SSPI Embedded Programming Demo Using Raspberry Pi User Guide

Reference Design

FPGA-RD-02295-1.1

February 2025

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUNDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Inclusive Language

This document was created consistent with Lattice Semiconductor’s inclusive language policy. In some cases, the language in underlying tools and other items may not yet have been updated. Please refer to Lattice’s inclusive language FAQ 6878 for a cross reference of terms. Note in some cases such as register names and state names it has been necessary to continue to utilize older terminology for compatibility.

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
CS	Chip Select
EPROM	Erasable Programmable Read-Only Memory
FTDI	Future Technology Devices International
GPIO	General Purpose Input Output
IOCTL	Input Output Control
MISO	Controller In Target Out
MOSI	Controller Out Target In
RBPI	Raspberry Pi
Rx	Receiver
SPI	Serial Peripheral Interface
SSPI	Target Serial Peripheral Interface
Tx	Transmitter

1. Introduction

This demo interfaces the Raspberry Pi SPI driver to the SSPI embedded programming source code in the Lattice Radiant™ and Diamond™ software directories. Refer to Source Code Directory for the location of the embedded programming source code. This demo is applicable to Lattice devices including MachXO2™, MachXO3™, CrossLink™, CrossLinkPlus™, and Nexus™-based FPGA devices.

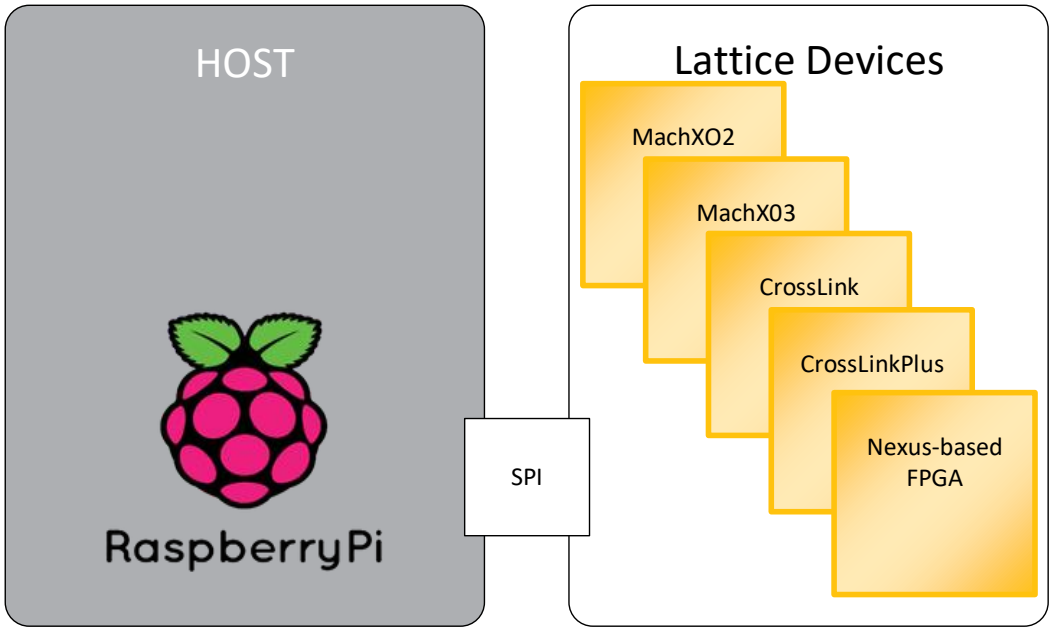

Figure 1.1. Raspberry Pi SSPI Interface with Lattice Device

The .sed and .sea files are used as the data file and algorithm file for the programming procedure, respectively. Generation of the .sed and .sea files are covered in Appendix A.

Download the reference design files from the Lattice SSPI Embedded Programming Demo Using Raspberry Pi Reference Design web page.

1.1. Embedded Programming Source Code Architecture

Table 1.1 and Table 1.2 list the existing files needed and files to be created for embedded programming, respectively. The files to be created are the compiler and driver files.

Table 1.1. Existing Files Needed for Embedded Programming

File	Needs Customization	Description
main.c	Yes	Contains the main() function
hardware.h, hardware.c	Yes	SPI hardware abstraction layer; interfaced with host driver
intrface.h, intrface.c	No	File abstraction layer; works together with util.h to integrate .sed and .sea files to core.c
SSPIEm.h, SSPIEm.c	No	Entry calls to the programming algorithm
core.h, core.c	No	Main FPGA programming algorithm
opcode.h	No	Definitions of codes in the bit files
debug.h	No	Definitions of codes printed when debug is enabled
util.h, util.c	No	Checksum utility function

Table 1.2. Files to be Created for Embedded Programming

File	Description
Makefile	Makefile that builds the executable sspiem-rbpi with the standard make utility
rbpi-spi.c, rbpi-spi.h	SPI hardware driver using Raspberry Pi on-chip SPI port

Figure 1.2 shows the **e**mbedded programming source code architecture**.** For this demo, the executable file will be run using the command terminal. SSPIEm.c handles the parsing of both the .sed and .sea files. With the help of intrface.c and util.c, the .sed and .sea files are converted to FPGA programming commands for core.c and hardware.c. hardware.c is the interface of the hardware driver to the FPGA programming commands in core.c. driver.c contains the Raspberry Pi driver functions to be interfaced with hardware.c.

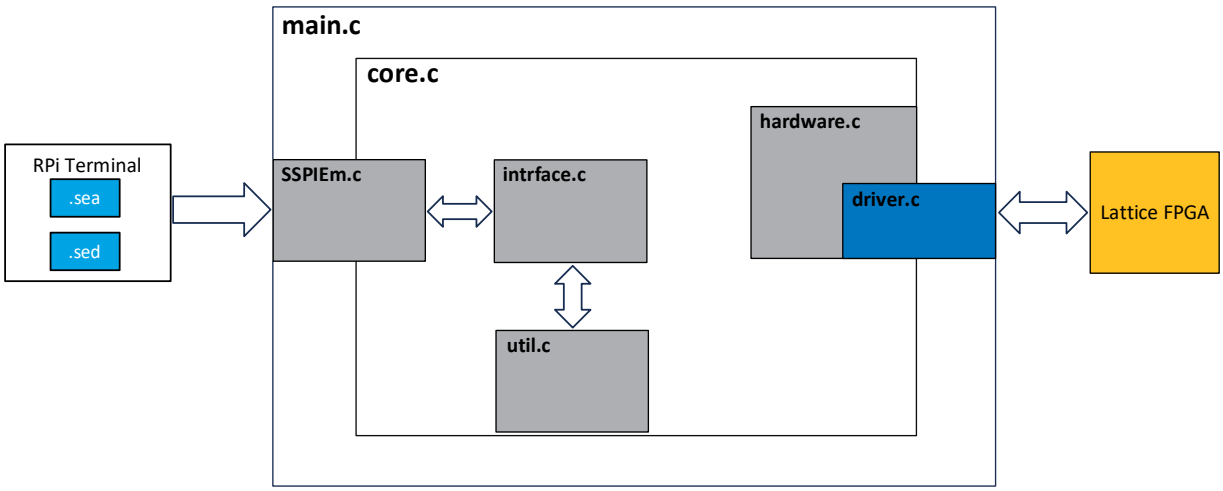

Figure 1.2. Embedded Programming Source Code Architecture

Only two existing source files (main.c and hardware.c) will be modified as indicated in Table 1.1. Additionally, one source file (driver.c or rbpi-spi.c) will be created. Refer to Modifying the Source Code and Writing the Driver for more information. Refer to Compiling and Running the Demo for the Makefile.

There are two types of embedded source codes:

- **File-based source code** uses file-based data storage, such as on Linux systems. This means that data and algorithms are stored in .sed and .sea files, which can be read and written during runtime. This approach makes modifying the source code easier since you do not need to recompile the code; you only need to update the .sed and .sea files during runtime.
- **EPROM-based source code** uses erasable programmable read-only memory (EPROM) for storing data and algorithms, which are compiled into the system. In this case, .sed and .sea files are converted into .c arrays and added to the program during compilation. This method is particularly useful for microcontroller devices but it also means you must recompile the code every time you update the .c arrays containing the .sed and .sea data.

1.2. Source Code Directory

Figure 1.3 shows the location of the embedded programming installation directory for the Radiant Programmer and Diamond Programmer. This demo focuses on the sspiembedded source code. Figure 1.4 shows the list of files in the embedded programming source code directory for the Radiant Programmer and Diamond Programmer.

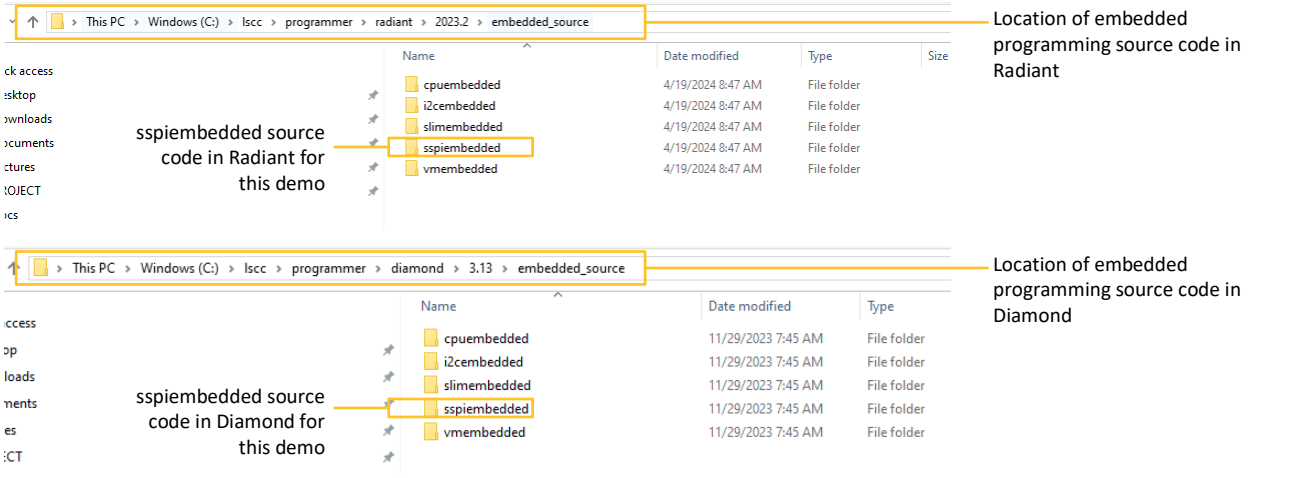

Figure 1.3. Embedded Programming Source Code Directories

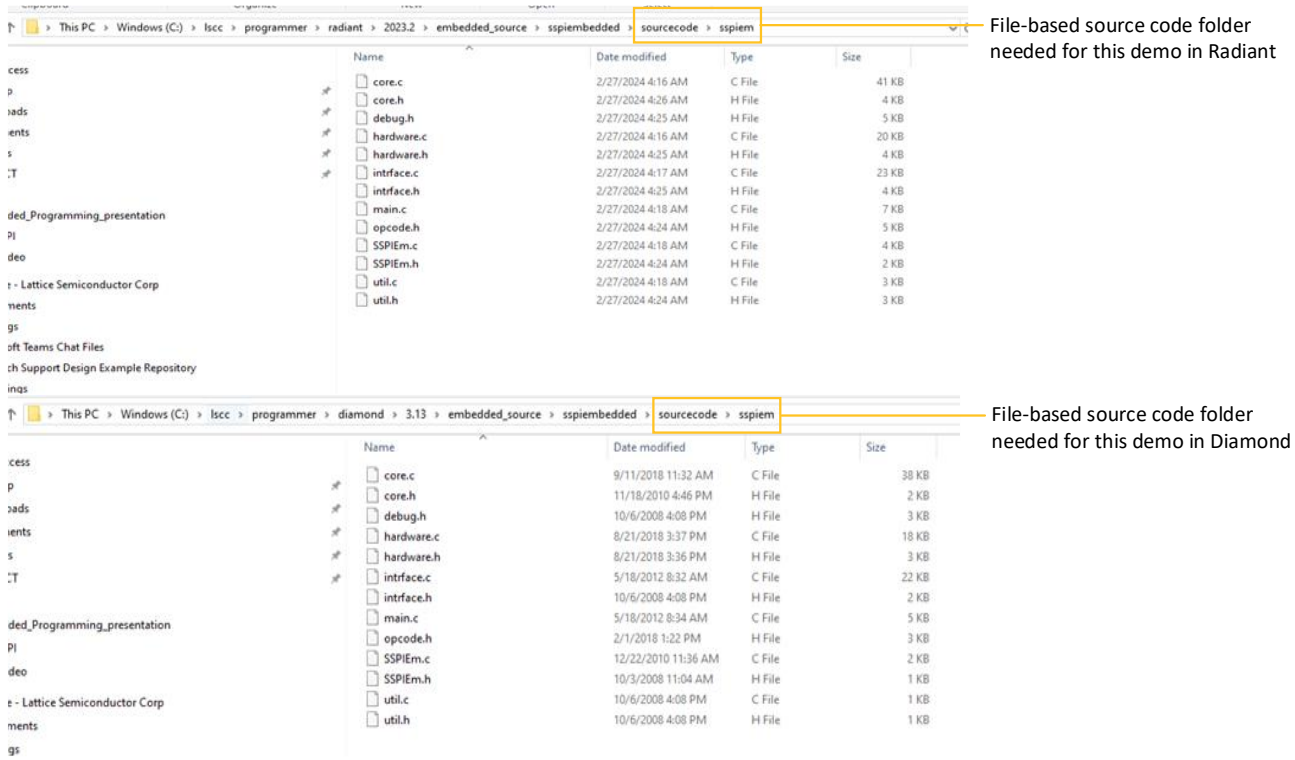

Figure 1.4. Embedded Programming Source Code Lists for File-Based Source Code

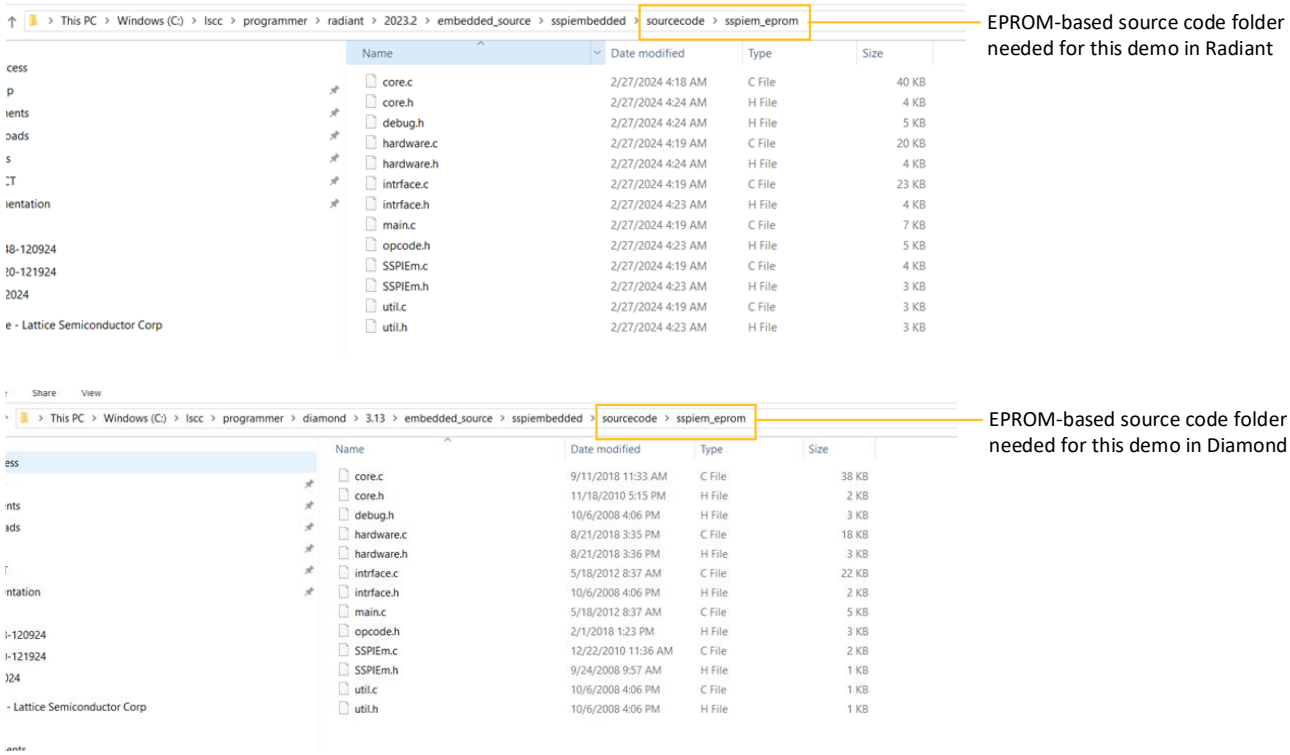

Figure 1.5. Embedded Programming Source Code Lists for EPROM-Based Source Code

2. Modifying the Source Code and Writing the Driver

This section discusses the modification of the source code files main.c and hardware.c. This section also discusses the creation of the driver file and how it is interfaced with hardware.c.

2.1. Updating main.c for File-Based Source Code

The main.c file contains code for initializing the driver and calling FPGA programming commands. This section discusses the modifications required in main.c for file-based source code. Before modifying main.c, ensure that the appropriate header files and libraries are included. The following header files and libraries are required in main.c:

```
#include <unistd.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include "rbpi-spi.h"
#include "debug.h"
#include "hardware.h"
#include <stdio.h>
#include "SSPIEm.h"
```

2.1.1. Adding an Instruction Function

An instruction function prints instructions to the user on using the demo. These instructions appear when the arguments of the executable file are incorrect or incomplete. The following shows the print_usage() function added into main.c:

```
void print_usage()
{
    fprintf(stderr, "sspiem-rbpi usage:\n");
    fprintf(stderr, "  form 1: sspiem-rbpi -help\n");
    fprintf(stderr, "          - This help.\n");
    fprintf(stderr, "  form 2: sspiem-rbpi <spi clock speed> <sea_file>
<sed_file> <-debug_command>\n");
    fprintf(stderr, "          - Program FPGA:\n");
    fprintf(stderr, "            <spi clock speed> expressed in KHz, range 1 to
20000> Consult the guide for details.\n");
    fprintf(stderr, "            <sea_file> is the algorithm file produced by
Lattice Radiant or Diamond tool.\n");
    fprintf(stderr, "            <sed_file> is the data file produced by Lattice
Radiant or Diamond tool.\n");
    fprintf(stderr, "            <-debug_commands> -debug_on:turn on debug
printing, -debug_off:turn off debug printing\n");
    fprintf(stderr, "            \n");
    fprintf(stderr, "Example: sspiem-rbpi 5000 algo.sea data.sed -debug_on\n");
}
```

2.1.2. Updating main() Function

The following shows the modifications in the main() function:

```
int main(int argc, char *argv[])
{
    int siRetCode = 0;
    int speed;

    printf("Lattice Semiconductor Corp.\n");
    printf("SSPI Embedded(tm) V%s 2023\n", VME_VERSION_NUMBER);

    if (argc == 5) {
        speed = atoi(argv[1]);
        if (!strcmp(argv[4],"-debug_on")){
            a_uiDebug=1;
        }
        else if(!strcmp(argv[4],"-debug_off")){
            a_uiDebug=0;
        }
        else{
            print_usage();
            exit(0);
        }

        if (rbpi_init(speed * 1024)) {
            rbpi_assert_cs(1);
            rbpi_assert_cs(0);
            siRetCode = SSPIEm_preset(argv[2], argv[3]);
            siRetCode = SSPIEm(0xFFFFFFFF);

            printf("\n\n");
            if ( siRetCode != 2 ) {
                printf( "+=======+\n" );
                printf( "| FAIL! |\n" );
                printf( "+=======+\n" );
                printError(siRetCode);
            }
            else {
                printf( "+=======+\n" );
                printf( "| PASS! |\n" );
                printf( "+=======+\n" );
            }
        }
        usleep(100*1000);
        rbpi_exit();
    } else
        print_usage();

    return siRetCode;
}
```

Enable debug printing.

Initialize SSPI driver.

Pass along .sed and .sea files and start of programming algorithm.

The variable a_uiDebug is used to enable and disable debug printing. See TRANS_transceive_stream() Implementation for other modifications needed for debug printing. The functions rbpi_init and rbpi_exit initializes and closes the Raspberry Pi SPI drivers, respectively. Refer to Creating rbpi-spi.c for more details on the driver file (rbpi-spi.c). Regardless of platform, it is advisable to initialize the driver in the main() function of main.c for easier access and control.

2.2. Updating main.c for EPROM-Based Source Code

The main.c file contains code for initializing the driver and calling FPGA programming commands. This section discusses the modifications required in main.c for EPROM-based source code. Before modifying main.c, ensure that the appropriate header files and libraries are included. The following header files and libraries are required in main.c:

```
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include "SSPIEm.h"
#include "rbpi-spi.h"
#include "debug.h"
#include "hardware.h"
#include "test_impl_1_data.h"
#include "test_impl_1_algo.h"
```

Add the .h files for the converted .c arrays for .sed and .sea files.

2.2.1. Updating main() Function

The following shows the modifications done in the main() function:

```
int main( )
{
    int siRetCode = 0;
    char Message[512];
    unsigned char *setAlgoPtr;
    unsigned int setAlgoSize  = g_iAlgoSize;
    unsigned char *setDataPtr;
    unsigned int setDataSize  = g_iDataSize ;

    setAlgoPtr = g_pucAlgoArray;
    setDataPtr = g_pucDataArray;

    print_out_string( "                Lattice Semiconductor Corp.\n" );
    sprintf(Message,"\n            SSPI Embedded(tm) V%s
2012\n",VME_VERSION_NUMBER);
    print_out_string( "                Ported by Rhodz\n" );
    //print_out_string( Message );


    rbpi_init(3000* 1024) ;
    a_uiDebug=1;
    rbpi_assert_cs(1);
```

This part sets the values of set Algosize and set DataSize and sends the pointer array of the algorithm and data to setAlgoPtr and setDataPtr respectively.

The goal is to not have any external settings after compilation so set the frequency and printing debug manually before compilation.

```
    rbpi_assert_cs(0);

    siRetCode = SSPIEm_preset(setAlgoPtr, setAlgoSize, setDataPtr, setDataSize);
    siRetCode = SSPIEm(0xFFFFFFFF);
    if ( siRetCode != 2 ) {

        printf ("\n\n");
        printf( "+=======+\n" );
        printf( "| FAIL! |\n" );
        printf( "+=======+\n\n" );
        printError(siRetCode);

    }
    else {
        printf( "+=======+\n" );
        printf( "| PASS! |\n" );
        printf( "+=======+\n\n" );
    }
    return siRetCode;
}
```

SSPIEm_preset() sends the algorithm and data array and their respective sizes. SSPIEm() starts the programming operation.

2.3. Updating hardware.c

The only areas requiring modification in hardware.c are some functions needed for interfacing with the Raspberry Pi drivers. Table 1.1 lists the functions to be modified in hardware.c. Modification involves adding code into the functions (pre-existing functions provide only usage guidelines and contain no code). hardware.c is the same for file-based and EPROM-based systems so both source codes must have the same modifications.

Table 2.1. Functions to be Modified in hardware.c

Function	Description
wait()	Delay function in between FPGA programming commands
TRANS_transmitBytes()	Write data in buffer to FPGA
TRANS_receiveBytes()	Read data from FPGA to buffer
TRANS_startranx()	Assert chip-select signal (CS)
TRANS_endtranx()	De-assert chip-select signal (CS)
TRANS_cstoggle()	Toggle chip-select signal (CS/TMS)
TRANS_runClk()	Pulse SPI clock signal eight times or more; used to send dummy bytes

These functions will be interfaced to the Raspberry Pi SPI drivers. This means that when these functions are called by the FPGA programming algorithm, the driver functions should correspondingly be called to implement the SPI communication in the Raspberry Pi SPI port. The following header files and libraries are required in hardware.c:

```
#include <unistd.h>
#include <stdint.h>
#include <stdio.h>
#include <assert.h>
#include "rbpi-spi.h"
#include "intrface.h"
#include "opcode.h"
#include "debug.h"
#include "hardware.h"
```

2.3.1. wait() Implementation

The wait() function is used by the FPGA programming algorithm to implement delay after programming commands. The following are two recommended methods of implementing the wait() function:

- Using the usleep() function.
- Using a counter.

Both methods may be used for the successful implementation of embedded programming.

The first method uses the usleep() function. The usleep() function is an internal Linux command for delay implementation. Experimentation with Raspberry Pi shows that usleep(0) translates to a delay of at least 65 µs which is unnecessarily long. For better control of this delay, the second method can serve as an alternative. The following code snippet is a sample implementation with the first method:

```
int wait(int a_msTimeDelay)
{
	__useconds_t t = a_msTimeDelay * 1000;

	usleep(t);

	return 1;
}
```

The second method uses a counter in the form of a for loop to control the delay. A multiplier to a_msTimeDelay is added to control the length of the delay. A delay time that is too small may cause failure in programming, especially when verification is required. Finding the optimal delay time is necessary for correct implementation of embedded programming. It is the user's responsibility to find the optimal delay time. The following code snippet is a sample implementation with the second method:

```
int wait(int a_msTimeDelay)
{
	int count_ms=0;

	for(count_ms;count_ms<a_msTimeDelay*100000;count_ms++){

	}

	return 1;
}
```

2.3.2. TRANS_transmitBytes() and TRANS_receiveBytes() Implementation

TRANS_transmitBytes() is responsible for transmitting data. The arguments are the pointer buffer containing the data to be sent and the expected number of bits to be sent. This function is called by the FPGA programming algorithm when transmitting data through the SPI bus. In this demo, the function rbpi_tx() is called every time TRANS_transmitBytes() is called. The function rbpi_tx() is the driver function that handles control of the SPI bus for data transmission. This function is discussed in rbpi_tx() and rbpi_rx() Implementation. The following code snippet shows the TRANS_transmitBytes() function:

```
int TRANS_transmitBytes(unsigned char *trBuffer, int trCount)
{
      return rbpi_tx((uint8_t*)trBuffer, (int) trCount);
}
```

TRANS_receiveBytes() is responsible for receiving data. The arguments are the empty pointer buffer which is to be filled by the data to be received and the expected number of bits to be received. This function is called by the FPGA programming algorithm when receiving data from the SPI bus. In this demo, the function rbpi_rx() is called every time TRANS_receiveBytes() is called. The function rbpi_rx() is the driver function that handles control of the SPI bus for data reception. This function is discussed in rbpi_tx() and rbpi_rx() Implementation. The following code snippet shows the TRANS_receiveByes() function:

```
int TRANS_receiveBytes(unsigned char *rcBuffer, int rcCount)
{
      return rbpi_rx((uint8_t*)rcBuffer, (int) rcCount);
}
```

2.3.3. TRANS_starttranx() and TRANS_endtranx() Implementation

TRANS_starttranx() initiates the start of SPI bus communication. For SPI transaction, the communication starts by pulling chip select (CS) low. The function rbpi_assert_cs() is the driver function used for CS control. The following code snippet shows the TRANS_starttranx() function:

```
int TRANS_starttranx(unsigned char channel)
{
	assert(channel == 0);

	if(channel != 0){
		//wait(10);
		return 0;}
	else{
		rbpi_assert_cs(1);
		//wait(10);
	}
	if (a_uiDebug)
		printf("Start of transmission.\n");
	return 1;
}
```

TRANS_endtranx() terminates the communication in the SPI bus. For SPI transaction, the communication ends by pulling CS high. The function rbpi_assert_cs () is again used for CS control. The following code snippet shows the TRANS_endtranx() function:

```
int TRANS_endtranx()
{
	//wait(10);
	if (a_uiDebug)
		printf("End of transmission.\n");
	return rbpi_assert_cs(0);
}
```

Note: The dedicated SPI CS signal is not used as the Lattice embedded programming source code requires manual control of the CS pin. Refer to rbpi_assert_cs() Implementation for more information.

2.3.4. TRANS_cstoggle() and TRANS_runClk() Implementation

TRANS_cstoggle() toggles the CS pin. It uses rbpi_assert_cs() to control the CS signal, which is also the function used by TRANS_starttranx() and TRANS_endtranx(). The following code snippet shows the TRANS_cstoggle() function:

```
int TRANS_cstoggle(unsigned char channel)
{
	assert(channel == 0);

	if(channel != 0)
		return 0;
	else {
		rbpi_assert_cs(0);
		rbpi_assert_cs(1);

		rbpi_assert_cs(0);
	}
	if (a_uiDebug)
		printf("Toggle Chip Select./n");
	return 1;
}
```

TRANS_runClk() drives extra clocks. It uses the rbpi_tx command but only sends the 0xFF command. The following code snippet shows the TRANS_runClk() function:

```
int TRANS_runClk(int clk)
{
	uint8_t dummy = 0xff;

	return rbpi_tx(&dummy, 8);
}
```

2.3.5. TRANS_transceive_stream() Implementation

TRANS_transceive_stream() connects hardware.c to core.c. The default source code does not contain the code for debug printing. For this demo, this function is updated to enable debug printing. Note that when using other embedded systems, the printf command might not work. Therefore, debug printing only works for this demo. For embedded systems other than Raspberry Pi, further development on debug printing is needed. All functions that print the send and receive bytes are commented with “Debug printing”.

```
int TRANS_transceive_stream(int trCount, unsigned char *trBuffer,
int trCount2, int flag, unsigned char *trBuffer2, int mask_flag, unsigned char
*maskBuffer)
{
      int i                          = 0;
      unsigned short int tranxByte   = 0;
      unsigned char trByte           = 0;
      unsigned char dataByte         = 0;
      int mismatch                   = 0;
      unsigned char dataID           = 0;

      if(trCount > 0)
      {
            /* calculate # of bytes being transmitted */
            tranxByte = (unsigned short) (trCount / 8);
            if(trCount % 8 != 0){
                  tranxByte ++;
                  trCount += (8 - (trCount % 8));
            }
            if (a_uiDebug){ //Debug Printing
            for(i=0;i<tranxByte;i++){
                  printf("transmit 1 byte of data 0x%02x.\n",trBuffer[i]);
            }}
            if( !TRANS_transmitBytes(trBuffer, trCount) )
                  return ERROR_PROC_HARDWARE;
      }
```

“Debug printing” comment

Sample debug printing code

For the complete code with debug printing added, contact Lattice. The variable a_uiDebug is set during runtime from main.c.

2.4. Creating rbpi-spi.c

rbpi-spi.c contains the functions interfaced with hardware.c. It connects the source code to the hardware drivers. It also contains the functions to initialize and close the SPI bus. Table 2.2 lists the functions contained in rbpi-spi.c.

Table 2.2. Functions to be Created in rbpi-spi.c

Function	Description
rbpi_init()	Initializes the SPI bus
rbpi_ioctl()	Main driver controller used by rbpi_tx() and rbpi_rx() for sending and receiving data
rbpi_tx()	Function used for sending data
rbpi_rx()	Function used for receiving data
rbpi_assert_cs()	Asserts and de-asserts CS pin
rbpi_exit()	Closes the SPI bus

ioctl is the main driver function used for this demo. ioctl (input and output control) is used to communicate with device drivers. This demo uses the ioctl capability to control drivers driving SPI dedicated pins. The following header files and libraries are required in rbpi-spi.c:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include <fcntl.h>
#include <errno.h>
#include <linux/spi/spidev.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include "rbpi-spi.h"
#include "hardware.h"
```

2.4.1. rbpi_init() Implementation

The function rbpi_init() initializes the settings of the SPI bus. The following code snippet shows a sample initialization of the SPI bus:

```
int rbpi_init(uint32_t spi_speed)
{
	int ret = 0;
	uint8_t mode = SPI_MODE_0;
	uint8_t bits = 8;
	uint32_t reg;
	uint32_t shift;

	// setup spi via ioctl

	speed = spi_speed;

	spi_fd = open("/dev/spidev0.0", O_RDWR);
	if (spi_fd < 0) {
		fprintf(stderr, "Failed to open /dev/spidev0.0: %s\n",
strerror(errno));
		return 0;
	}

	ret |= ioctl(spi_fd, SPI_IOC_WR_MODE, &mode);
	ret |= ioctl(spi_fd, SPI_IOC_RD_MODE, &mode);
	ret |= ioctl(spi_fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
	ret |= ioctl(spi_fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
	ret |= ioctl(spi_fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
	ret |= ioctl(spi_fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
```

SPI Mode and bit size per word

SPI Speed assignment

Initialization of the SPI bus

- SPI_IOC_WR_MODE and SPI_IOC_RD_MODE set the mode of the SPI transaction. This demo uses SPI Mode 0.
- SPI_IOC_WR_BITS_PER_WORD and SPI_IOC_RD_BITS_PER_WORD set the number of bits in each SPI transfer word. This demo uses 8 bits per word.
- SPI_IOC_WR_MAX_SPEED_HZ and SPI_IOC_RD_MAX_SPEED_HZ assign the maximum transfer speed in Hz.

These ioctl requests are used to initialize the SPI bus. For more information regarding these requests, see https://www.kernel.org/doc/Documentation/spi/spidev.

2.4.2. rbpi_ioctl() Implementation

The function rbpi_ioctl() is the main function that controls the ioctl driver and implements the communication on the SPI bus.

```
static int rbpi_ioctl(unsigned char *tx_buf, unsigned char *rx_buf, int len)
{
	struct spi_ioc_transfer req;

	memset(&req, 0, sizeof(req));

	req.tx_buf = (uintptr_t) tx_buf;
	req.rx_buf = (uintptr_t) rx_buf;
	req.len = len;

	return ioctl(spi_fd, SPI_IOC_MESSAGE(1), &req) == -1;
}
```

SPI Channel

Start SPI transaction request

Struct for SPI transfer

- struct spi_ioc_transfer is the struct used by the ioctl() function call for describing the SPI transaction.
- tx_buf, rx_buf, and len are members of struct spi_ioc_transfer() that are needed to run SPI transactions. The tx_buf and rx_buf members contain the buffer pointer for the data to be sent and the data to be received, respectively. The len member contains the number of bits to be sent or received.
- The source code has separate functions for transmission and reception namely TRANS_transmitBytes() and TRANS_receiveBytes(). This means that each ioctl call uses only either tx_buf or rx_buf.
- SPI_IOC_MESSAGE(1) is a request for start of SPI transaction. For more information regarding this command, see https://www.kernel.org/doc/Documentation/spi/spidev.

2.4.3. rbpi_tx() and rbpi_rx() Implementation

The functions rbpi_tx() and rbpi_rx() use rbpi_ioctl() to initiate the SPI transaction.

```
int rbpi_tx(uint8_t *buf, uint32_t bits)
{
      uint32_t bytes = bits/8;
      assert(bits % 8 == 0 && bits >= 8);

#ifdef PRINT_STATUS
      static long n = 0;

      if (++n % 512 == 0) {
            printf(".");
            fflush(stdout);
      }
#endif

      if(rbpi_ioctl(buf, NULL, bytes)) {
            fprintf(stderr, "SPI ioctl write failed: %s\n", strerror(errno));
            return 0;
      } else
            return 1;
}
```

Bits to bytes conversion

rbpi_ioctl function call

```
int rbpi_rx(unsigned char *buf, int bits)
{
      uint32_t bytes = bits/8;
      assert(bits % 8 == 0 && bits >= 8);

#ifdef PRINT_STATUS
      static long n = 0;

      if (++n % 512 == 0) {
            printf(".");
            fflush(stdout);
      }
#endif

      if(rbpi_ioctl(NULL, buf, bytes)) {
            fprintf(stderr, "SPI ioctl read failed\n");
            return 0;
      } else
            return 1;
}
```

Bits to bytes conversion

rbpi_ioctl function call

- The argument len of rbpi_tx() and rbpi_rx() is converted to bytes as rbpi_ioctl() uses bytes for length of data to be sent or received.
- For rbpi_tx(), the pointer buffer buf is passed along to the tx_buf argument of rbpi_ioctl() while the rx_buf argument is set to NULL. For rbpi_rx(), the pointer buffer buf is passed along to the rx_buf argument of rbpi_ioctl() while the tx_buf argument is set to NULL.

2.4.4. rbpi_assert_cs() Implementation

The function rbpi_assert_cs() pulls the CS pins high and low. Note that these CS pins are not the dedicated CS pins of Raspberry Pi. Dedicated CS pins are controlled automatically by the ioctl driver. However, the Lattice embedded programming source code requires manual control of CS. Therefore, in this demo, GPIO pins are used.

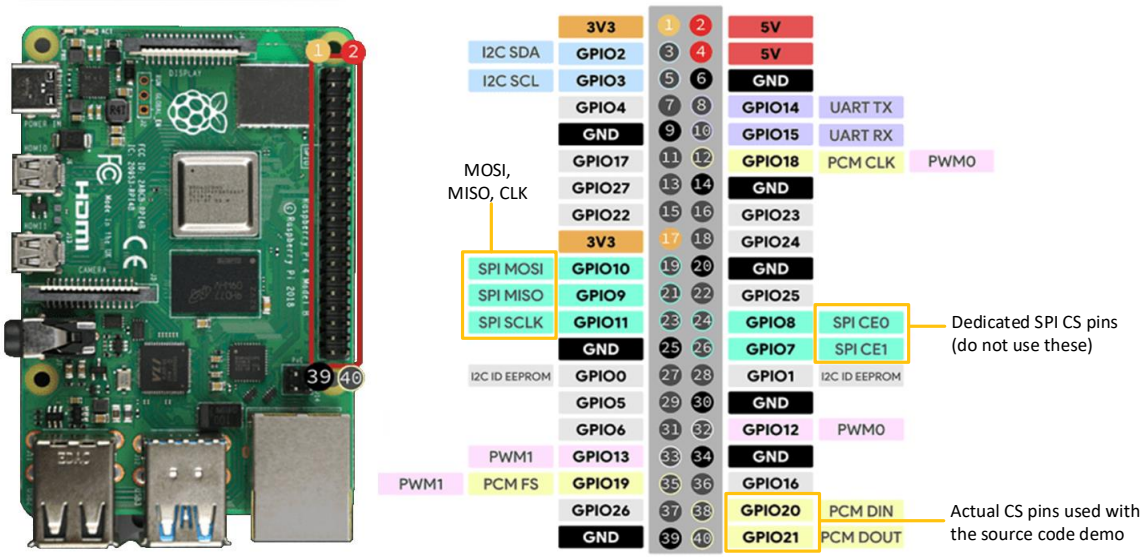

Figure 2.1. Raspberry Pi Chip Select Assignment in Hardware

The following code snippets show sample implementations of CS toggling:

```
int rbpi_assert_cs(int asserted)
{
   wait(0.05);
   *(mem + (asserted ? GPCLR0 : GPSET0) + (gpio_cs >> 5)) = (1 << (gpio_cs &
0x1F));

   count=0;

   return 1;
}
```

```
int rbpi_assert_cs(int asserted)
{
      for(count=0;count<10;count++){
   *(mem + (asserted ? GPCLR0 : GPSET0) + (gpio_cs >> 5)) = (1 << (gpio_cs &
0x1F));}

   count=0;

   return 1;
}
```

A wait() function is used before assertion or de-assertion is done. This is important because delay is needed before the assertion and de-assertion commands especially during a verify operation. Alternatively, a for loop can be used to control the delay of assertion and de-assertion.

2.4.5. rbpi_exit() implementation

The function rbpi_exit() closes the SPI bus and the gpio control.

```
void rbpi_exit()
{
      if (spi_fd > -1)
            close(spi_fd);

      if (gpio_fd > -1)
            close(gpio_fd);

      spi_fd = -1;
      gpio_fd = -1;
}
```

3. Compiling and Running the Demo

After modifying the source code and writing the driver, your files should now be ready for compiling and running.

1. To compile the demo, a Makefile will be used containing the following:

```
OBJ = main.o core.o hardware.o intrface.o SSPIEm.o util.o rbpi-spi.o

CFLAGS = -Wall

sspiem-rbpi: $(OBJ)
	cc -o sspiem-rbpi $(OBJ)

clean:
	rm -f *.o libmpsse/*.o sspiem-rbpi
```

   Ensure the Makefile and all source codes including the driver files are in the same folder.

2. Run *make sspiem-rbpi* as shown in Figure 3.1.

   When compiling for the first time, the list of objects compiled appears in the terminal. After running this command, the executable file to run the demo is generated. If there are no changes, running the command generates a prompt stating that sspiem-rbpi is up to date.

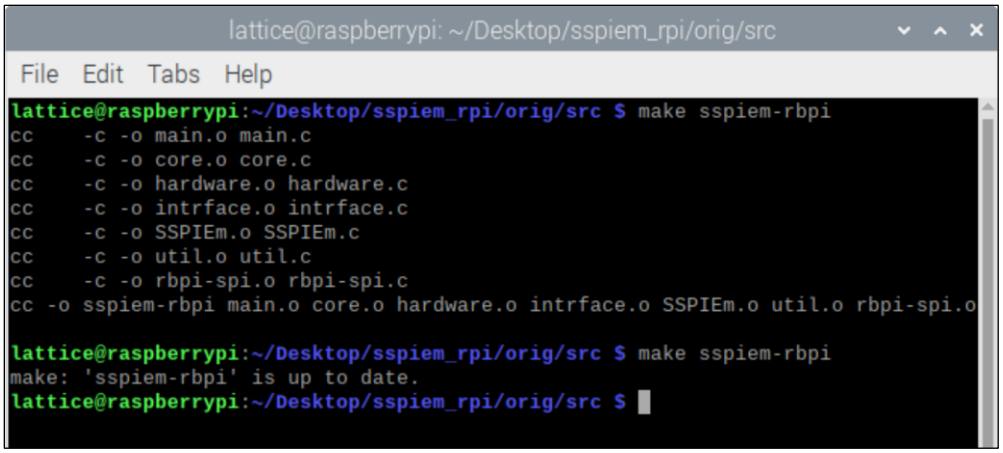

Figure 3.1. Sample Compilation

3. Run the executable file.

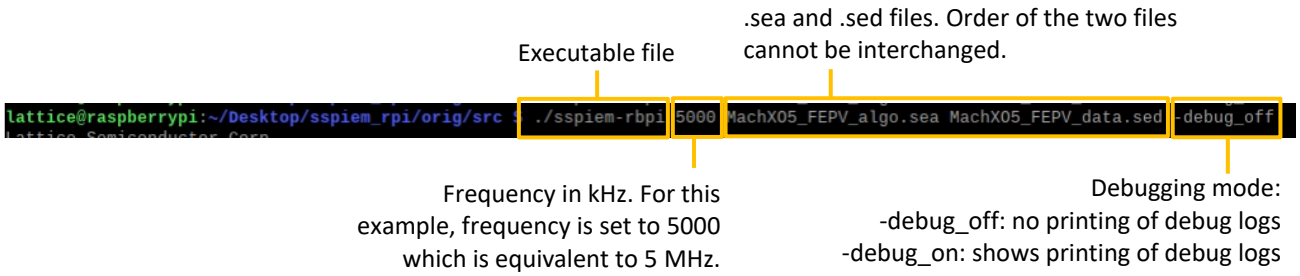

Figure 3.2. Example of Running the Executable File in File-Based Demo

- The executable file is generated after compiling the source code and the driver. It initiates and ends the SPI programming.
- The first argument of the executable file is the frequency in kHz. In this example, frequency is set to 5000 kHz or 5 MHz.
- The second and third arguments are the .sea and .sed files. These files contain the programming commands and the configuration data.
- The fourth argument turns debug printing on and off. Turning on debug printing shows the data sent and received.
- All arguments of the executable file are required to run the demo.

Figure 3.3 shows an example of running the executable file with -debug_off. With debug off, the sample run of the demo only shows a PASS result in the terminal. There are no logs for sending and receiving commands.

Figure 3.3. Sample Run without Debug Printing

Figure 3.4 shows an example of running the executable file with -debug_on. With debug_on, the sample run of the demo shows in the terminal the data transmitted and received. Debug mode should be enabled only when debugging points of failure. Otherwise, enabling debug mode adds to programming time because of the printing of logs.

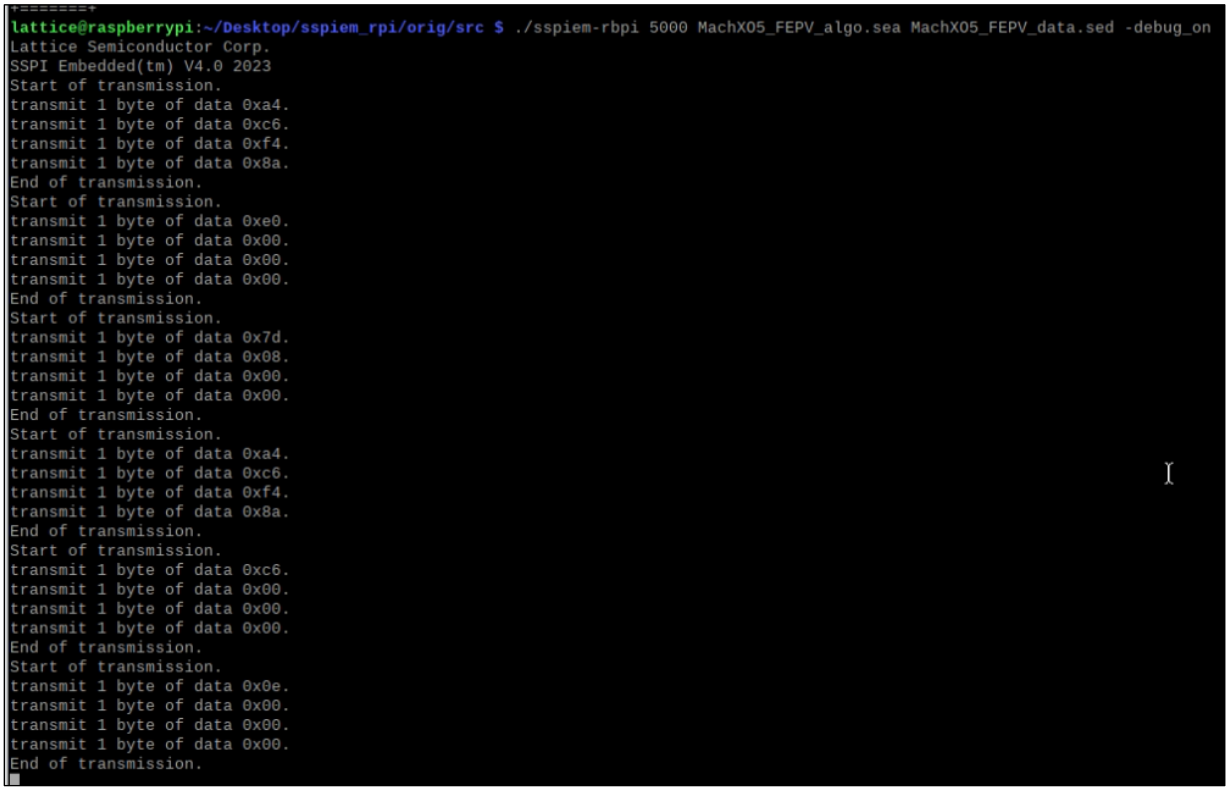

Figure 3.4. Sample Run with Debug Printing

Figure 3.5 shows an example of running the executable file in the EPROM-based demo. For the EPROM-based system, there are no arguments because the frequency, debug printing, .sea, and .sed files are set during compilation.

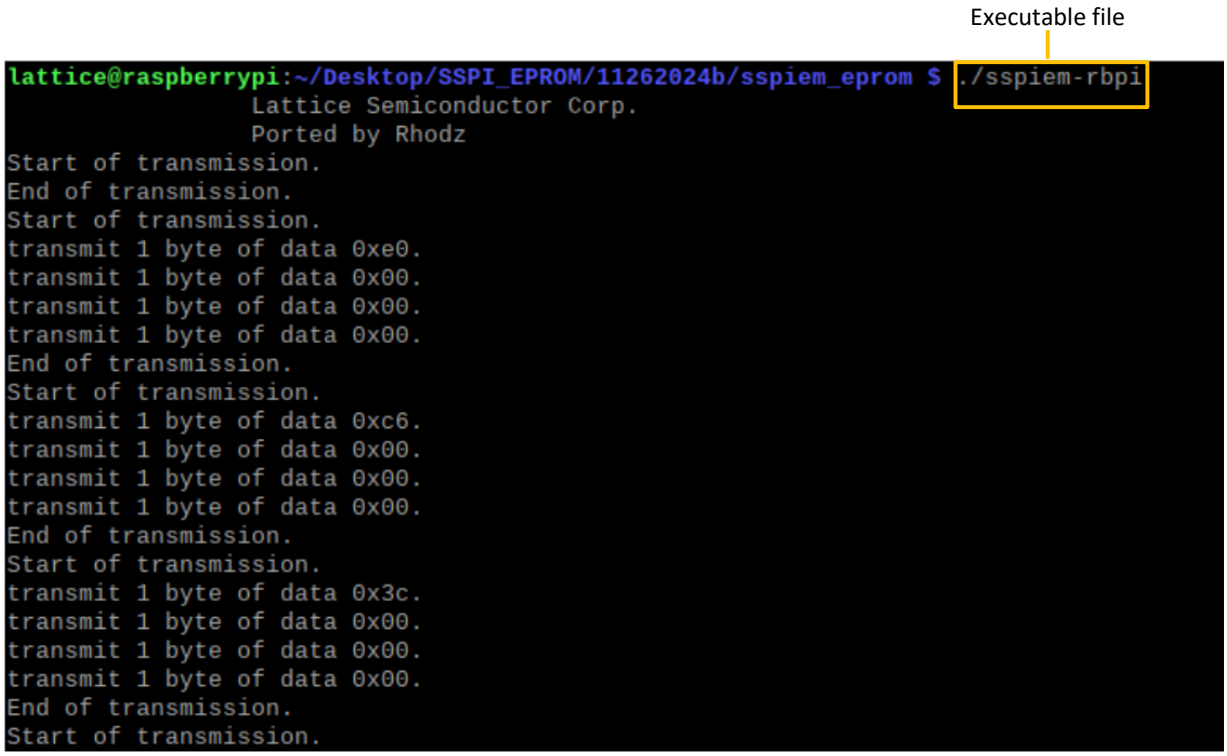

Figure 3.5. Example of Running the Executable File in EPROM-Based Demo

4. Debugging Tips

The following are tips to help with debugging:

- Ensure the function specifications for sending commands are followed. For example, with TRANS_transmitBytes and TRANS_receiveBytes, the arguments needed are the pointer buffer containing bits to be sent and the size of bits to be sent. These arguments are not optional and are used by the programming algorithm. Furthermore, the ioctl() driver used also requires these arguments.

```
int TRANS_transmitBytes(unsigned char *trBuffer, int trCount)
{
      return rbpi_tx((uint8_t*)trBuffer, (uint32_t) trCount);
}
int TRANS_receiveBytes(unsigned char *rcBuffer, int rcCount)
{
      return rbpi_rx((uint8_t*)rcBuffer, (uint32_t) rcCount);
}
```

- Ensure the CS signal is manually controlled by a separate GPIO and not the dedicated SPI CS pin.

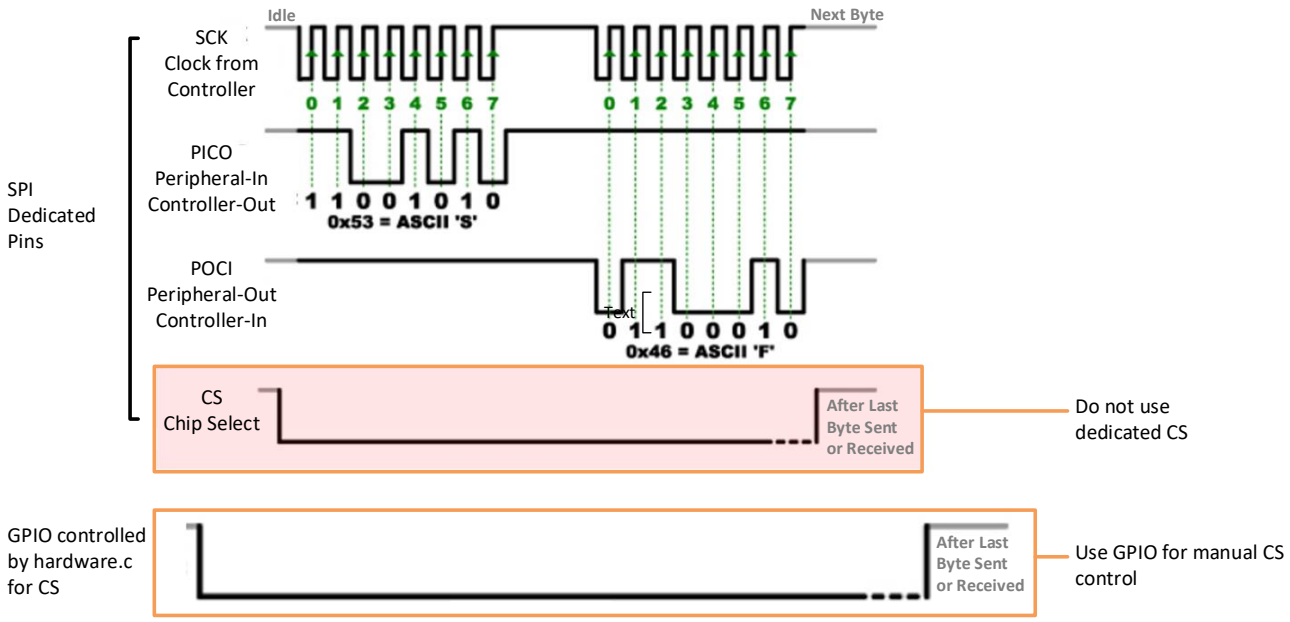

Figure 4.1. Illustration of Manual Chip Select Control

- Ensure there is enough time for verification. This is done by adding wait time before asserting or de-asserting CS. For this demo, approximately 3 µs is sufficient to finish verification. Verification time depends on the controller used. It is the user's responsibility to find the optimal delay time for the controller used. Refer to rbpi_assert_cs() Implementation for information on adding delay before assertion and de-assertion of chip select (CS).

- Check waveform captures to ensure they match how programming is performed by the Radiant or Diamond programmer.

  When debugging an embedded programming implementation, compare how the Radiant or Diamond programmer performs the programming. Figure 4.2 shows an implementation with a verified failure.

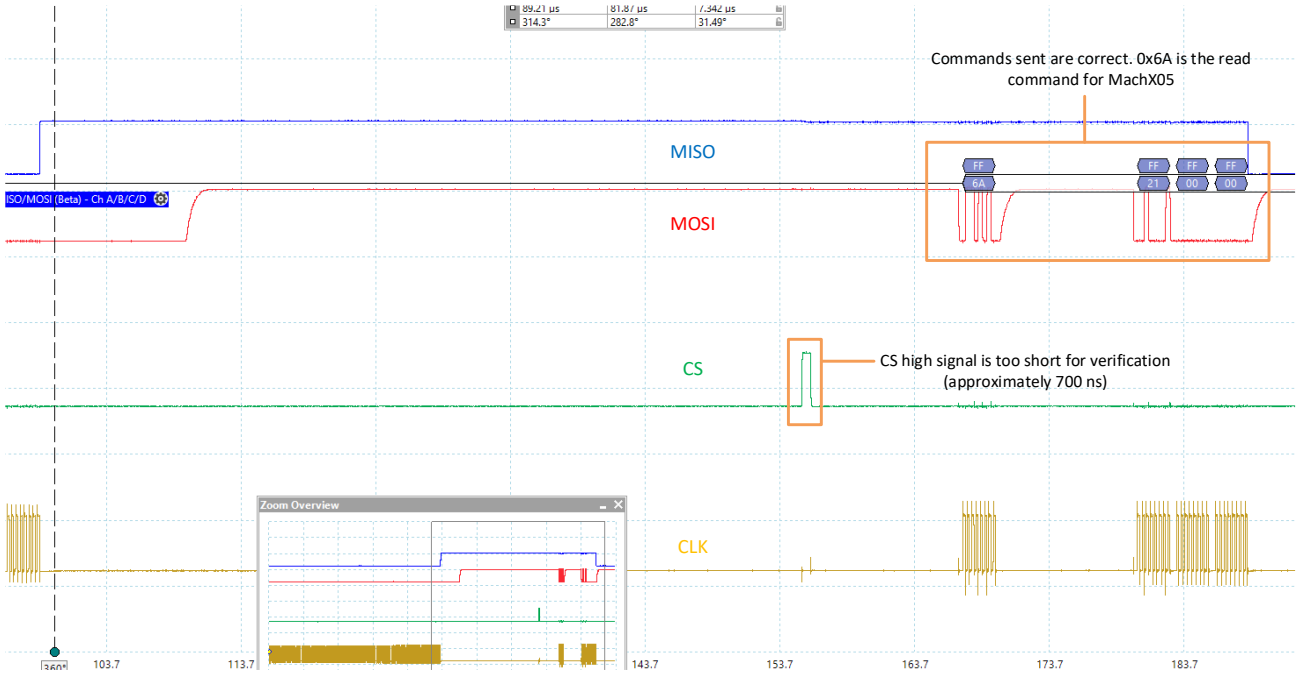

Figure 4.2. Verified Failure (CS High Signal is Too Short)

To obtain a successful implementation for comparison, testing was done with the HW USBN 2B programming cable and the Radiant Programmer. The Radiant Programmer was found to require a longer CS high signal for the verification operation as shown in Figure 4.3.

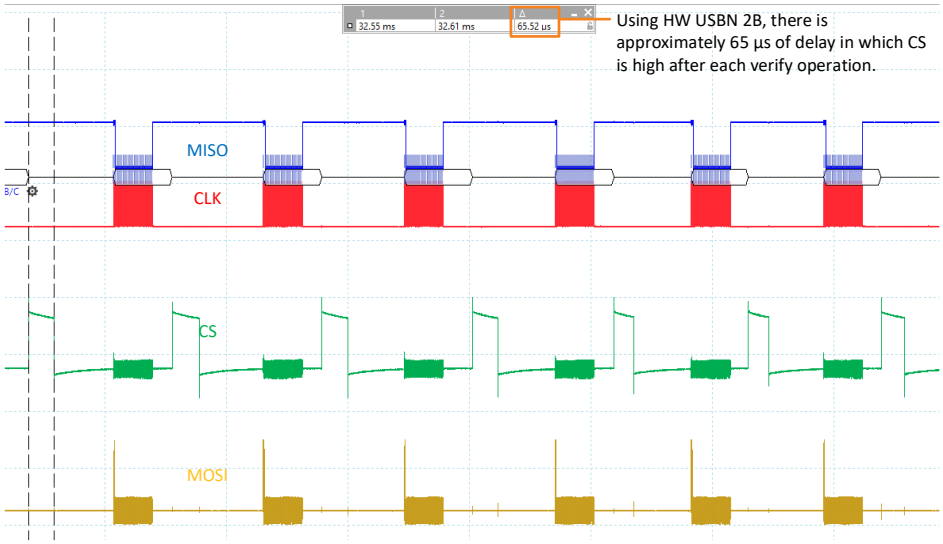

Figure 4.3. HW USBN 2B Implementation of CS High Signal

After adding delay according to rbpi_assert_cs() Implementation, embedded programming is successful.

5. Hardware Validation Summary

Table 5.1 shows the results from validation of the demo on different development boards. Failures at high frequencies can be attributed to board layout deficiencies. Check the datasheet for the maximum operating frequency of the device. Then, follow the hardware checklist of the device and use the proper layout techniques to optimize the operating frequency.

Table 5.1. Hardware Validation Results

Frequency	MachXO5™-NX Development Board	MachXO3LF Starter Kit	CrossLinkPlus LIF-MDF6000 Master Link Board (Revision B)
5 MHz	Pass	Pass	Pass
20 MHz	Pass	Pass	Fail (Pass at 19 MHz)
25 MHz	Fail	Pass	Fail

Note: When using Lattice development boards for testing, ensure that the FTDI chip of the board is disabled. Check the development board schematic or user guide for guidance on disabling the FTDI chip.

Appendix A. Generation of .sed and .sea Files

This section describes the steps for generating the .sed and .sea files needed for embedded programming. The Radiant Programmer can directly generate the .sed and .sea files. For the Diamond Programmer, the Deployment Tool is needed to generate the .sed and .sea files.

A.1. Generating .sed and .sea Files Using Radiant Programmer

1. Open the Radiant Programmer and set up your operation. For example, fast configuration of the MachXO5-NX configuration SRAM using the slave SPI port.

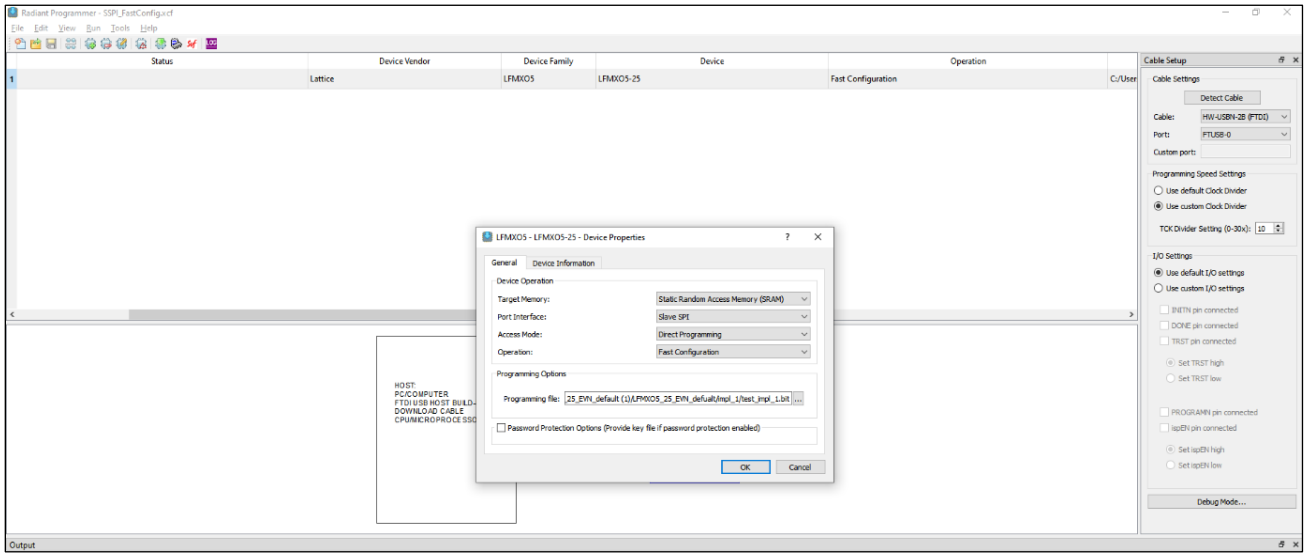

Figure A.1. Radiant Programmer .xcf File Generation

2. For the EPROM-based system, click the **Generate Embedded Code** button to open the Embedded Options window. Check the **Convert VME files to HEX (.c) for Prom-Based Embedded VME** checkbox and then click **OK**.

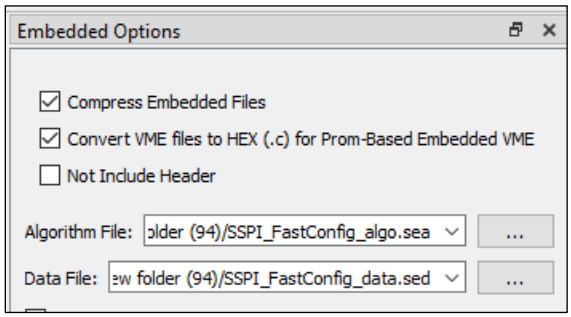

Figure A.2. Embedded Options Window

Note: You will need to write the header files for the generated .c arrays.

3. Click the **Generate Embedded Code** button.

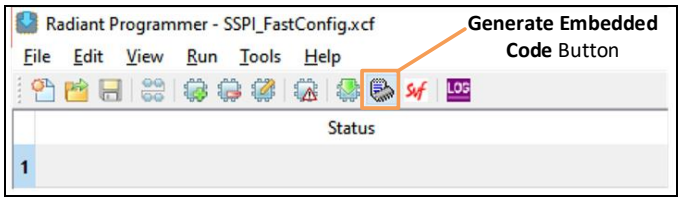

Figure A.3. Generate Embedded Code Button

The .sed and .sea files are generated as indicated in the Output window.

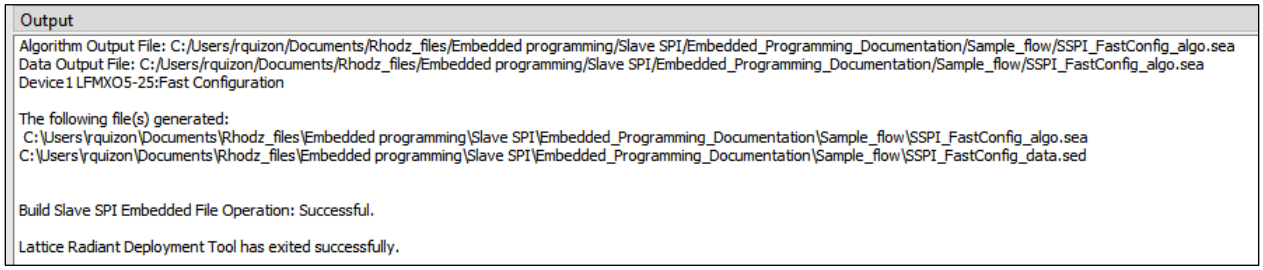

Figure A.4. Generation of .sed and .sea Files

The .sed and .sea files appear in the .xcf file directory.

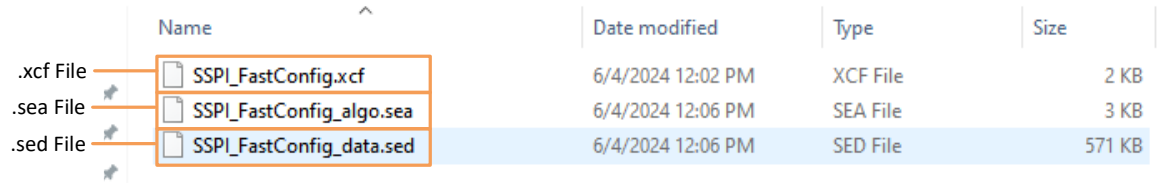

Figure A.5. Generated .sed and .sea Files in .xcf File Directory

For the EPROM-based system, c array files also appear in the .xcf file directory.

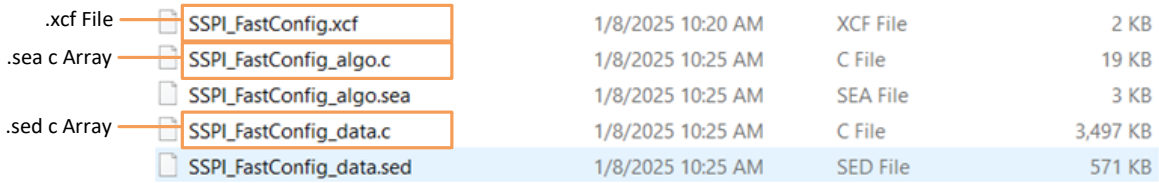

Figure A.6. Generated .c Files in .xcf Directory

A.2. Generating .sed and .sea Files Using Diamond Programmer and Deployment Tool

1. Open the Diamond Programmer and set up your operation. For example, Erase, Program, Verify for SSPI operation.

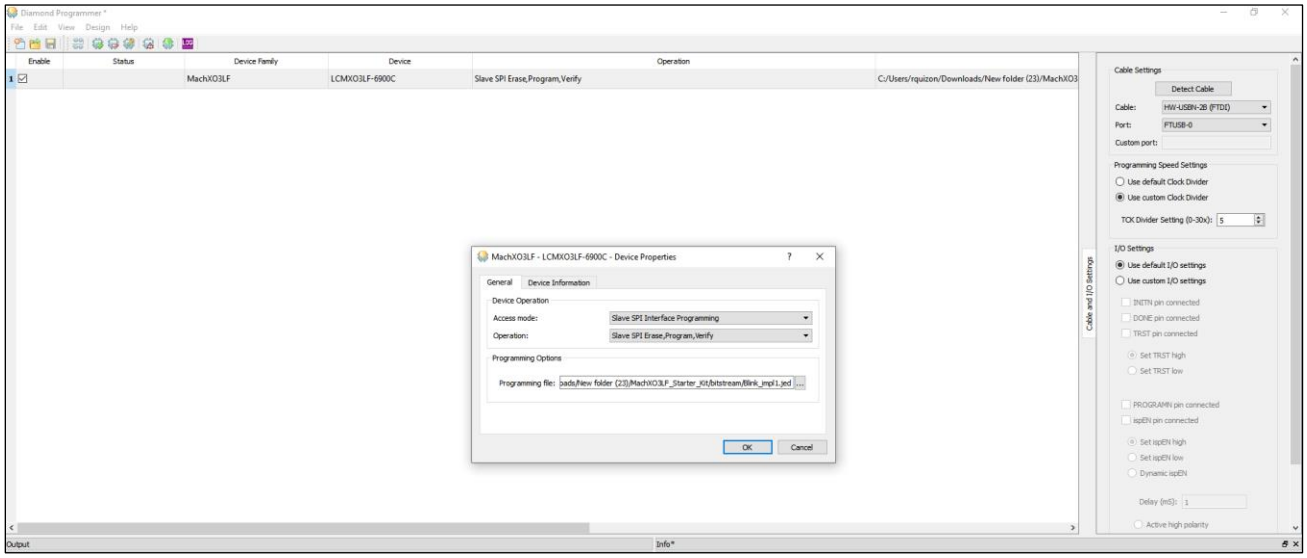

Figure A.7. Diamond Programmer .xcf File Generation

2. Save the .xcf file in a directory.
3. Open the Deployment Tool (Design>Utilities>Deployment Tool).

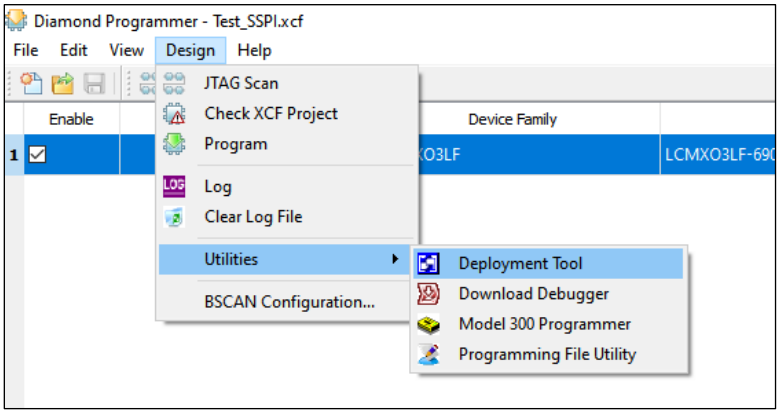

Figure A.8. Accessing the Deployment Tool

4. Select the options as shown to generate the .sed and .sea files.

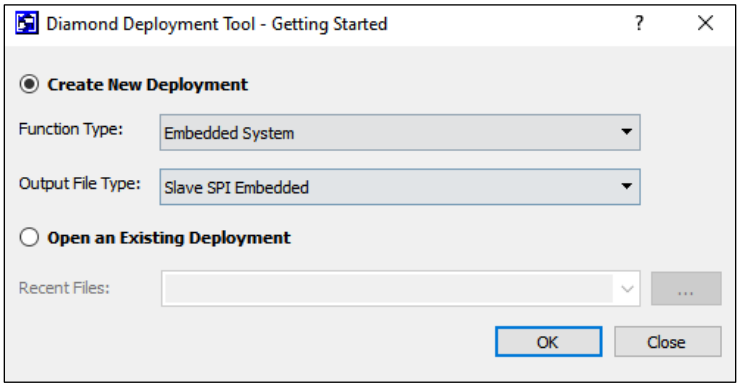

Figure A.9. Slave SPI Embedded Generation in Deployment Tool

5. Check the **Input XCF file** checkbox, select the .xcf file previously saved in step 2, and click **Next**.

Figure A.10. Selecting .xcf File

6. The settings as shown appear in the next page. Leave the settings as is for this demo. For the EPROM-based system, check the **Convert VME files to HEX (.c) for Prom-Based Embedded VME** checkbox. Click **Next.**

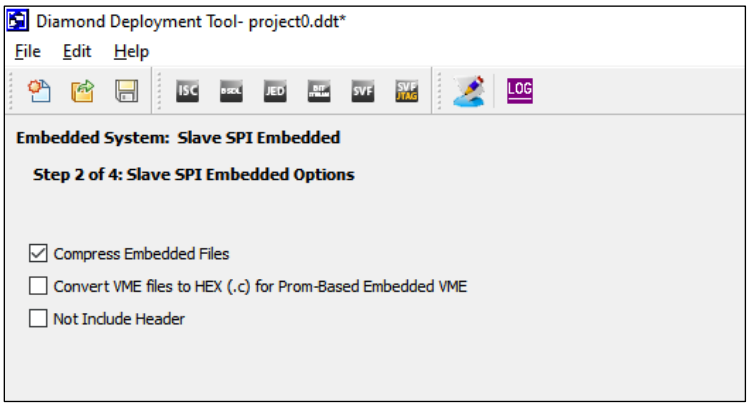

Figure A.11. Embedded File Generation Options

Note: You will need to write the header files for the generated .c arrays.

7. Confirm the location of the .sed and .sea files and click **Next**.

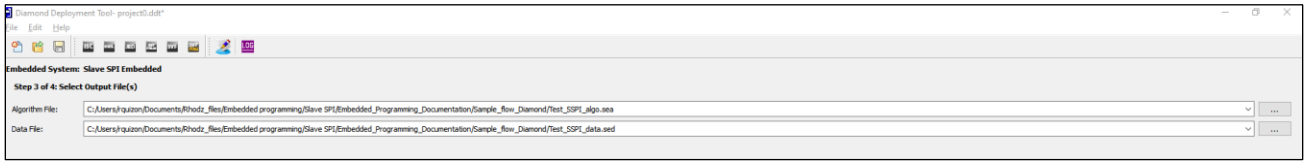

Figure A.12. Location of .sed and .sea Files

8. Click **Generate**. The .sed and .sea files are generated as indicated in the window.

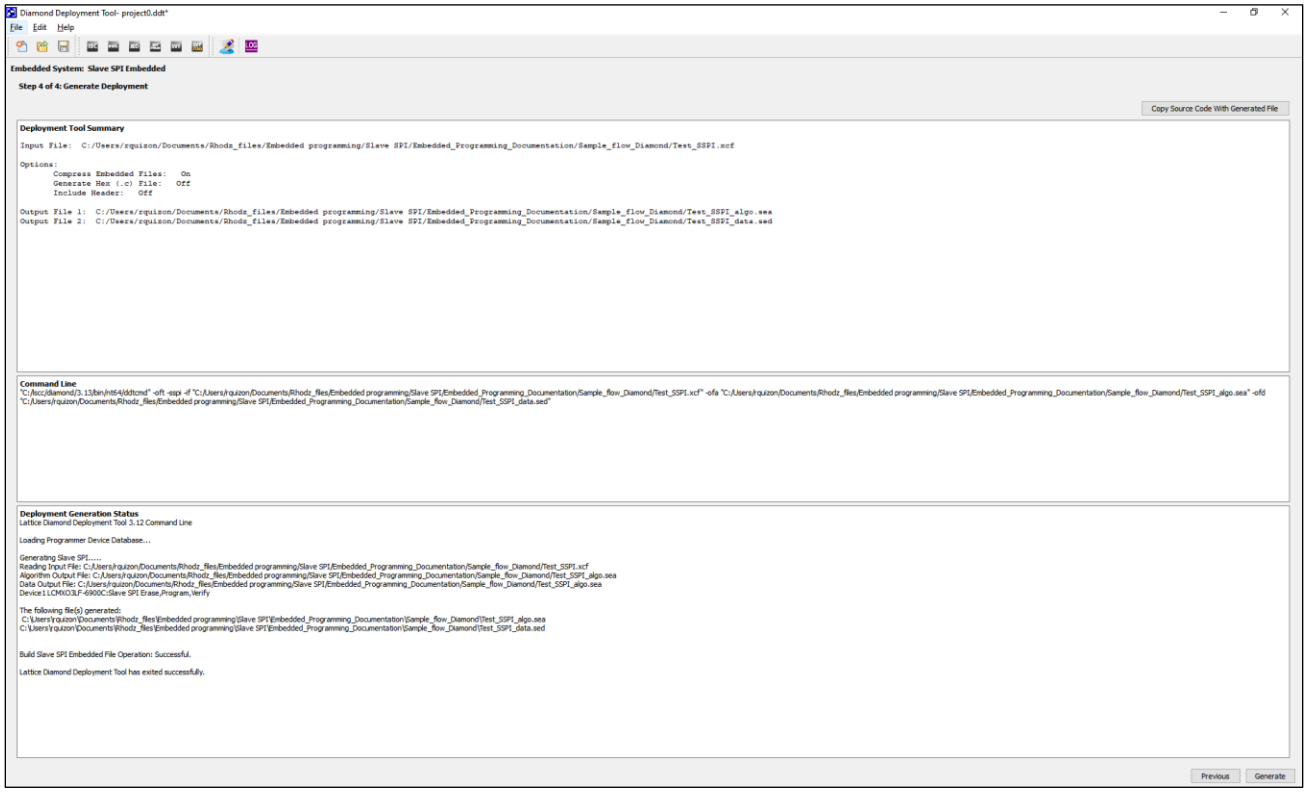

Figure A.13. Generation of SSPI Embedded Files

The .sed and .sea files appear in the .xcf file directory.

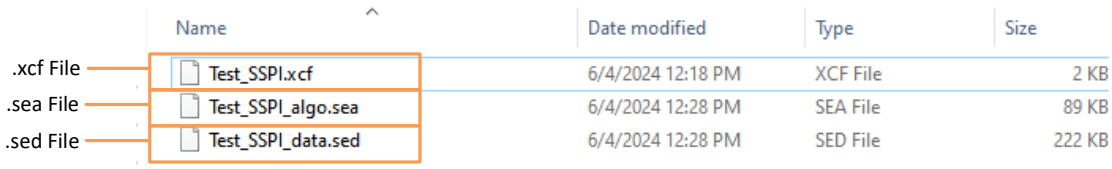

Figure A.14. Generated SSPI Embedded Files

References

- Programming Tools User Guide for Radiant Software 2024.1
- Lattice Diamond 3.13 Programming Tools User Guide
- Spidev documentation
- SSPI Embedded Programming using RPi Reference Design web page
- MachX05-NX Development Board web page
- MachX03LF Starter Kit web page
- CrossLinkPlus LIF-MDF6000 Master Link Board web page
- Lattice Solutions Reference Designs web page
- Lattice Diamond Software User Guide
- Lattice Radiant Software User Guide
- Lattice Insights for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, please refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.1, February 2025

Section	Change Summary
Abbreviations in This Document	Added *EPROM*.
Introduction	• Added description of embedded source code types in the Embedded Programming Source Code Architecture section. • Updated figure title and labels in Figure 1.4. Embedded Programming Source Code Lists for File-Based Source Code. • Added Figure 1.5. Embedded Programming Source Code Lists for EPROM-Based Source Code.
Modifying the Source Code and Writing the Driver	• Updated section title and description in the Updating main.c for File-Based Source Code section. • Added the Updating main.c for EPROM-Based Source Code section. • Added statement about hardware.c being the same for file-based and EPROM-based source codes in the Updating hardware.c section.
Compiling and Running the Demo	• Updated figure title in Figure 3.2. Example of Running the Executable File in File-Based Demo. • Added Figure 3.5. Example of Running the Executable File in EPROM-Based Demo and description.
Appendix A. Generation of .sed and .sea Files	• Added step 2 on EPROM-based system in the Generating .sed and .sea Files Using Radiant Programmer section. • Added Figure A.6. Generated .c Files in .xcf Directory and description. • Updated step 6 to include EPROM-based system in the Generating .sed and .sea Files Using Diamond Programmer and Deployment Tool section.
References	Added link to reference design web page.

Revision 1.0, September 2024

Section	Change Summary
All	Initial release.

www.latticesemi.com