// of the last page
PageSize = 256;
TotalPages = OTPDataSize/PageSize;
Addr = OTPStartAddress;

// Program OTP data page by page
For(PageID = 0; PageID < TotalPages; PageID++)
{
    // Halt target chip processor core
    HaltCpuCore();

    // Set CMD
    WriteIO (S_CMD, S_CMD_PROG);

    // Set address to be programmed
    WriteIO (S_ADDRESS, Addr);

    // Set data size to be programmed
    WriteIO (S_SIZE, PageSize);


    //Extract 256-byte data from the hex-file
    //from address: Addr into buffer - "Data".
    //HEX_ReadOTPData() must be implemented by Programmer.
    Data = HEX_ReadOTPData( Addr, PageSize);

    // Set data to be programmed
    For(i = 0; i < PageSize; i += 4)
    {
    data32 = (Data[i] << 0) + (Data[i + 1] << 8) +
              Data[i + 2] << 16) + (Data[i + 3] << 24);
    WriteIO (S_DATA + i, data32);
    }

    // Init Status
    WriteIO (S_STATUS, S_STATUS_INIT);
```

```
    // Init Result
    WriteIO (S_RESULT, S_RESULT_INIT);

    // Run target chip processor core
    RunCpuCore();

    // Wait until command executing finished
    Status = PollCmdStatus();
    If(Status == FALSE) return FAIL;

    Addr += PageSize;
}

// Set CMD
WriteIO (S_CMD, S_CMD_NONE);

Return PASS;

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

4.10 Step 8 – Verify OTP Flash

This step is used to guarantee that the OTP data is written without any errors. Figure 17 shows the process of Verify OTP Flash.

Figure 17. Flow chart of Verify OTP Flash



The programmer reads the CRC value of the OTP data programmed to the target chip OTP flash and then compares it with the CRC value of the user hex data. If not equal, the programmer must stop and return a failure.

In this chapter, the CRC standard is CRC-32-IEEE802.3. Please see Step 6 – Verify Flash for details.

Pseudocode – Step 8. Verify OTP Flash

```
//-------------------------------------------------------------
//1. Calculate the CRC value of OTP data in the hex-file
//HEX_CalculateOTPCRC() must be implemented by Programmer.
hexCRC = HEX_CalculateOTPCRC();

//2. Verify OTP data in the target chip

// Halt target chip processor core
HaltCpuCore();

// Set CMD
WriteIO (S_CMD, S_CMD_VERIFY);

// Set address to verify from
WriteIO (S_ADDRESS, OTP_START_ADDR);

// Set data size to verify
WriteIO (S_SIZE, OTP_VERIFY_SIZE);

// Init Status
WriteIO (S_STATUS, S_STATUS_INIT);

// Init Result
WriteIO (S_RESULT, S_RESULT_INIT);

// Run target chip processor core
RunCpuCore();

// Wait until command executing finished
Status = PollCmdStatus();
If(Status == FALSE) return FAIL;

// Read CRC value in the target chip (stored in S_DATA)
ReadIO (S_DATA, out data32);

chipCRC = data32;

//3. Compare them
If (chipCRC != hexCRC) return FAIL;

// Set CMD
WriteIO (S_CMD, S_CMD_NONE);
```

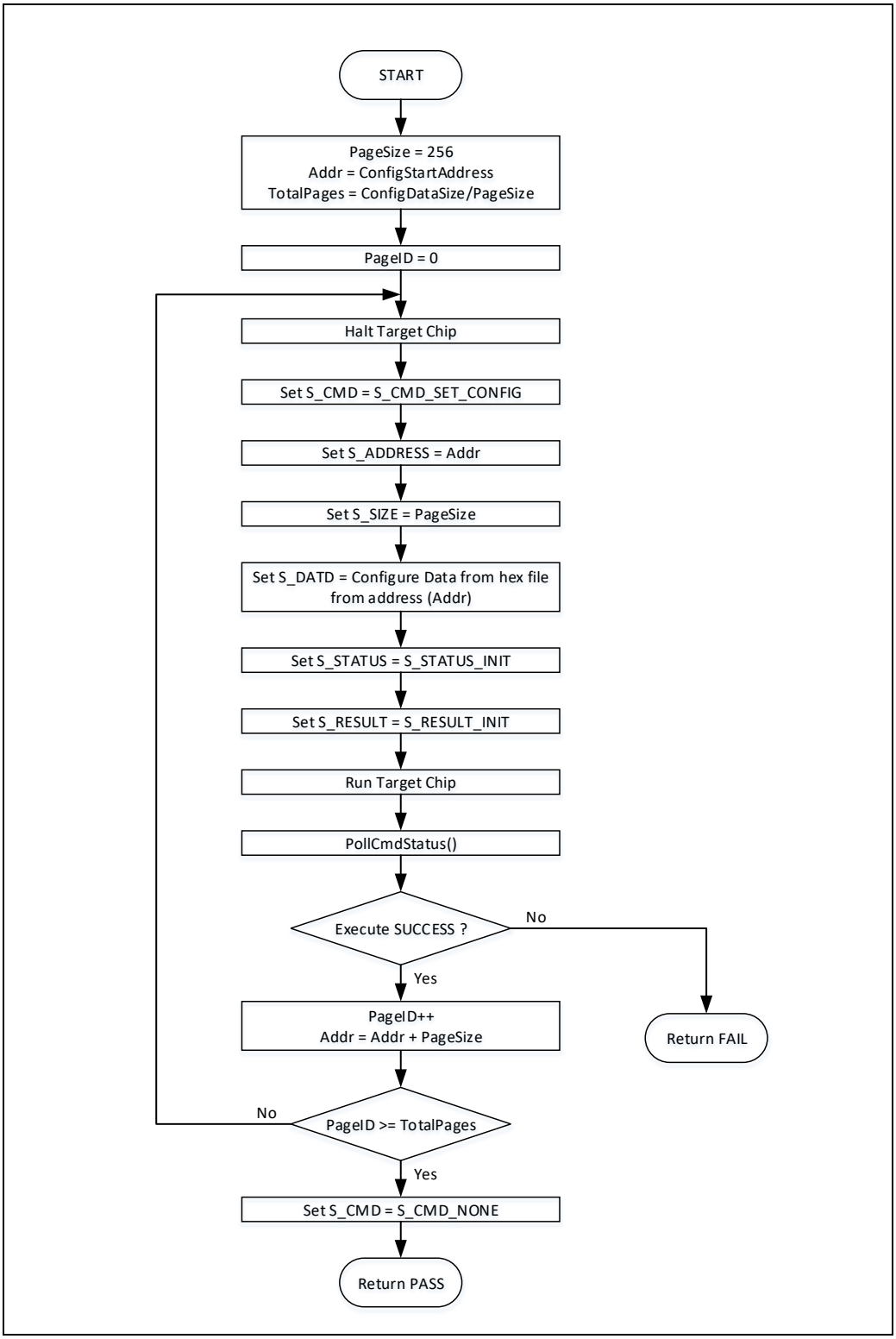
```
Return PASS;

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

For example, if S_ADDRESS = 0x11000400, S_SIZE = 0x108, the CRC value of the data in OTP flash area 0x11000400 ~ 0x11000507 will be calculated and verified.

4.11 Step 9 – Set Configuration Data

This step is used to transfer chip Configuration Words data to target chip. Figure 18 shows the implementation of Set Configuration Data procedure.

Figure 18. Flow chart of Set Configuration Data



Pseudocode – Step 9. Set Configuration Data

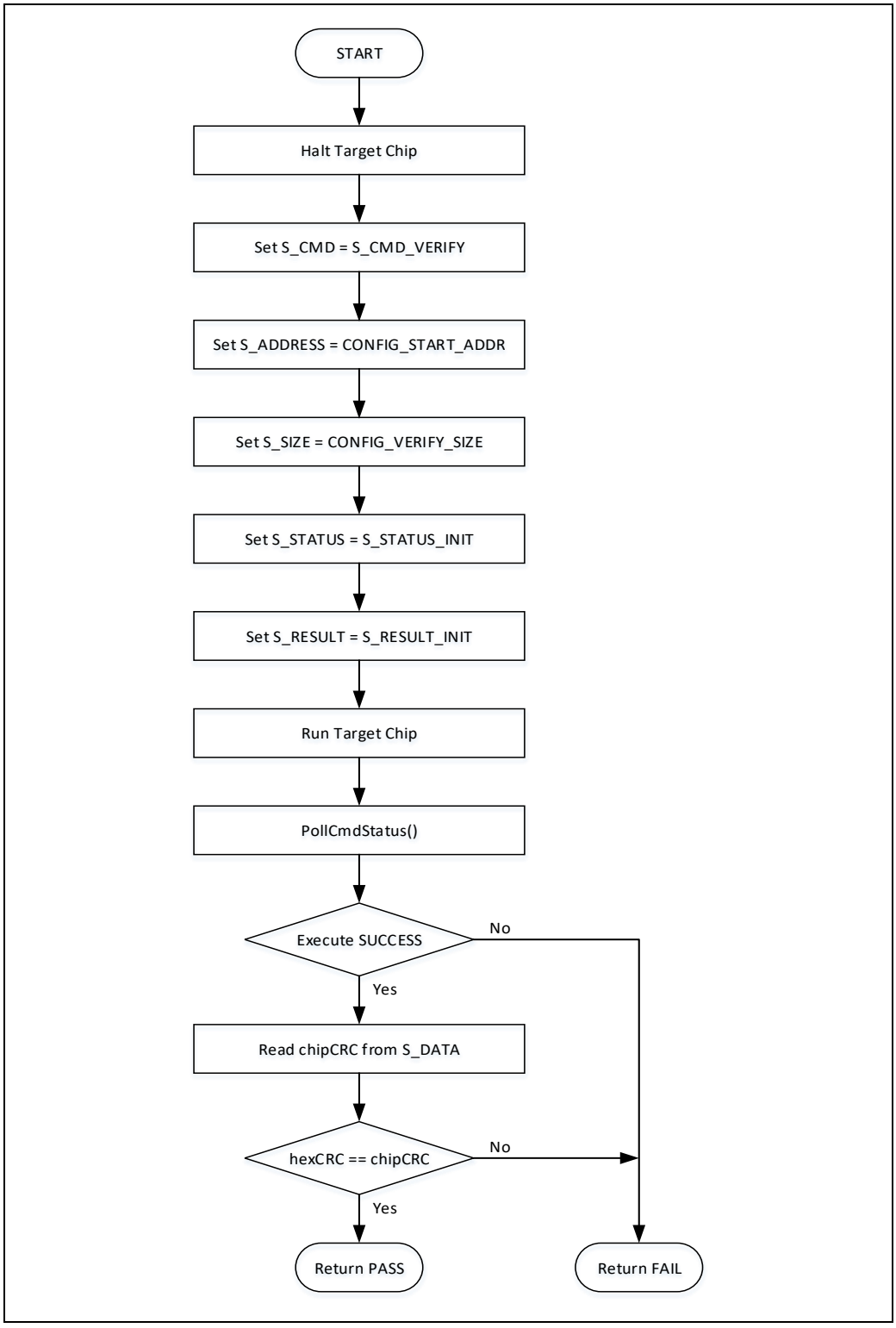
```
//-------------------------------------------------------
// Configuration data start address and data size can be calculated
// from user hex file
// Configuration data is located at address from 0x11000600
// Note: If 'ConfigDataSize' is not multiple of 256, the last page
// size may be not 256, please set 'PageSize' as the actual size
// of the last page
PageSize = 256;
TotalPages = ConfigDataSize/PageSize;
Addr = ConfigStartAddress;

// Program Configuration data page by page
For(PageID = 0; PageID < TotalPages; PageID++)
{
   // Halt target chip processor core
   HaltCpuCore();

   // Set CMD
   WriteIO (S_CMD, S_CMD_PROG);

   // Set address to be programmed
   WriteIO (S_ADDRESS, Addr);

   // Set data size to be programmed
   WriteIO (S_SIZE, PageSize);


   //Extract 256-byte data from the hex-file
   //from address: Addr into buffer - "Data".
   //HEX_ReadConfigData() must be implemented by Programmer.
   Data = HEX_ReadConfigData( Addr, PageSize);

   // Set data to be programmed
   For(i = 0; i < PageSize; i += 4)
   {
   data32 = (Data[i] << 0) + (Data[i + 1] << 8) +
            Data[i + 2] << 16) + (Data[i + 3] << 24);
   WriteIO (S_DATA + i, data32);
   }

   // Init Status
   WriteIO (S_STATUS, S_STATUS_INIT);
```

```
    // Init Result
    WriteIO (S_RESULT, S_RESULT_INIT);

    // Run target chip processor core
    RunCpuCore();

    // Wait until command executing finished
    Status = PollCmdStatus();
    If(Status == FALSE) return FAIL;

    Addr += PageSize;
}

// Set CMD
WriteIO (S_CMD, S_CMD_NONE);

Return PASS;

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

4.12 Step 10 – Verify Configuration Data

This step is used to guarantee that the Configuration data is written without any errors. Figure 19 shows the process of Verify Configuration Data.

Figure 19. Flow chart of Verify Configuration Data



The programmer reads the CRC value of the Configuration data programmed to the target chip and then compares it with the CRC value of the user hex data. If not equal, the programmer must stop and return a failure.

In this chapter, the CRC standard is CRC-32-IEEE802.3. Please see Step 6 – Verify Flash for details.

Pseudocode – Step 10. Verify Configuration Data

```
//-------------------------------------------------------------
//1. Calculate the CRC value of Configuration data in the hex-file
//HEX_CalculateConfigCRC() must be implemented by Programmer.
hexCRC = HEX_CalculateConfigCRC();

//2. Verify Configuration data in the target chip

// Halt target chip processor core
HaltCpuCore();

// Set CMD
WriteIO (S_CMD, S_CMD_VERIFY);

// Set address to verify from
WriteIO (S_ADDRESS, CONFIG_START_ADDR);

// Set data size to verify
WriteIO (S_SIZE, CONFIG_VERIFY_SIZE);

// Init Status
WriteIO (S_STATUS, S_STATUS_INIT);

// Init Result
WriteIO (S_RESULT, S_RESULT_INIT);

// Run target chip processor core
RunCpuCore();

// Wait until command executing finished
Status = PollCmdStatus();
If(Status == FALSE) return FAIL;

// Read CRC value in the target chip (stored in S_DATA)
ReadIO (S_DATA, out data32);

chipCRC = data32;

//3. Compare them
If (chipCRC != hexCRC) return FAIL;

// Set CMD
WriteIO (S_CMD, S_CMD_NONE);
```

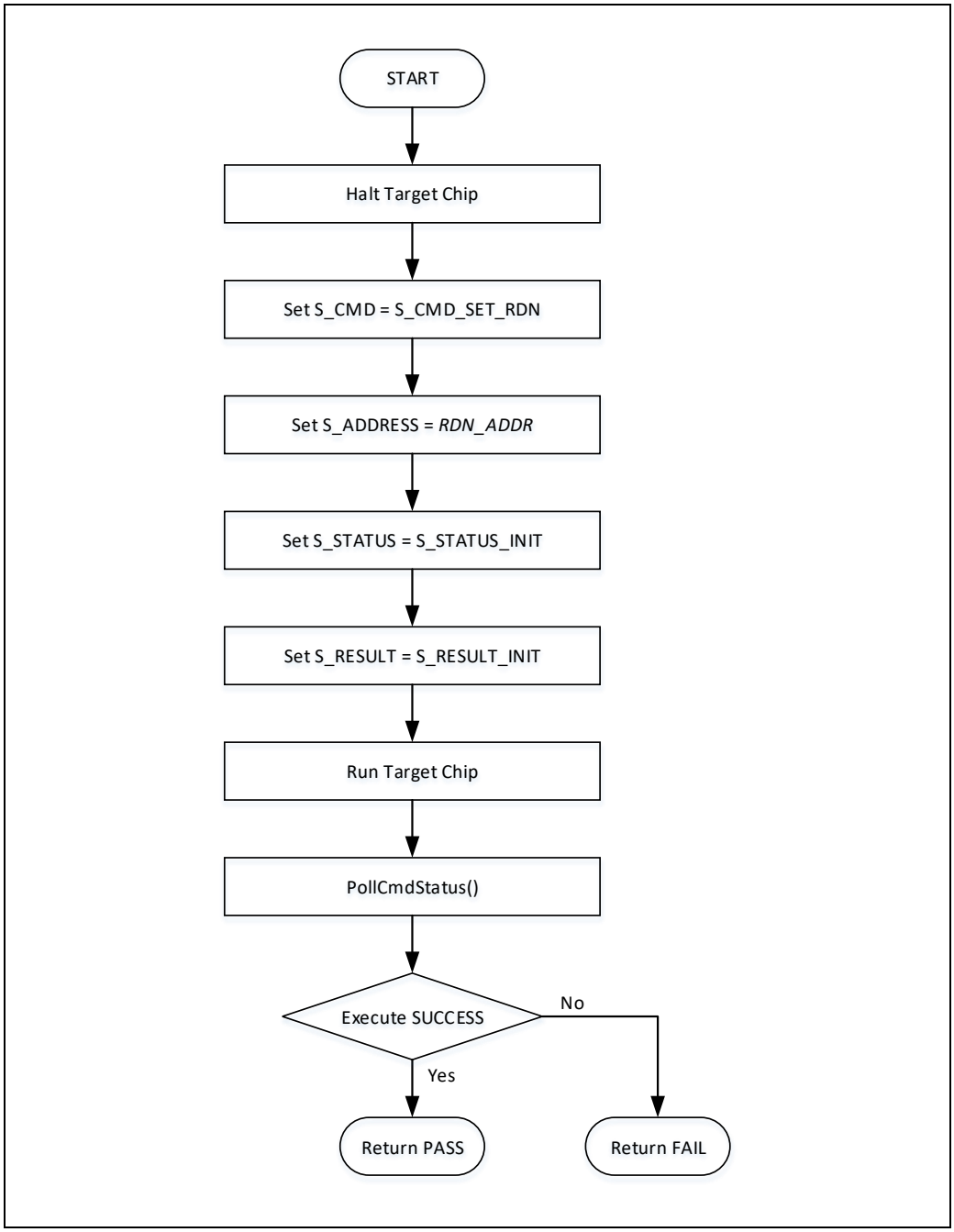
```
Return PASS;

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

For example, if S_ADDRESS = 0x11000600, S_SIZE = 0x104, the CRC value of the data in Configure Data area 0x11000600 ~ 0x11000703 will be calculated and verified.

4.13 Step 11 – Set Random Number Protect

This step is used to set target chip random number protect. The programmer will transfer the address to the target chip, which the random numbers to be programmed to. The address should be 4-byte aligned. Figure 20 shows the process of Set Random Number Protect. **Note that RDN_ADDR is the address that the random numbers to be programmed to and the customers should define the value of RDN_ADDR.**

Figure 20. Flow chart of Set Random Number Protect



Pseudocode – Set Random Number Protect

```
//-----------------------------------------------------------
// Halt target chip processor core
HaltCpuCore();

// Set CMD
WriteIO (S_CMD, S_CMD_SET_RDN);

// Set the address that random number to be programmed to
// The actual value of RDN_ADDR should be defined by customers
WriteIO (S_ADDRESS, RDN_ADDR);

// Init Status
WriteIO (S_STATUS, S_STATUS_INIT);

// Init Result
WriteIO (S_RESULT, S_RESULT_INIT);

// Run target chip processor core
RunCpuCore();

// Wait until command executing finished
Return PollCmdStatus();

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

Appendix A. Electrical Specifications

Table 9. Recommended operating conditions

Symbol	Parameter	Conditions	Min	Nom	Max	Unit
V_{DD}	Supply voltage	-	2.97	3.3	3.63	V
V_{SS}	Supply ground	-	-	0	-	V
V_{DDA}	Analog supply voltage	-	2.97	3.3	3.63	V
V_{SSA}	Analog ground	-	-	0	-	V
V_{IH}	High-level input voltage	V_{DD} = 3.3 V	2.0	-	V_{DD}+0.3	V
V_{IL}	Low-level input voltage	V_{DD} = 3.3 V	V_{SS}-0.3	-	0.8	V
I_{OH}	High-level output source current when V_{OH} = $V_{OH(MIN)}$	STRENGTH=0 STRENGTH=1 STRENGTH=2 STRENGTH=3	-	-	5 10 15 20	mA
I_{OL}	Low-level output sink current when V_{OL} = $V_{OL(MAX)}$	STRENGTH=0 STRENGTH=1 STRENGTH=2 STRENGTH=3	-	-	5 10 15 20	mA
T_J	Junction temperature	-	-40	-	+125	°C

Table 10. Electrical characteristics

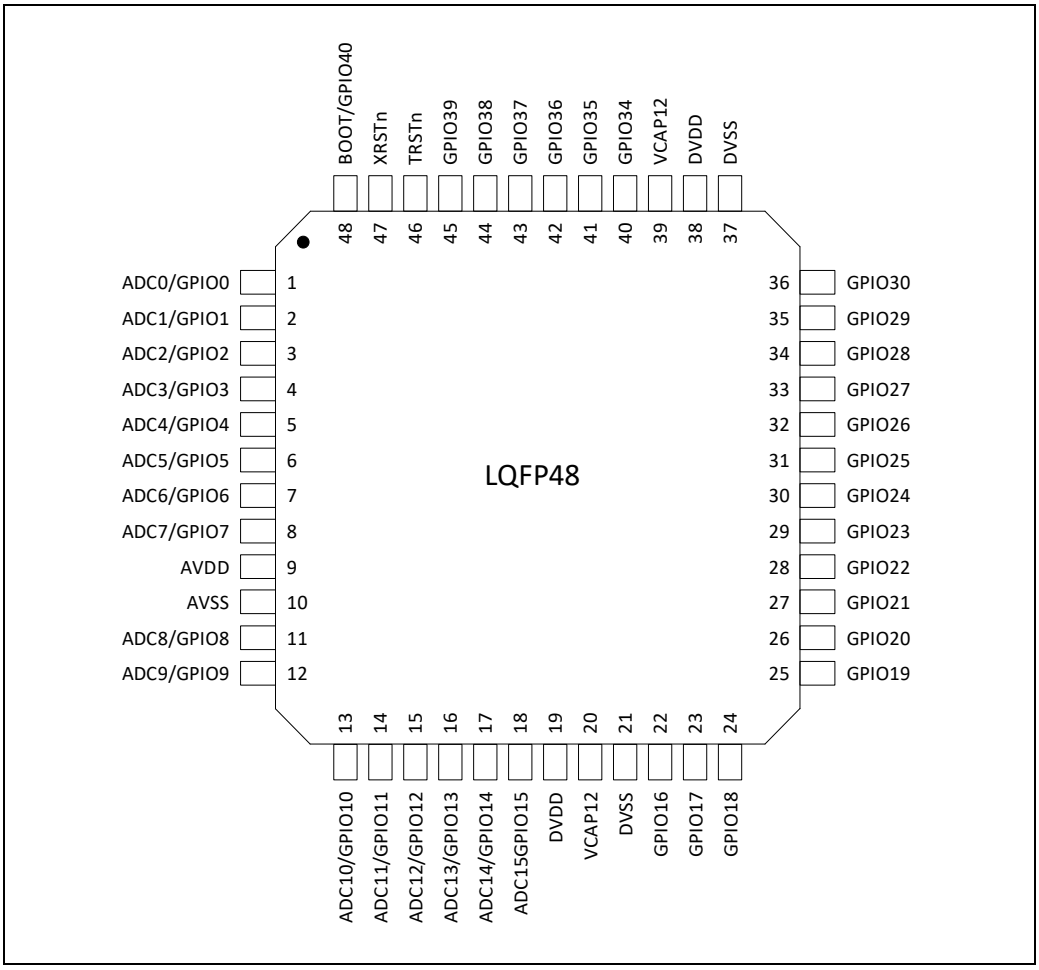
Symbol	Parameter	Conditions	Min	Typ	Max	Unit
V_{OH}	High-level output voltage	I_{OH} = I_{OH} MAX	VDD-0.4	-	-	V
V_{OL}	Low-level output voltage	I_{OL} = I_{OL} MAX	-	-	0.4	V
I_{IL}	Low-level input current (Pin with pull-up enabled)	V_{DD} = 3.3V, V_{IH} = 0 V	-	-	10	uA
I_{IH}	High-level input current (Pin with pull-up enabled)	V_{DD} = 3.3V, V_{IH} = V_{DD}	-	-	10	uA
I_{OZ}	Output current tri-state (Pin with pull-up disabled)	V_{DD} = 3.3V V_O = V_{DD}/0V	-	-	1	uA

Table 11. AC characteristics

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
T_{XRST}	External reset pulse width	-	500	-	-	us

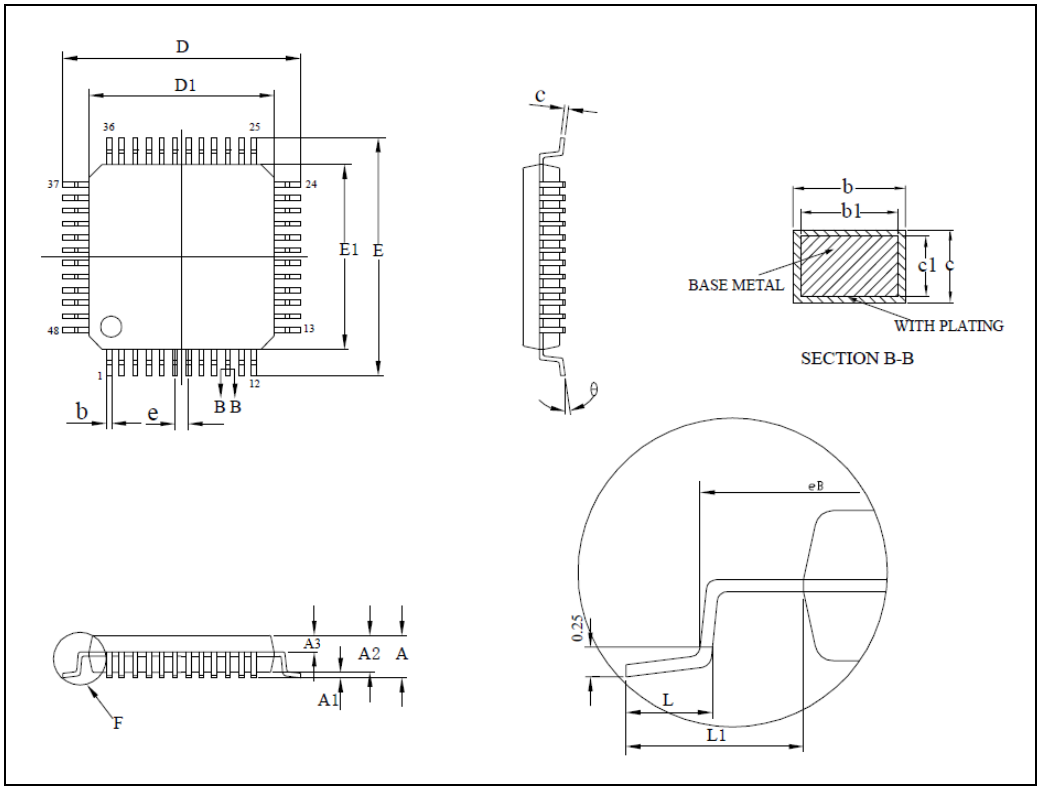
Appendix B. SPC1168 pinouts and package information

Figure 21. SPC1168 LQFP48 pinout



(1) The above figure shows the package top view.

Figure 22. LQFP48 – 48 pin, 7 x 7 mm low-profile quad flat package outline



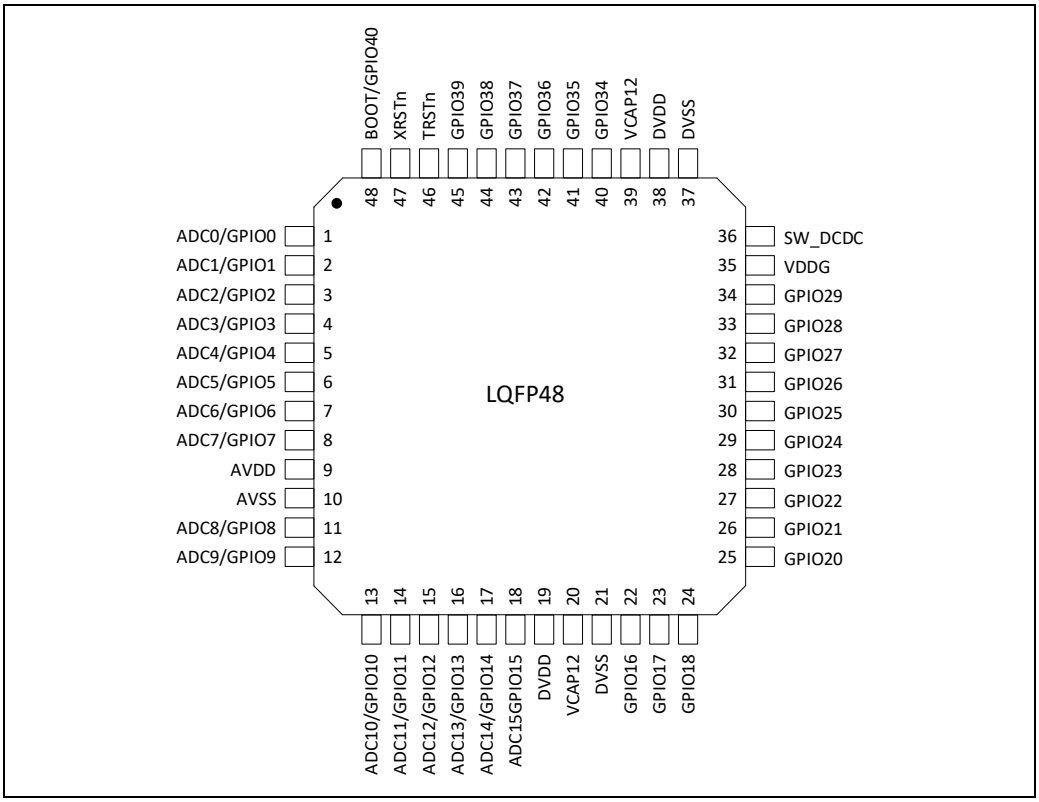
(1) Drawing is not to scale.

Table 12. LQFP48 - 48 pin, 7 x 7 mm low-profile quad flat package mechanical data

Symbol	Millimeters (mm)		
	Min	Typ	Max
A	-	-	1.60
A1	0.05	-	0.15
A2	1.35	1.40	1.45
A3	0.59	0.64	0.69
b	0.19	-	0.27
b1	0.18	0.20	0.23
c	0.13	-	0.18
c1	0.12	0.13	0.14
D	8.80	9.00	9.20
D1	6.90	7.00	7.10
E	8.80	9.00	9.20
E1	6.90	7.00	7.10
eB	8.10	-	8.25
e	-	0.5	-
L	0.40	-	0.75
L1	-	1.00	-
θ	0	-	7°

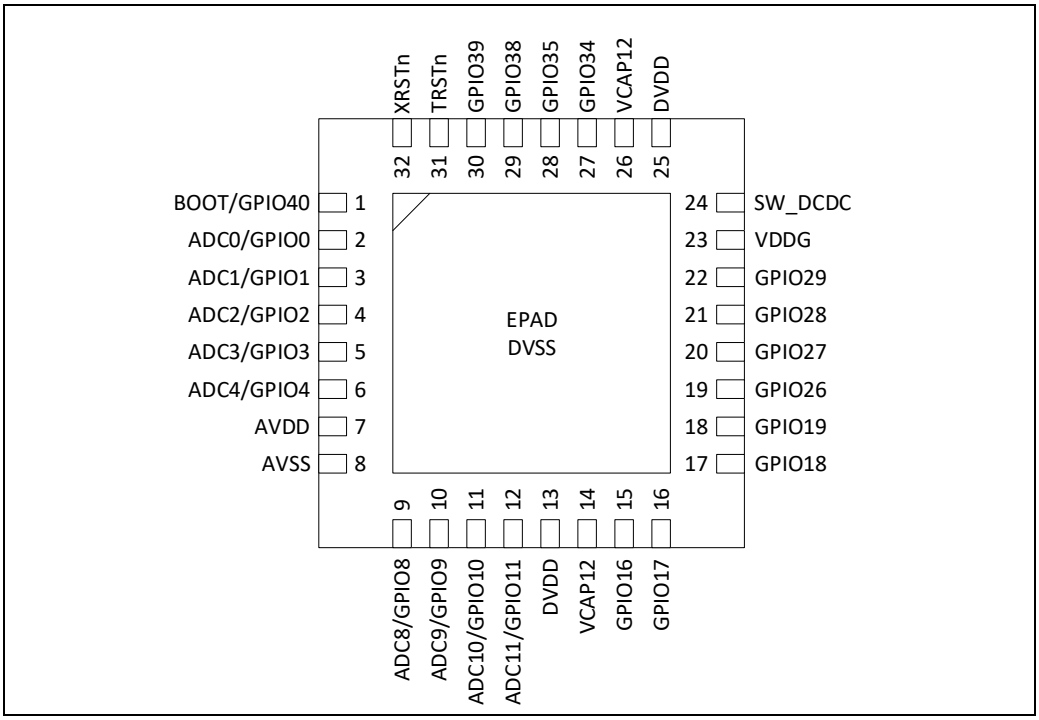
Appendix C. SPC1158 pinouts and package information

Figure 23. SPC1158 LQFP48 pinout



(1) The above figure shows the package top view.

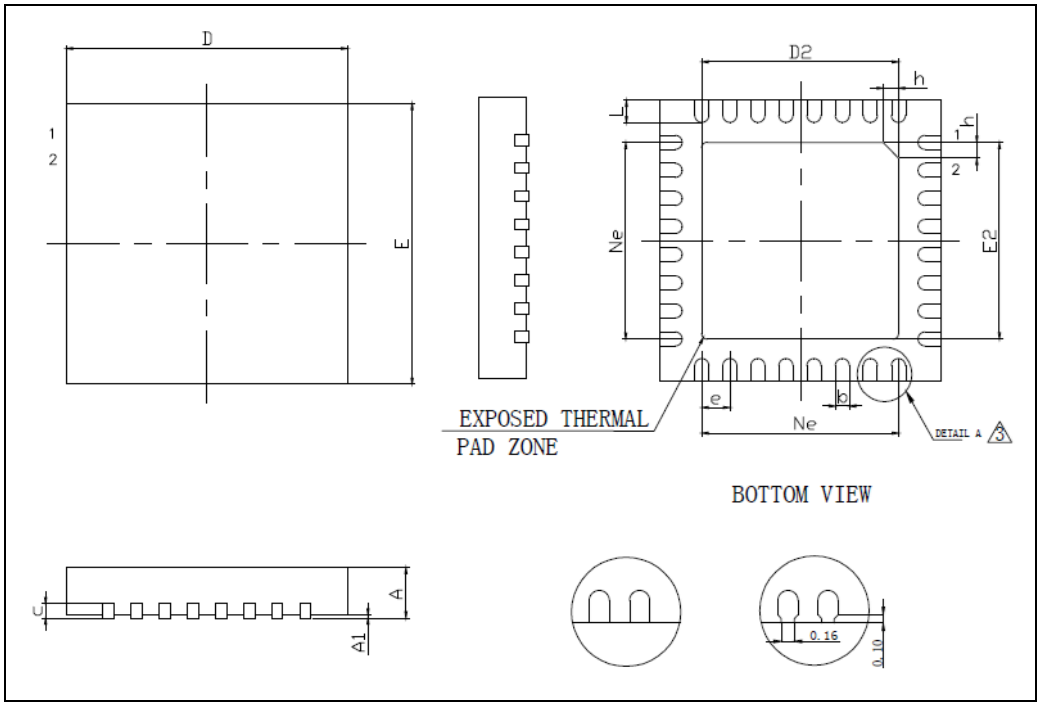
Figure 24. SPC1158 QFN32 pinout



(1) The above figure shows the package top view.

Please see Figure 22 and Table 12 for SPC1158 LQFP48 package information. SPC1158 QFN32 package information is listed in Figure 25 and Table 13.

Figure 25. QFN32 - 32 pin, 5 x 5 mm quad flat no-lead package outline



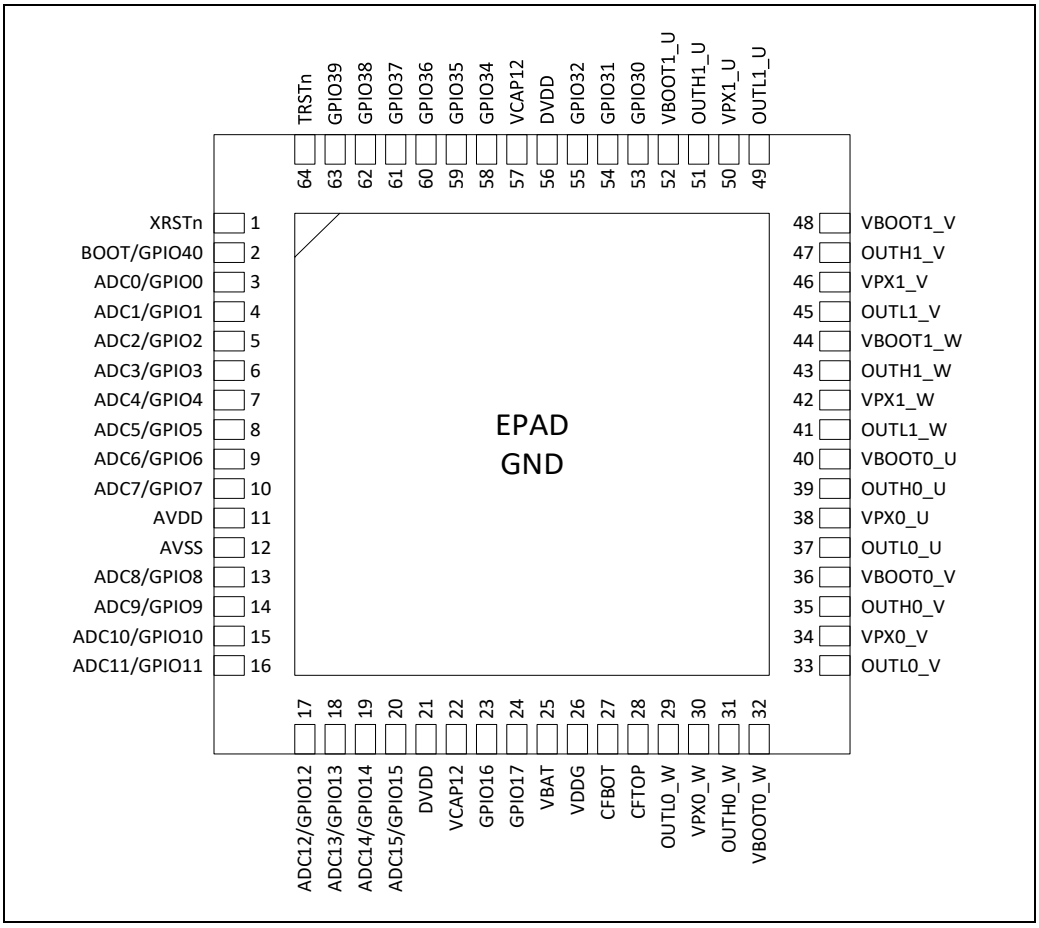
(1) Drawing is not to scale.

Table 13. QFN32 - 32 pin, 5 x 5 mm quad flat no-lead package mechanical data

Symbol	Millimeters (mm)		
	Min	Typ	Max
A	0.70	0.75	0.80
A1	-	0.02	0.05
b	0.18	0.25	0.30
c	0.18	0.20	0.25
D	4.90	5.00	5.10
D2	3.40	3.50	3.60
e	-	0.50	-
Ne	-	3.50	-
E	4.90	5.00	5.10
E2	3.40	3.50	3.60
L	0.35	0.40	0.45
h	0.30	0.35	0.40

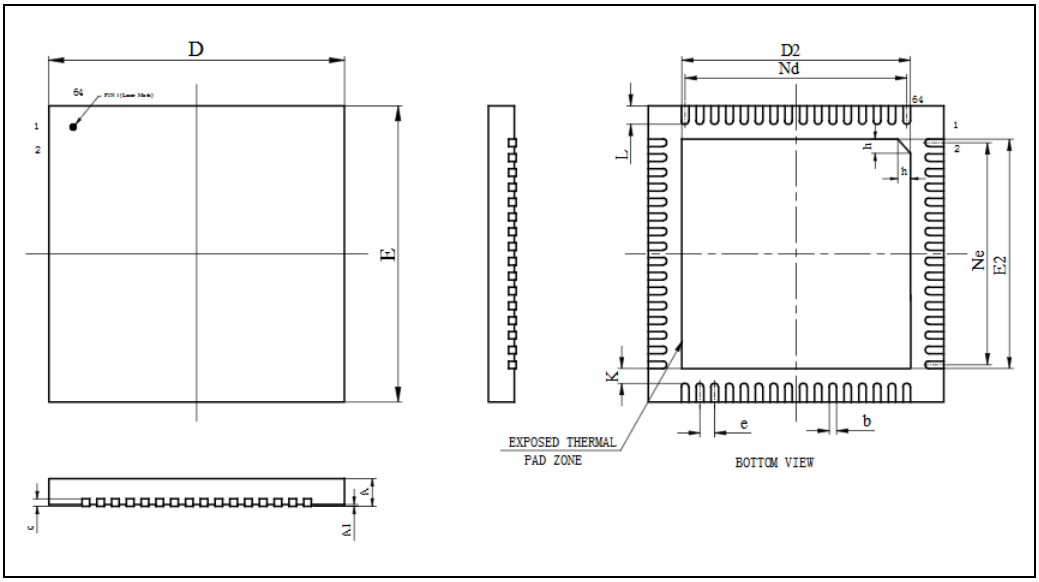
Appendix D. SPD1178 pinouts and package information

Figure 26. SPD1178 QFN64 pinout



(1) The above figure shows the package top view.

Figure 27. QFN64 – 64 pin, 8 x 8 mm quad flat no-lead package outline



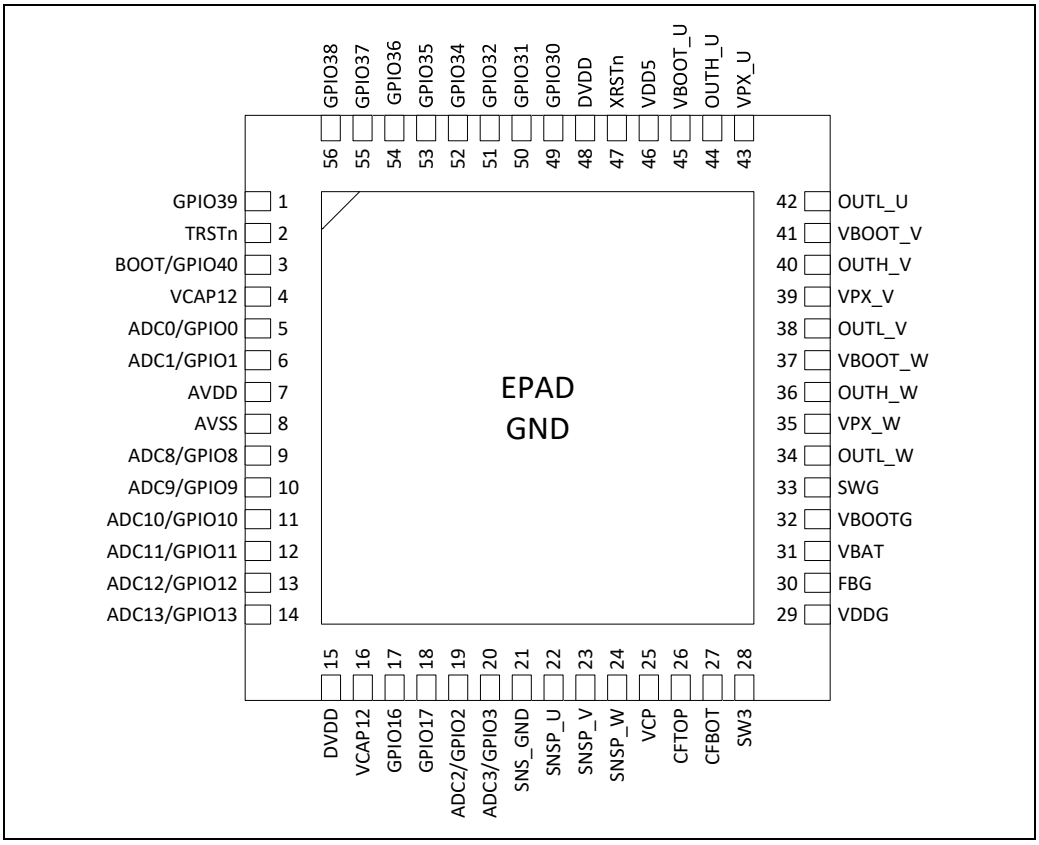
(1) Drawing is not to scale.

Table 14. QFN64 - 64 pin, 8 x 8 mm quad flat no-lead package mechanical data

Symbol	Millimeters (mm)		
	Min	Typ	Max
A	0.70	0.75	0.80
A1	-	0.02	0.05
b	0.15	0.20	0.25
c	0.18	0.20	0.25
D	7.90	8.00	8.10
D2	6.10	6.20	6.30
e	-	0.40	-
Nd	-	6.00	-
E	7.90	8.00	8.10
E2	6.10	6.20	6.30
Ne	-	6.00	-
L	0.45	0.50	0.55
K	0.20	-	-
h	0.30	0.35	0.40

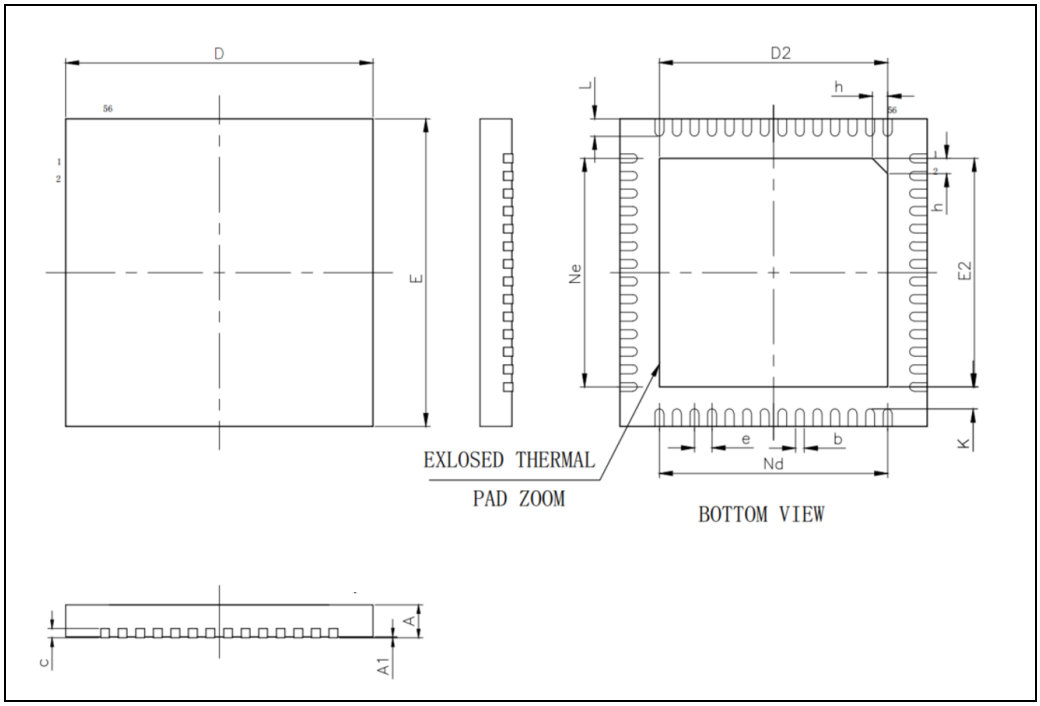
Appendix E. SPD1188 pinouts and package information

Figure 28. SPD1188 QFN56L pinout



(1) The above figure shows the package top view.

Figure 29. QFN56L – 56 pin, 7 x 7 mm quad flat no-lead package outline



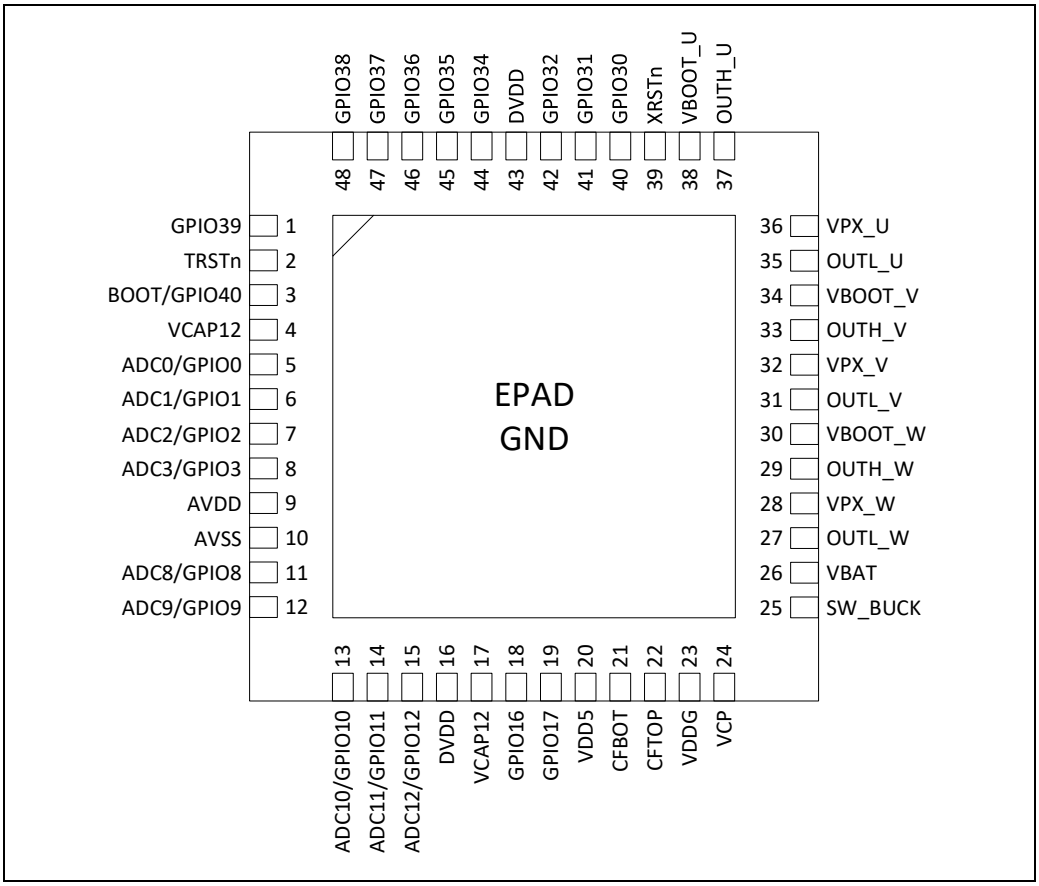
(1) Drawing is not to scale.

Table 15. QFN56L - 56 pin, 7 x 7 mm quad flat no-lead package mechanical data

Symbol	Millimeter (mm)		
	Min	Typ	Max
A	0.70	0.75	0.80
A1	-	0.02	0.05
b	0.15	0.20	0.25
c	0.18	0.2	0.25
D	6.90	7.00	7.10
D2	5.10	5.20	5.30
e	0.40		
Ne	5.20		
Nd	5.20		
E	6.90	7.00	7.10
E2	5.10	5.20	5.30
L	0.35	0.40	0.45
h	0.30	0.35	0.40

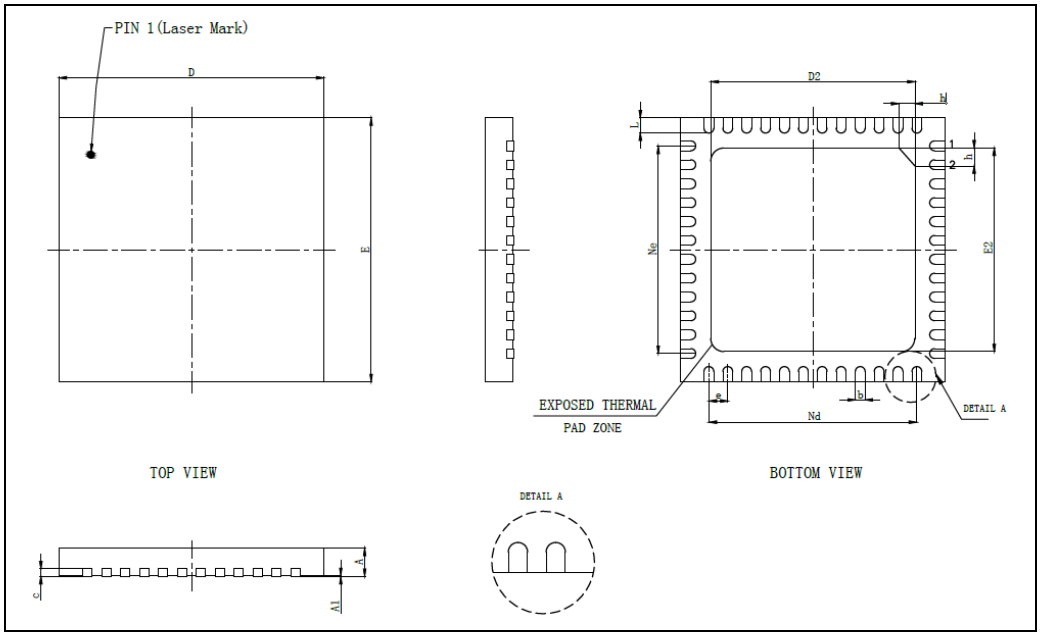
Appendix F. SPD1148 pinouts and package information

Figure 30. SPD1148 QFN48L pinout



(1) The above figure shows the package top view.

Figure 31. QFN48L – 48 pin, 7mm x 7mm quad flat no-lead package outline



(1) Drawing is not to scale.

Table 16. QFN48L – 48 pin, 7mm x 7mm quad flat no-lead package mechanical data

Symbol	Millimeter		
	Min	Nom	Max
A	0.70	0.75	0.80
A1	0	0.02	0.05
b	0.18	0.25	0.30
c	0.18	0.20	0.23
D	6.90	7.00	7.1
D2	5.30	5.40	5.50
e	0.50		
Ne	5.50		
Nd	5.50		
E	6.90	7.00	7.10
E2	5.30	5.40	5.50
L	0.35	0.40	0.45
h	0.30	0.35	0.40

5 Revision history

Table 17. Document revision history

Date	Revision	Changes
5-May-2019	1	Initial release.
23-April-2020	2	1. Update Figure 8, add Step 11. 2. Update Table 5, add command S_CMD_SET_RDN. 3. Add Chapter 4.13 for Step 11.
21-May-2020	3	1. Add Figure 7. 2. Add Appendix F for SPD1148.