1.0 Introduction to AmP chip programming

A power solution creation using the AmP chip involves several steps, beginning from a system power tree, followed by obtaining a solution through 3 different options (see flowchart) and culminating in programming the AmP chip to be ready for deployment in the field. This application note focuses on the various ways an AmP chip can be programmed for installation on customer PCB for mass production.

2.0 Programming the AmP chip

Please note that throughout the document master and slave modes are mentioned. These modes are in reference to the AmP device.

The AmP platform supports two modes of configuration through the SPI-compliant serial interface (Serial Peripheral Interface):

In the slave mode, the AmP device is loaded with its configuration file (.HAX) by an external processor or controller or the AmPLink.

In the master mode, the AmP device loads its configuration file (.HEX) from an external non-volatile memory. Please refer to the table “Pin functions and assignments” for details on master and slave pin assignment.

3.0 Program the AmP chip(s) using external controller/processor (Slave mode)

AmP devices can be configured at boot time through the SPI bus from an MCU/processor. In this case the AmP device becomes an SPI slave and the MCU/processor (master) sends the contents of the “design_slv.hax” file to the AmP chip (slave) and verifies the checksum of the received data calculated by the AmP chip against the checksum stored in the “design_slv.hax” file.

The file “design_slv.hax” is generated by the WeAmP tool and can be obtained through option A, B, and C as shown in the flow chart on page 1.

Slave Mode Diagram

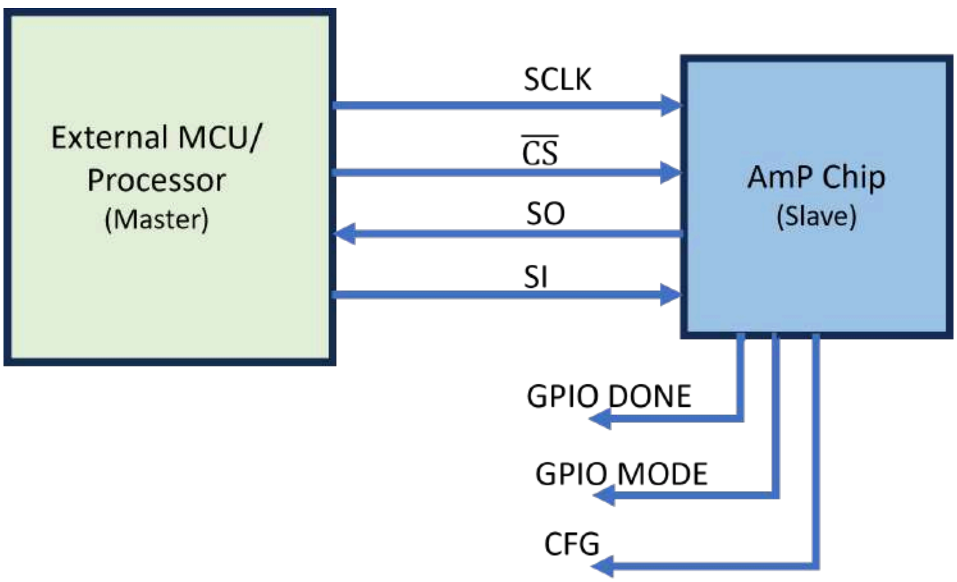


Figure 1. Programming AmP chip using an external controller/processor/AmPLink

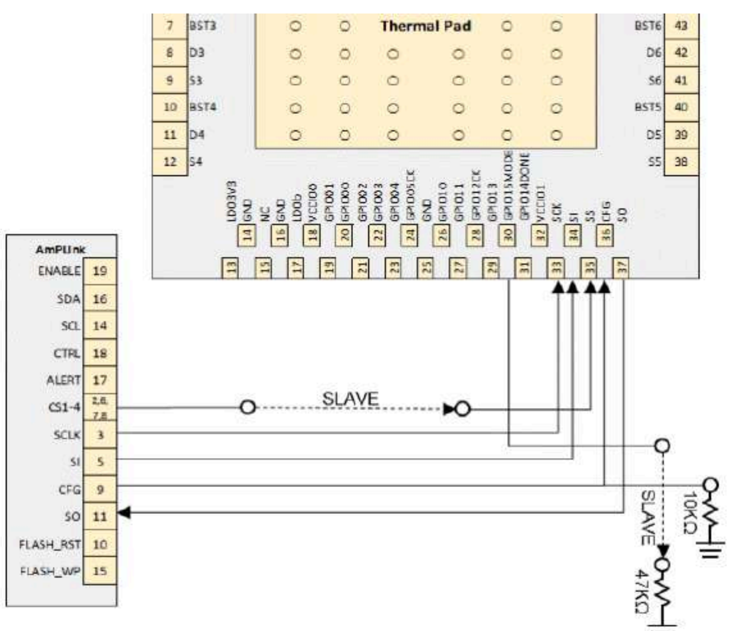
Pin Functions and Descriptions

Function	Description	
SCLK	SPI Clock input when AmP is slave; output clock when AmP is master (Active Low)	
CS	SPI Chip Select line. Input when AmP is slave, output when AmP is master	
SO/MISO	SPI Serial Out transmits SPI commands	
SI/MOSI	SPI Serial In input receives SPI data	
GPIODONE	Before config, pin is shared with DONE output. Pin is pulled low once device config is successfully finished and subsequently can be used as a normal GPIO	
GPIOMODE	Wakes AmP chip up. Pin serves dual function:	
	Before config	Mode function
	After Config	Normal GPIO
	Mode function:	
	Master mode	Pulled high to VCCIO (3.3 V) through 47 kΩ
	Slave mode	Pulled low to GND through 47 kΩ
CFG	Config pin. **Pulled low to GND through 10 kΩ**. CFG input states:	
	Positive edge	AmP held in reset
	Negative edge	AmP reconfig starts
	Any other states?? Constant Low? High?	

Basic Setup

A basic setup to configure the AmP device using an external processor/controller is shown in Figure 1. The processor is the master which drives the SPI lines: SCLK, CS, and SI pins to configure the slave (AmP chip).

AmP Slave Mode



SCLK, CS, SO, SI, GPIO DONE, GPIO MODE, CFG pin configuration when AmP device is in Slave mode

AmP devices follow SPI protocol for configuration. SPI comes in several varieties, the AmP devices follow the following SPI convention:

SPI Mode	CPOL (Clock Polarity)	CPHA (Clock Phase)	Clock Polarity in Ideal State	Clock Phase used to Sample and/or Shift Data
3	**1**	**1**	Logic High	Data sampled on the rising edge and shifted out on the falling edge
0	**0**	**0**	Logic Low	Data sampled on the rising edge and shifted out on the falling edge

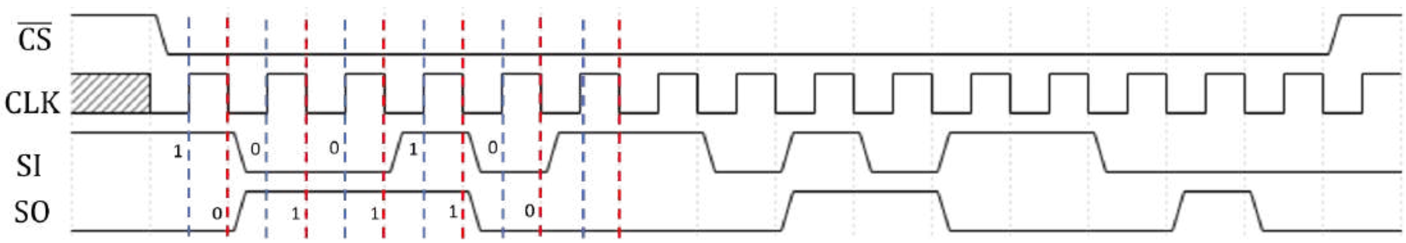


Figure 2. A timing diagram for Mode 3 showing clock polarity and phase

In slave mode, the AmP device can operate in systems where there are one or more devices on the single SPI bus. It will act as a proper slave device and only drive its output when the Chip Select (CS) for the device is activated. This enables multiple AmP devices on the same SPI bus or combinations with AmP devices and other suitable SPI devices on the same bus. When the AmP device is not selected its output SO pin will be high impedance.

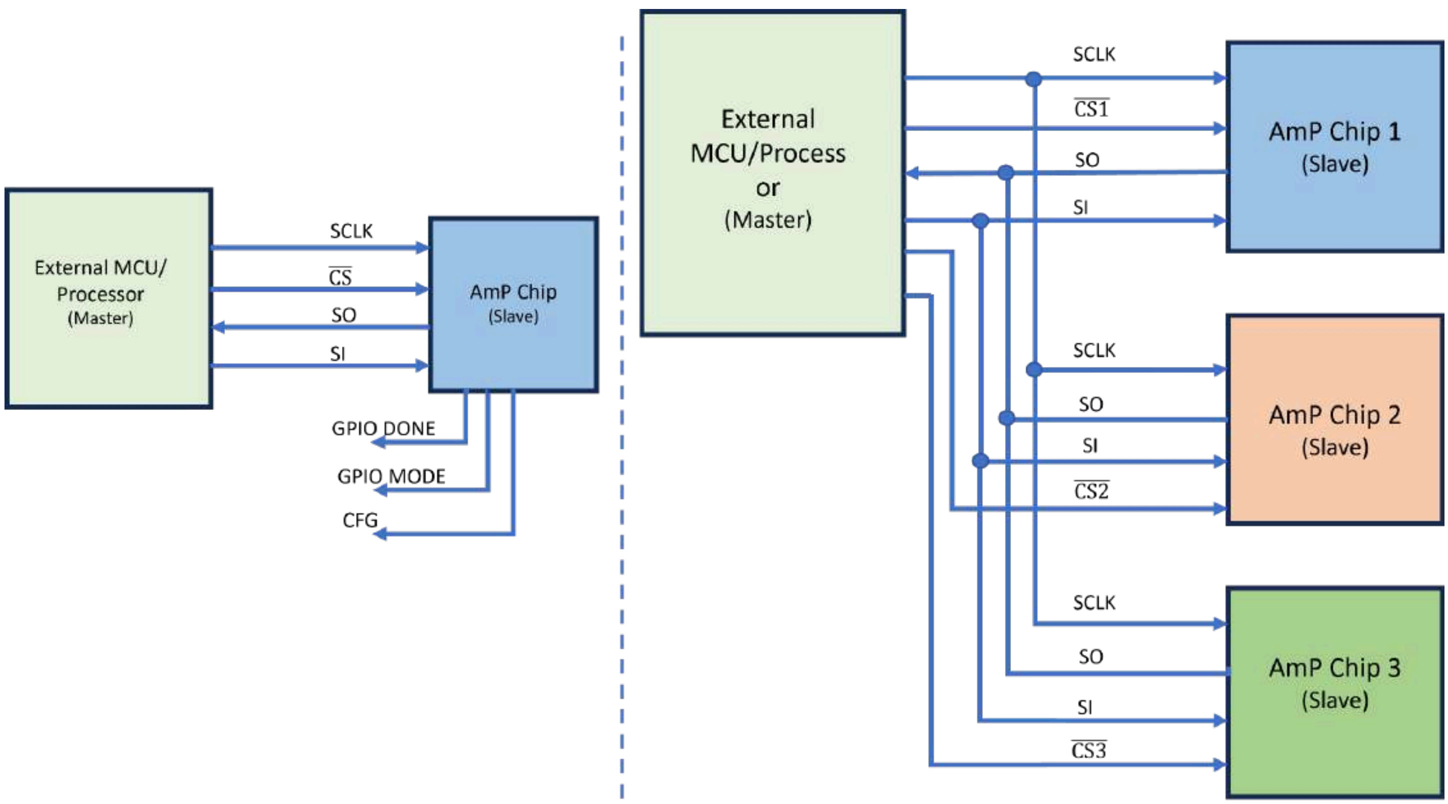


Figure 3. a) Single master and single slave. b) Single master and three independent slaves

Hax File Description

The .hax file contains everything required to program and verify the Amp device configuration. Some of the important entries are described below

```
1b 00 00 00                                       # report status
1b 00 00 00
44 00 ff ff                                       # indicate data width
ad ba da 55                                       # valid data authenticate
11 28 17 08                                       # load config image
00 01 00 80 00 00 18 00 24 04 00 24 24.......     # config data
.........................................................
ec 87 e5 00                                       # error check request + checksum for comparison
1b 00 00 00                                       # report status
d7 00 00 00 00 00 00 00 00 00 00 00               # slave done / complete config + NOPs
```

To program a device from an MCU it is simply a case of serializing this file and sending it to the SPI.

Example C code to do this is shown next

C code for a MCU / single board computer

'''

```
#include <stdio.h>
#include <stdlib.h>

#define BLOCK_SIZE 1024

#define STATUS1_POS 29626
#define STATUS2_POS 29627
#define STATUS1 0x01
#define STATUS2 0x58

void sendToSPI(unsigned char *data, size_t size) {
   // Implement your SPI sending logic here

   printf("Sending to SPI: \n");
   for (size_t i = 0; i < size; ++i) {
      printf("%d, ", data[i]);
   }
   printf("\n");
}

int main() {
   // Open .hax file
   FILE *file = fopen("test_slv.hax", "rb");
   if (file == NULL) {
      perror("Error opening file");
      return 1;
   }

   // Determine the file size
   fseek(file, 0, SEEK_END);
   size_t file_size = ftell(file);
   fseek(file, 0, SEEK_SET);

   // Allocate memory to store the entire file
   unsigned char *file_data = (unsigned char *)malloc(file_size);
```

```
if (file_data == NULL) {
    perror("Error allocating memory");
    fclose(file);
    return 1;
}

// Read the entire file into memory
size_t bytes_read = fread(file_data, 1, file_size, file);
if (bytes_read != file_size) {
    perror("Error reading file");
    free(file_data);
    fclose(file);
    return 1;
}
fclose(file);

// Print the hex data
printf("Hex Data:\n");
for (size_t i = 0; i < file_size; ++i) {
    printf("%c", file_data[i]);
    if ((i + 1) % 2 == 0) {
        printf(" ");
    }
}
printf("\n");

// Assuming file_data now contains the hex data, convert it to a list of bytes
size_t hex_count = file_size / 2;
unsigned char *spi_data = (unsigned char *)malloc(hex_count);
if (spi_data == NULL) {
    perror("Error allocating memory");
    free(file_data);
    return 1;
}

printf("\nList of Bytes:\n");
for (size_t i = 0; i < hex_count; ++i) {
    char byte[3] = {file_data[i * 2], file_data[i * 2 + 1], '\0'};
    spi_data[i] = strtol(byte, NULL, 16);
    printf("%d, ", spi_data[i]);
}
printf("\n \n");

// Send the data to SPI in blocks of 1024 bytes
size_t offset = 0;
while (offset < hex_count) {
    size_t block_size = (offset + BLOCK_SIZE < hex_count) ? BLOCK_SIZE : (hex_count - offset);
    sendToSPI(&spi_data[offset], block_size);
    offset += block_size;
}

// Verify the checksum on the device response
unsigned char status1_read, status2_read;
// Assuming you have a function to read data from the SPI peripheral
// Example: receiveFromSPI(&status1_read, 1);
// You should replace this with your actual function to read from SPI
```

```
    // The loop is just for illustration purposes; adjust as needed
    for (size_t i = 0; i < hex_count; ++i) {
        if (i == STATUS1_POS) {
            status1_read = spi_data[i];
        } else if (i == STATUS2_POS) {
            status2_read = spi_data[i];
        }
    }

    // Verify the checksum
    if (status1_read == STATUS1 && status2_read == STATUS2) {
        printf("\nChecksum verification: OK\n");
    } else {
        printf("\nChecksum verification: FAILED\n");
    }

    // Clean up
    free(file_data);
    free(spi_data);

    return 0;
}
```

Python code for a MCU / single board computer

```
'''

# convert a hax file from this format :

1b 00 00 00

1b 00 00 00

44 00 ff ff

ad ba da 55

11 28 17 08

00 01 00 80 00 00 18 00 24 04 00 24 24 00 04 00 00 00 24 24 00 04 0e 04 88 08 00 21 22 01 04 00 00 20.....


# To a list of bytes :

27, 0, 0, 0, 27, 0, 0, 0, 68, 0, 255, 255, 173, 186, 218, 85, 17, 40, 23, 8, 0, 1, 0, 128, 0, 0, 24, 0, 36, 4, 0, 36, 36, 0, 4, 0, 0, 0, 36, 36, 0, 4, 14, 4, 136, 8, 0, 33, 34, 1, 4, 0, 0, 32......


# and send them to the SPI perhipheral in blocks of 1024 (limitation of this SPI Master)


# then verify the checksum on AMP device

'''


# import libraries to control the SPI and GPIO

import Adafruit_BBIO.GPIO as GPIO

from Adafruit_BBIO.SPI import SPI

import time

import os


# define the haxfile name

haxfilename = "I483_chip_100_800pps_slv_2MHz-I480CLK.hax"


# the hax file contains a checksum which the Amp device verifies and returns a status

# the byte position where the checksum status is returned is defined here

status1pos = 29626

status2pos = 29627

status1 = 0x01

status2 = 0x58
```

```
# configure the SPI pins
os.system("config-pin p9.17 spi_cs  > /dev/null")      # AMP SS
os.system("config-pin p9.18 spi  > /dev/null")         # AMP SI
os.system("config-pin p9.21 spi  > /dev/null")         # AMP SO
os.system("config-pin p9.22 spi_sclk  > /dev/null")    # AMP SCLK

# toggle the config pin to reset the AMP device
# this is optional, only needed when loading a
# configuration into an already configured device
reloadconfig = True
#reloadconfig = False
if (reloadconfig) :
    GPIO.setup("P9_23", GPIO.OUT)   # AMP CFG
    GPIO.output("P9_23", GPIO.HIGH)
    GPIO.output("P9_23", GPIO.LOW)
    GPIO.setup("P9_23", GPIO.IN)    # release the CFG, its pulled low on board

try :
    # open the haxfile
    fyle = open(haxfilename,"r")
    print("opened: %s"%(haxfilename))

    # read the entire file
    rawdata = fyle.read()

    # close the file
    fyle.close()

    # replace any newlines in the file with spaces
    haxdata = rawdata.replace("\n"," ")

    # split into a list of hex bytes
    hexbytes = haxdata.split()
```



```
# convert the (string) hex data to a list of integers
SPIdata = []
hexcount = 0
for byte in hexbytes :
    SPIdata.append(int(byte,16))
    hexcount += 1       # keep track of the number of bytes converted

# connect to the SPI perhipheral
# spi = SPI(bus, device)
spi = SPI(0, 0)

# set to 10MHz
# msh - Maximum speed in Hz
spi.msh = 10000000

# set SPI mode
# mode - SPI mode as two bit pattern of Clock Polarity and Phase [CPOL|CPHA]; min– 0b00 = 0, max– 0b11 = 3.
# AMP device follows the 11 protocol
spi.mode = 0b11

# spi.xfr() can only handle 1 to 1024 bytes per call
# so take blocks of 0-1023 bytes to send to SPI
firstbyte = 0
txsize = 1023

bytesent = 0
bytesread = []
allbytesread = []

while (txsize) :
    # calculate the last byte to send
    lastbyte = firstbyte+txsize
    # send the bytes & read the result
    bytesread = spi.xfer2(SPIdata[firstbyte:lastbyte])
```

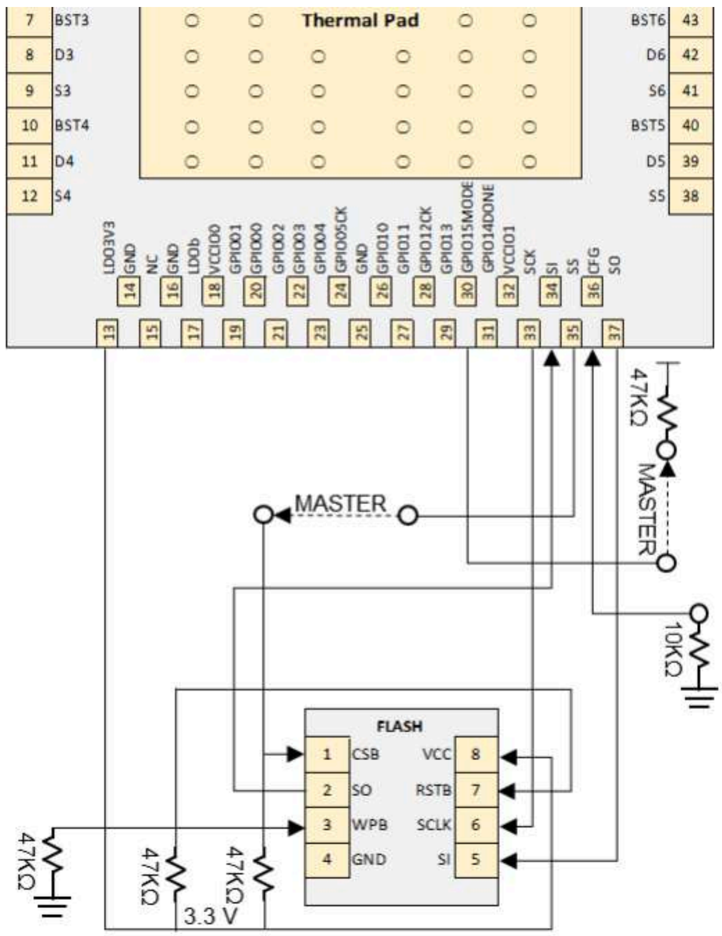
```
        # keep the read back status bytes for verify later
        for b in bytesread :
            if (bytesent == status1pos) : status1read = b
            if (bytesent == status2pos) : status2read = b
            bytesent += 1

        # move on to the next block of bytes
        firstbyte = firstbyte + txsize

        # check that the end of the block to be sent is not past the end of list of bytes
        if ((firstbyte + txsize)  > hexcount) :
            # if it is, send only up to hexcount
            txsize = (hexcount - firstbyte)

        # stop when firstbyte will go past the end of list of bytes
        if (firstbyte > hexcount) : txsize = 0

# report any error in opening the hax file
except FileNotFoundError as fnf_error :
    print(fnf_error)

# verify the AMP checksum
# the hax file contains an error check request + checksum and status request near the end
# ec 87 e5 00  <- error check + checksum
# 1b 00 00 00  <- status request
# AMP chip should respond with "01 58" to this
if ((status1read == status1) & (status2read == status2)) : verify = "OK"
# If not, verify has failed
else : verify = "FAILED"

# report the programming result
print("DONE: %d bytes read, %d bytes sent, verify: %s" %(hexcount,lastbyte,verify))
```

```
# release the SPI pins

os.system("config-pin p9.17 gpio > /dev/null")

os.system("config-pin p9.18 gpio > /dev/null")

os.system("config-pin p9.21 gpio > /dev/null")

os.system("config-pin p9.22 gpio > /dev/null")
```

4.0 Program the AmP chip(s) Using External SPI Chip (AmP is Master)

The AmP device simply receives valid input power and takes control of the external SPI memory chip to load its configuration/ The AmP device acts as a SPI master and controls the external SPI as a slave. Master mode is ideally suited for applications where the AmP device is independently providing power to the SPI chip.

Please note that for the mass production stage, the SPI memory chip can be pre-programmed beforehand board assembly.



SCLK, CS, SO, SI, GPIO DONE, GPIO MODE, CFG pin configuration when AmP device is in Master mode

Serial Peripheral Interface (SPI) Flash Memory Selection Note

AnDAPT recommends generic 256k-bits or greater density SPI flash memory devices with a single supply. The requirements are:

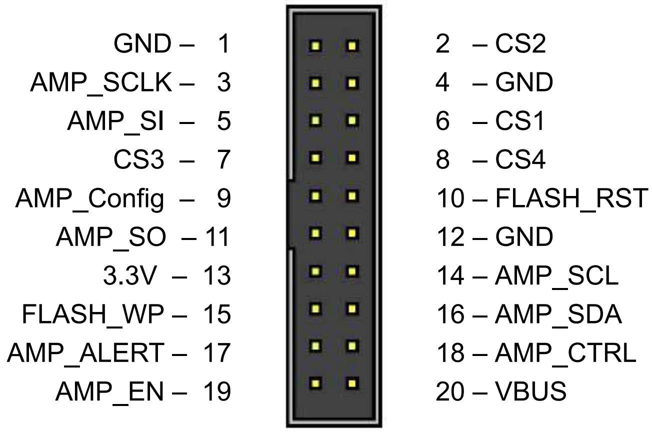
- Serial Peripheral Interface (SPI compatible)
  - Supports most common SPI Modes 0 or 3
  - Single supply (up to 3.3 V)
  - Command compatible with following SPI flash memories:

Adesto AT25DN256
Adesto AT25DF512C
Macronix MX25R8035F
Macronix MX25R512F
WinBond WX25X05CL
WinBond WX25X20CL
Micron M25P05-A
ISSI IS25LQ025B
ISSI IS25LQ512B

5.0 Powering Up Customer Board Using AmPLink USB Adaptor

Pre-production stage or evaluation/debugging stage might require programming the device or its corresponding SPI flash memory counterpart.The AmPLink USB adapter provides the hardware interface between the AmP device and the PC. It is used in conjunction with the AmPLink Control software to program and control the AmP device and/or flash memory.

AmPLink Pinout



AmPLink Functional Description

The AmPLink USB Adapter provides SPI, I2C and GPIO interfaces to the AmP evaluation board. The SPI bus is used to control the AmP device and program both AmP and flash memory. The I^2C bus provides control and monitoring of the power supply functions of the AmP device. GPIO is used for evaluation board configuration and to support functions on the SPI interface. All pins use 3.3V logic except where otherwise stated.

Pin Functional Description

SPI	
AMP_SCLK	Clock output Hi-Z when not in use
AMP_SI	MOSI output when communicating with AmP devices MISO input when programming flash devices Hi-Z when not in use
AMP_SO	MISO input when communicating with AmP devices MOSI output when programming flash devices Hi-Z when not in use
CS1, CS2, CS3, CS4	Chip select outputs Hi-Z when not in use
I^2C	
AMP_SCL	Clock output Open drain with internal 2.2kΩ pull up resistor
AMP_SDA	Bidirectional data line Open drain with internal 2.2kΩ pull up resistor
AMP_ALERT	alert signal input
AMP_CTRL	control signal output
Configuration	
AMP_EN	AmP device enable output
AMP_Config	Configures AmP device (see AnDAPT_AmP_Platform datasheet)

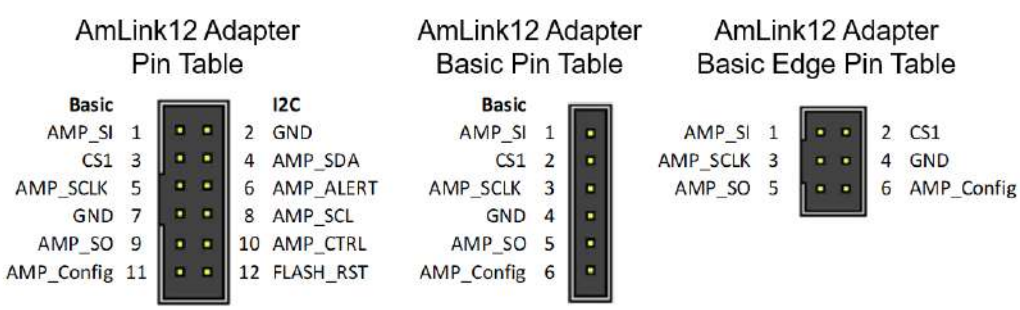
FLASH_WP	Flash write protect output
FLASH_RST	Flash reset output
Power	
GND	Connected to USB GND and shield
VBUS	5V output with 0.5A to 0.7A current limiting
3.3V	3.3V output with 0.5A current limiting

Reduced Pin Count AmPLink12 Adapter Extension

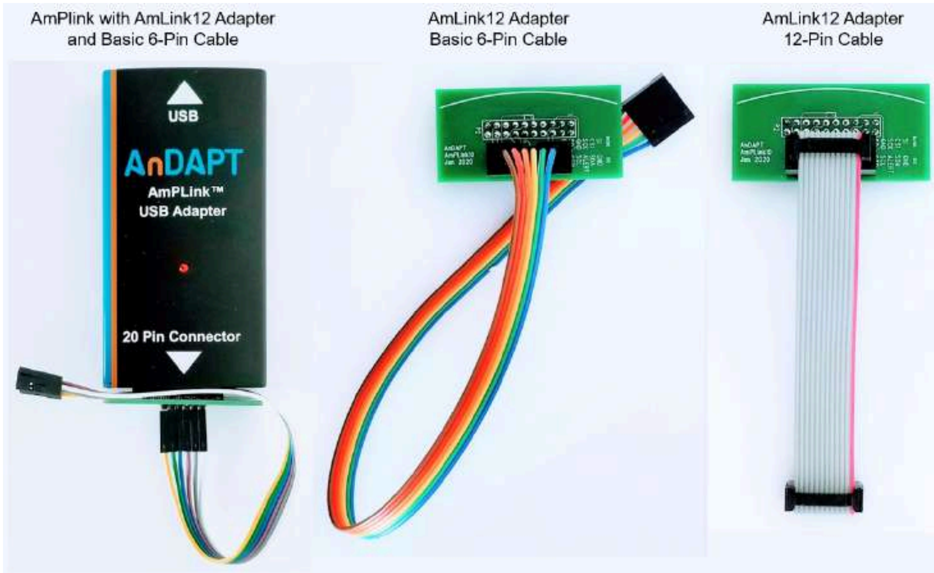
The AmPLink12 Adapter provides reduced pin counts for applications not requiring all the functionality of the 20-pin interface. This enables the application to have a smaller footprint with fewer connections. Three Standard Interface pinouts are recommended and supported as defined follows:

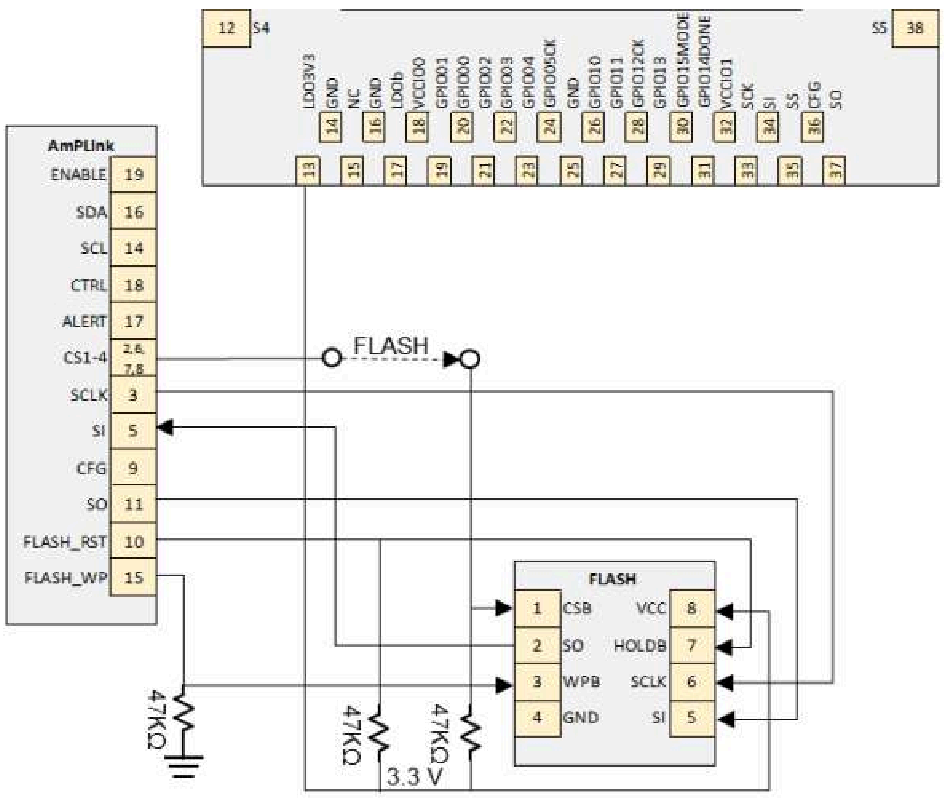
Standard Interface	Total Pins	SPI	I2C	Multi-Chip Prog Support	Pin Pitch (inch)	Cable Length (inch)
AmPLink12	12	Yes	Yes	Yes	0.1	4
AmPLink12 Basic	6	Yes	No	No	0.1	4
AmPlink12 Basic Edge	6	Yes	No	No	0.1	4

AmPLink12, Basic, and Basic Edge Pinouts



AmPLink Images





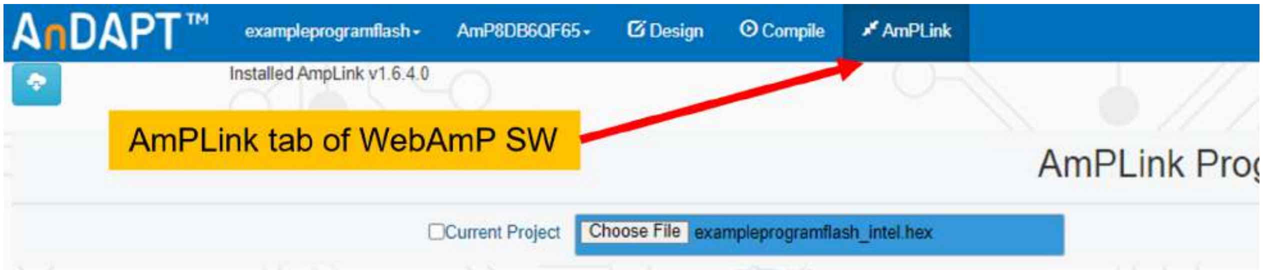
AmPLink connection to program SPI flash

Once the AmPLink is connected to the SPI flash memory as shown above, the user can follow instructions in Section 6.0 to program the SPI flash memory using an offline AmPLink tool.

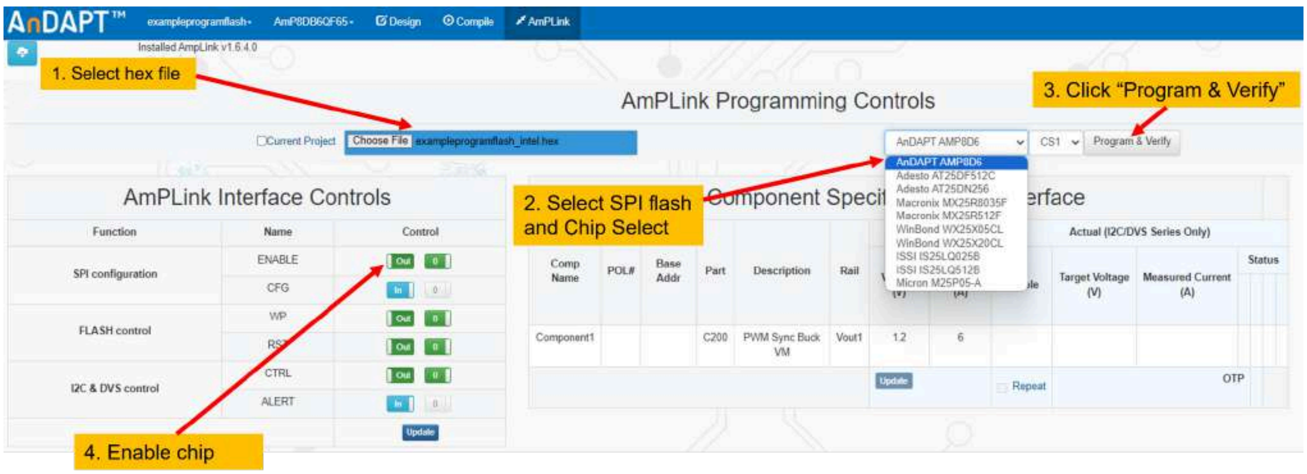
6.0 Program SPI Flash Memory Using AnDAPT's AmPLink Tool (Online and Offline)

Online Programming Using WebAmP

For online programming of SPI flash using AmPLink tool and WebAmP, user requires to login to WebAmP and navigate to the "AmPLink" tab-



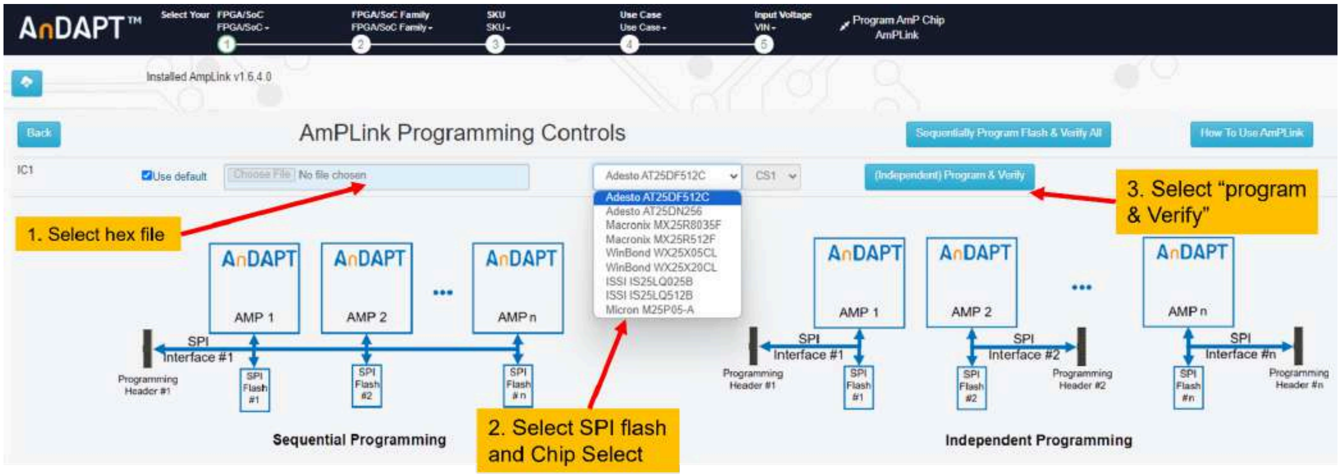
Next, follow the instructions as shown:



Online Programming Using WebAmP R.D.

For online programming of SPI flash using AmPLink tool and WebAmP R.D., user requires to navigate to the "Program AmP Chip" tab of WebAmP R.D. as shown:

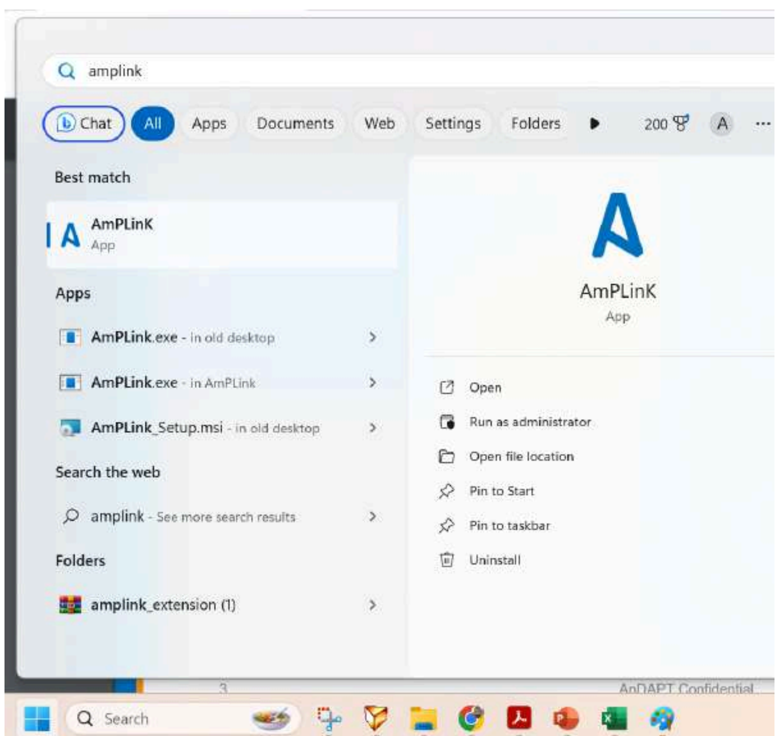
Next, follow instructions as shown,



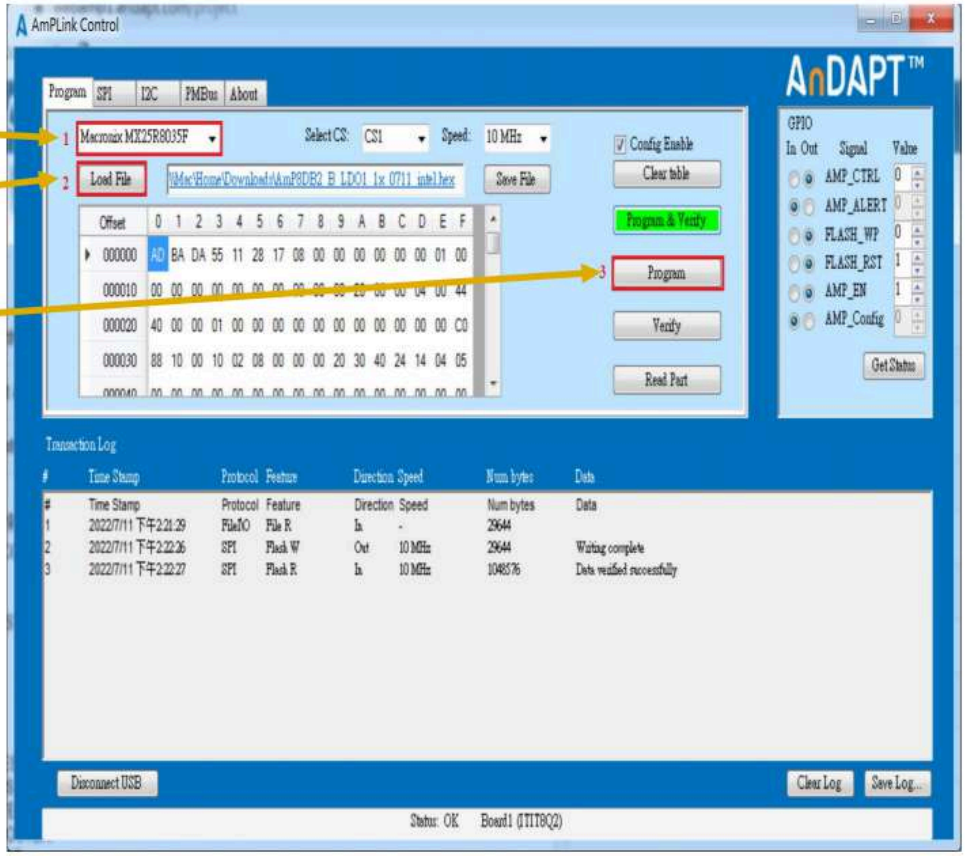
Offline Programming Using AmPLink Tool:

Instructions for offline programming of SPI flash using AmPLink tool:

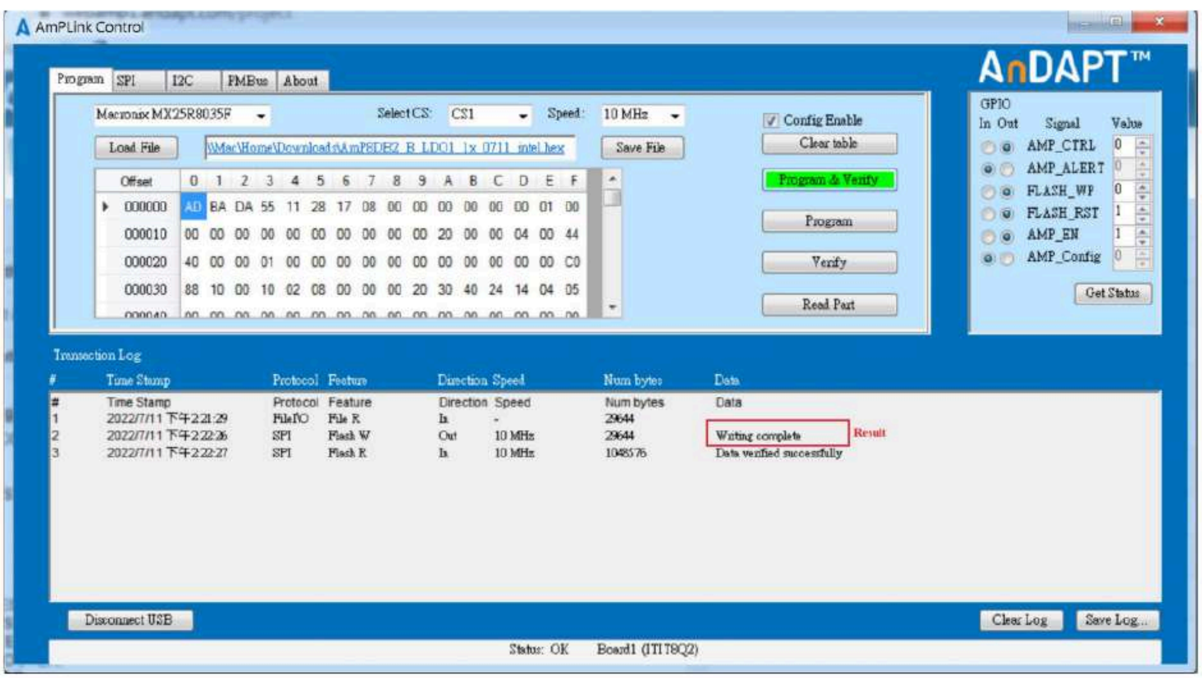
1. Please connect the AmPLink adapter between PC and the board followed by supplying input voltage to the board.
2. Open native application:



3. Select device (Macronix MX25R8035F)
4. Load .Hex file
5. Press "Program & Verify":

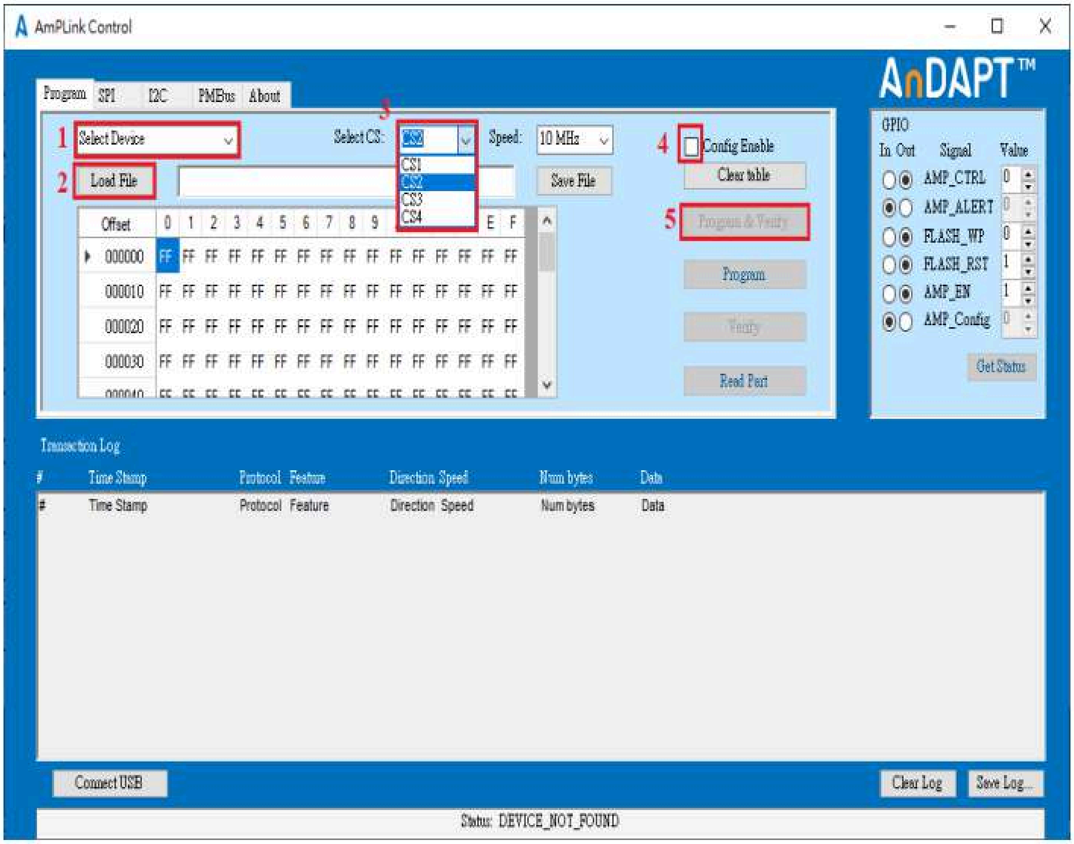


6. Confirm write to SPI:



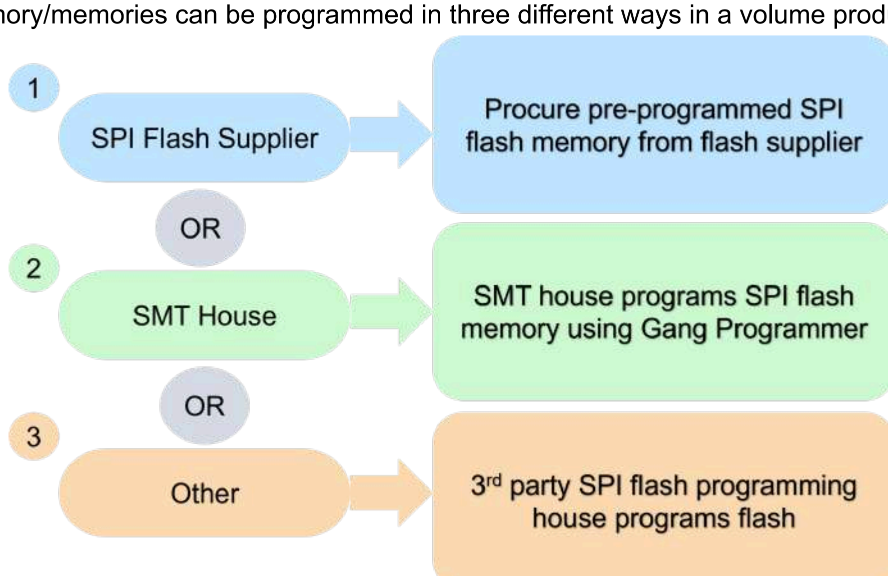
7. Program 2nd or 3rd SPI (if needed)
    1. Select deice (Macronix MX25R8035F)
    2. Load hex file
    3. Select CS2 (for 2nd chip), CS3 (for 3rd chip)

4. Unclick “Config Enable”
5. Press “Program & Verify”:



7.0 Other Options for Programming SPI Flash in Volume Production Environment

The SPI flash memory/memories can be programmed in three different ways in a volume production environment.



Revision History

Date	Revision
08/16/2022	Initial version
8/14/2023	Revised version 2.0 with flowchart and SPI programming added
1/4/2024	Revised version with C code, offline programming instructions, tables, and connection instructions in section 2, 3
1/9/2024	Added Python code, revised flowchart
3/18/2024	Details added about Master mode

On-Demand Power Management

www.AnDAPT.com

Trademarks