TRAINING MANUAL

SAM D21 XPRO USB Host MSC Bootloader

AN-8185

Prerequisites

- **Hardware Prerequisites**
  - Atmel® | SMART SAM D21 Xplained Pro Evaluation kit
  - Atmel OLED1 Xplained Pro Extension board
  - Micro-USB standard OTG cable
  - Micro-USB standard device cable
  - An empty USB storage device, for example a thumb-drive
- **Software Prerequisites**
  - Atmel Studio 6.2 (Beta or higher)
  - Atmel Software Framework (ASF) (3.19.0)
- **Estimated completion time**: 90min.

Introduction

Products with microcontrollers embedded will typically be shipped to the market with the firmware loaded into the part.

Whenever a new feature is implemented or a bug is fixed, the firmware on the product needs to be updated in the field for the updates to take effect. Lately this has often been done by either connecting an external programmer or a PC with special update software running on it. The process of updating the firmware becomes a lot easier if the product has the capability of updating its firmware by itself.

With USB host mode and self programming flash supported on the device it will be possible to do this by simply connect a USB disk with new firmware stored in it to the application and the MCU can upgrade its own firmware directly.

In this hands-on training we will develop a USB Host bootloader project for SAM D21 device. We are going to develop a bootloader that can detect a mass storage device (for example a USB thumb-drive) when connected to the USB-port. If this device contains an updated firmware image, the bootloader can update the flash of the device with new firmware.

The OLED display on the OLED1 Xplained Pro is used to give status of the firmware update process.

Table of Contents

Icon Key Identifiers

INFO Delivers contextual information about a specific topic.

TIPS Highlights useful tips and techniques.

TO DO Highlights objectives to be completed.

RESULT Highlights the expected result of an assignment step.

WARNING Indicates important information.

EXECUTE Highlights actions to be executed out of the target when necessary.

1 Training Module Architecture

This training material can be retrieved through different Atmel deliveries:

- As an Atmel Studio Extension (.vsix file) usually found on the Atmel Gallery web site (http://gallery.atmel.com/) or using the Atmel Studio Extension manager
- As an Atmel Training Executable (.exe file) usually provided during Atmel Training sessions

Depending on the delivery type, the different resources used in this training material (hands-on documentation, datasheets, application notes, software, and tools) will be found on different locations.

1.1 Atmel Studio Extension (.vsix)

Once the extension installed, you can open and create the different projects using "*New Example Project from ASF...*" in Atmel Studio.

INFO The projects installed from an extension are usually under "*Atmel Training > Atmel Corp. Extension Name*".

There are different projects which can be available depending on the extension:

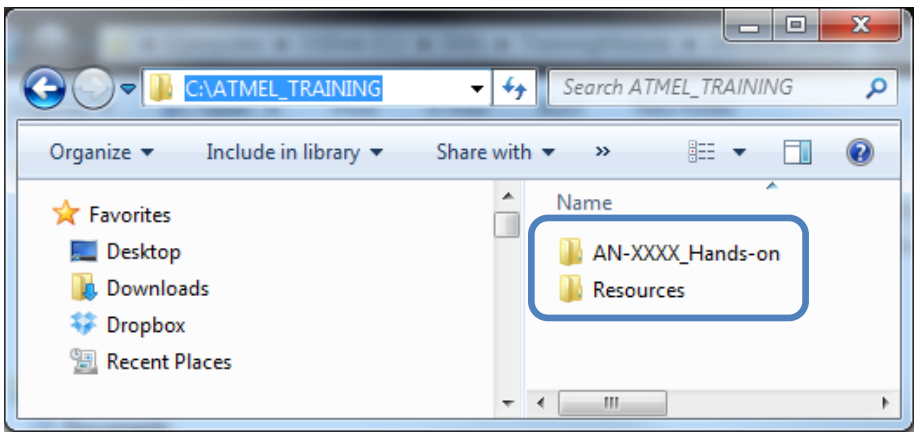
- **Hands-on Documentation**: contains the documentation as required resources
- **Hands-on Assignment**: contains the initial project that may be required to start
- **Hands-on Solution**: contains the final application which is a solution for this hands-on

INFO Each time a reference is made to some resources in the following pages, the user must refer to the Hands-on Documentation project folder.

1.2 Atmel Training Executable (.exe)

Depending where the executable has been installed, you will find the following architecture which is composed by two main folders:

- AN-XXXX_Hands-on: contains the initial project that may be required to start and a solution
- **Resources**: contains required resources (datasheets, software, and tools…)



INFO Unless a specific location is specified, each time a reference is made to some resources in the following pages, the user must refer to this Resources folder.

2 Assignment 1: Develop a Basic Application

In this chapter we will have a brief introduction about hardware set up we are going to use and steps for developing a basic application with this hardware. This will be a simple application that displays a string on the OLED display.

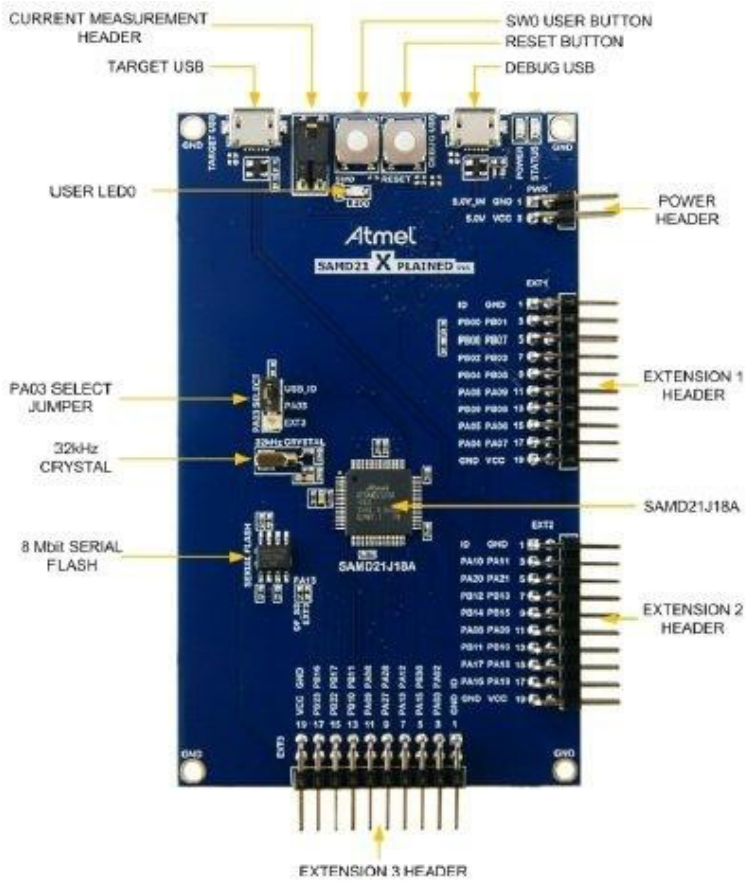
The overview of this chapter is as follows:

- Hardware Setup
- Basic Initialization
- Clock Configuration
- Adding OLED Display drivers
- OLED Initialization and Displaying Text

2.1 Hardware Setup

SAM D21 Xplained Pro evaluation kit will be used in this training session. This is an evaluation kit that allows connecting multiple extension boards. Extension board will normally contain components that can be used for evaluation purpose. An extension board can be connected to any Xplained Pro evaluation kit through any of the external connector present in the board. The SAM D21 Xplained Pro has three external connectors, marked EXT1, EXT2, and EXT3.

Figure 2-1. SAM D21 Xplained Pro



In this training we will use one extension board called OLED1 Xplained Pro. When the extension is to be connected, it will be introduced in the relevant chapter in the training.

A key feature of all Atmel Xplained Pro evaluation kits is the embedded debugger (EDBG). This is a debugger that is populated on the board, making it possible to debug code on the target device without any external hardware. The only connection needed to get started with the any Xplained Pro kit is a Micro-USB cable connected between board and PC. In our case, a Micro-USB cable is connected to the DEBUG USB port on the SAM D21 Xplained pro on one end and a computer running Atmel Studio 6.2 on the other end. The embedded debugger uses a standard interface, CMSIS-DAP (an open debugging interface made by ARM®). This allows not only Atmel, but also third parties to provide debug support for the kit.

To further allow for rapid development on the SAM D21 Xplained Pro it includes both a USB CDC (Virtual COM port) connected to a USART on the target MCU, as well as a data interface for higher speed data communication. This allows for getting data easily out of the target MCU for additional processing or verification on the debugging platform. Examples of this usage can be ADC sample plotting, logging debug data, or simulating external systems during debugging.

2.2 Basic Initialization

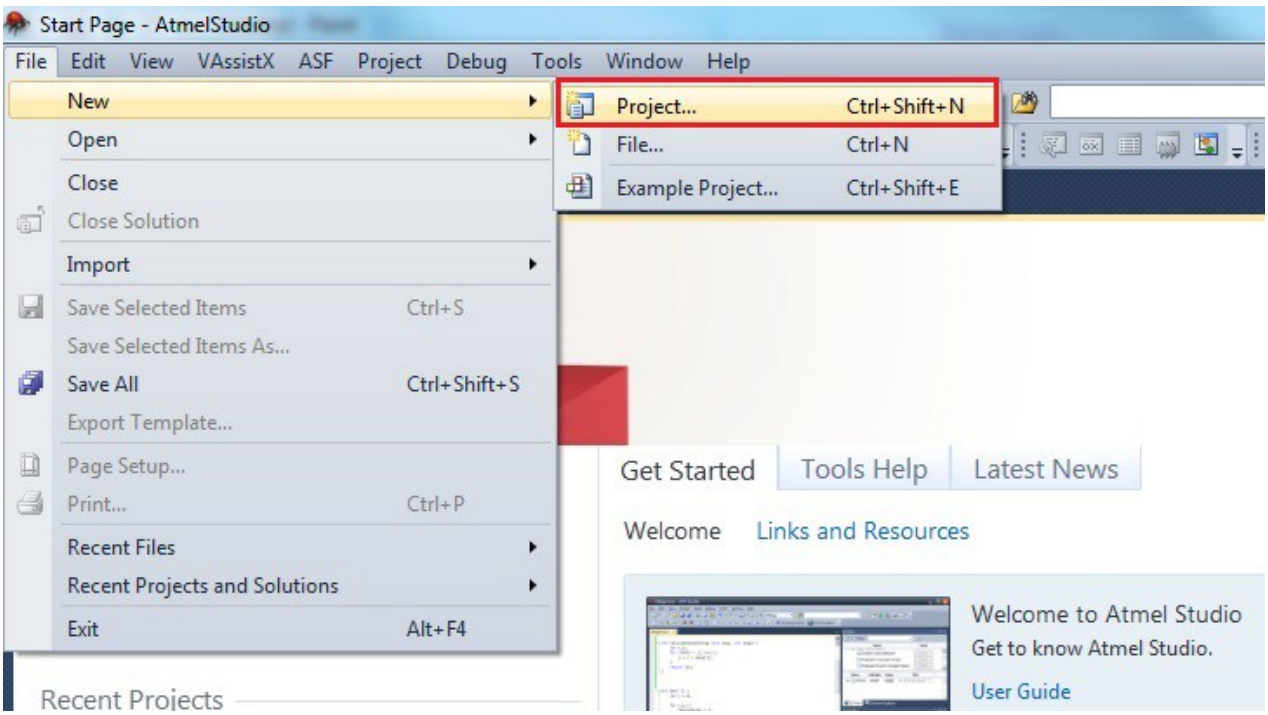
In Atmel Studio 6.2, we need to create a new project for the SAM D21 Xplained Pro kit. With this as base, we can add drivers/services from ASF Wizard as well as the application code itself to make a complete application.

Following steps will guide in creating a new project for SAM D21 Xplained PRO kit.

TO DO Create a new project for the SAM D21 Xplained Pro in Atmel Studio 6.2.

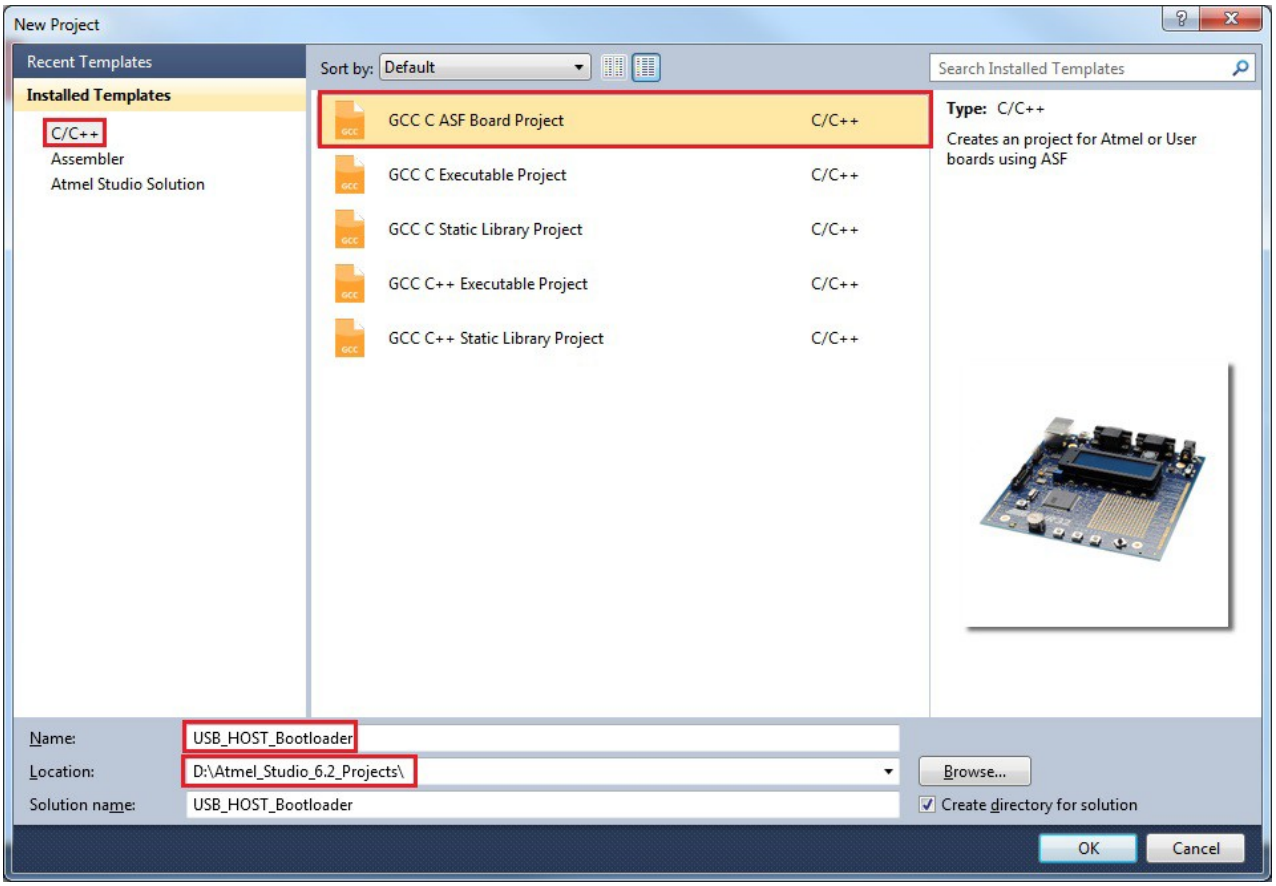
- To create a new project, Launch Atmel Studio and go to `File > New > Project...` menu. This is shown in Figure 2-2 and Figure 2-3:

Figure 2-2. New Project



- Select *GCC C ASF Board Project*

Figure 2-3. ASF Board

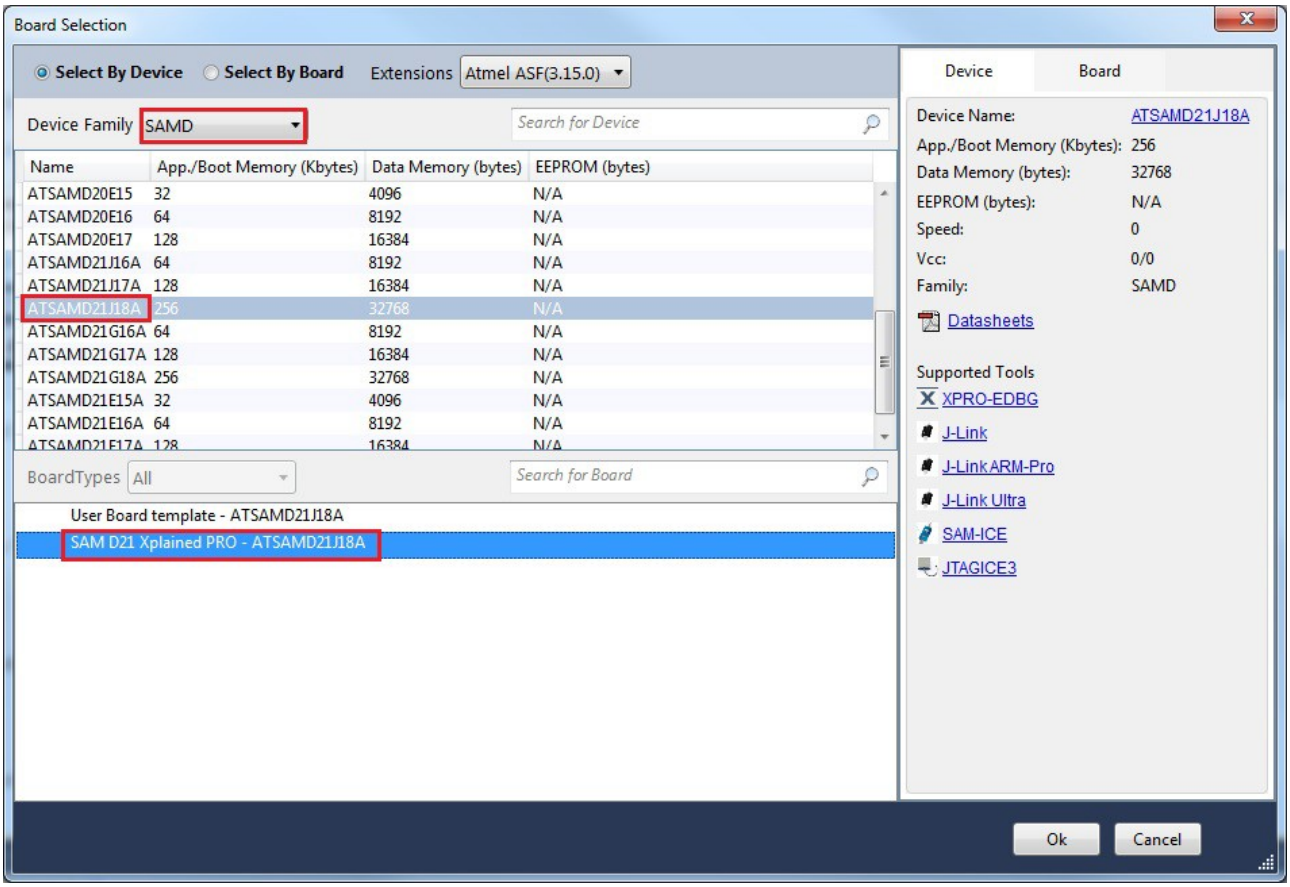


Give the project a name, for example "USB_HOST_Bootloader" and press “OK”.

TIPS We can select the location of the project by selecting a specific Folder in “Location” Tab. It is always recommended to have short path for project location.

- On the next screen, select the correct device and board with which we are going to work as shown in Figure 2-4

Figure 2-4. Device and Board Selection



Select “Select By Device”, (limit search to SAMD in device family) and select ATSAMD21J18A device. Select “SAM D21 Xplained Pro” in the Board types given below. Finally click on OK to create the new project.

The new project by default has a minimal application that will turn on or off the LED in SAM D21 Xplained Pro based on the state of the SW0 button. Pressing the SW0 button will turn the LED on, and releasing the button will turn the LED off. To verify that the SAM D21 Xplained Pro is connected correctly we will run this application and check that it produces the expected output.

TO DO Connect the SAM D21 Xplained Pro to your computer using the provided Micro-USB cable. The cable should be connected to the DEBUG USB port on the SAM D21 Xplained Pro kit.

WARNING Make sure the USB-cable from the computer is connected to the DEBUG USB port. Refer Figure 2-1 SAM D21 Xplained Pro on page 5.

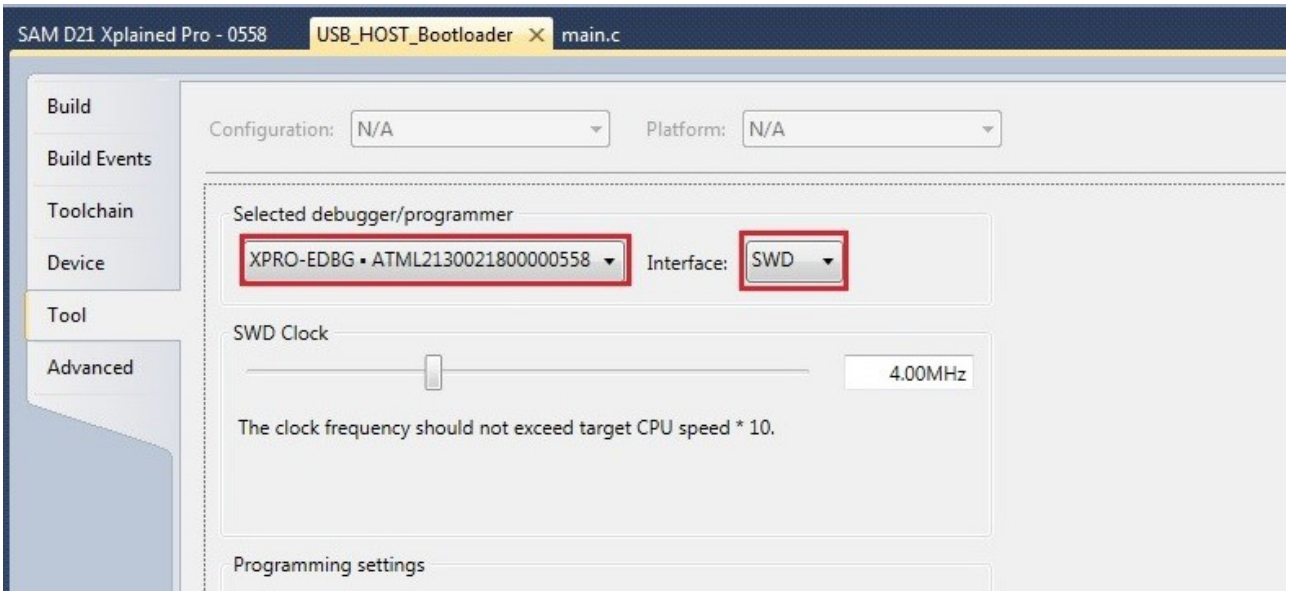
The drivers for the embedded debugger on the SAM D21 Xplained Pro will be installed automatically when connecting the board.

TIPS Ensure that the drivers for this EDBG are installed successfully before running the application. On successful installation of drivers, SAM D21 Xplained Pro page will be opened in Atmel Studio by default.

TO DO Configure the Tool and Interface in the Project properties.

Select the Project in solution explorer -> Right click and select "Properties" -> Click on the Tool Tab and select the correct the Debugger and Interface. EDBG will be listed in the drop-down here and SWD is the interface should be selected. Figure 2-5 shows this configuration.

Figure 2-5. Select Tool



TO DO Run the application.

To program and execute the application, we have two options; either we can start a debug session on the board, where we will be able to break and follow the application flow, or we can simply program the compiled code to the controller and execute the application. In this case we will simply program the code with no debugging, so we select the green arrow for "Start Without Debugging".

Figure 2-6. Start Without Debugging

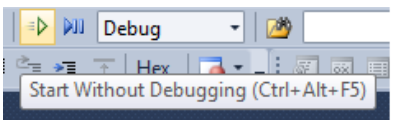
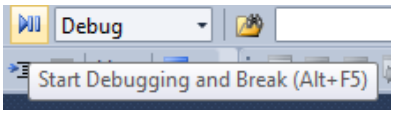


Figure 2-7. Start Debugging and Break



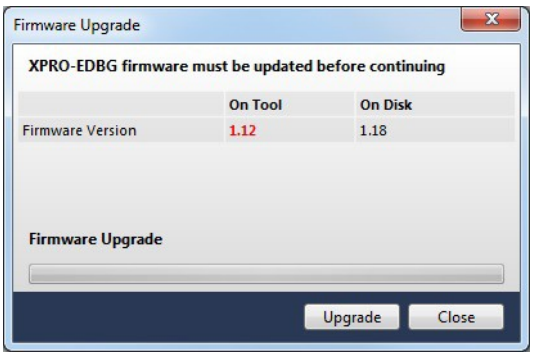
In case if we clicked “Start Without Debugging” option without configuring Tool and Interface for a project, Atmel Studio will throw an error saying that user does not configure the correct Tool and interface. Then user has to set the Tool and Interface as we said in the previous step in Figure 2-5 Select Tool.

INFO

If the firmware of embedded debugger is out of date, Atmel Studio will prompt to update the firmware. Press the "Upgrade" button to upgrade the embedded debugger firmware to the latest version. Programming/Debugging may not work properly if the firmware version is not up to date. So it is always recommended to upgrade the firmware whenever Atmel Studio prompts.

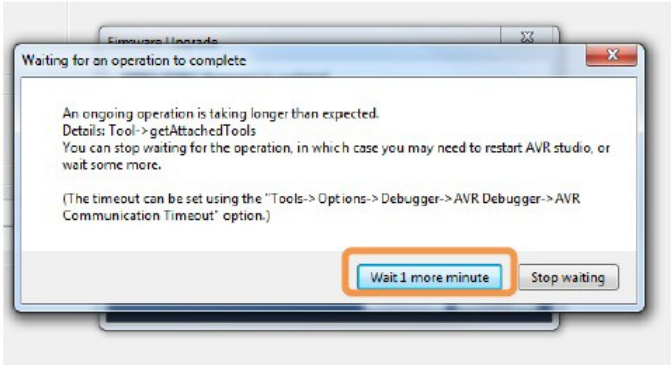
After the device is programmed using the "Start without Debugging" option, we have to press the Reset button to start the application.

Figure 2-8. Firmware Upgrade Prompt



WARNING Upgrade operation may take a few minutes; wait for the operation to complete.

Figure 2-9. Firmware Upgrade Timeout



RESULT After pressing reset button, Application code will be started. If the kit is programmed correctly the LED0 will turn on whenever the SW0 button on the SAM D21 Xplained Pro is pressed.

By default, following is what the main code looks like.

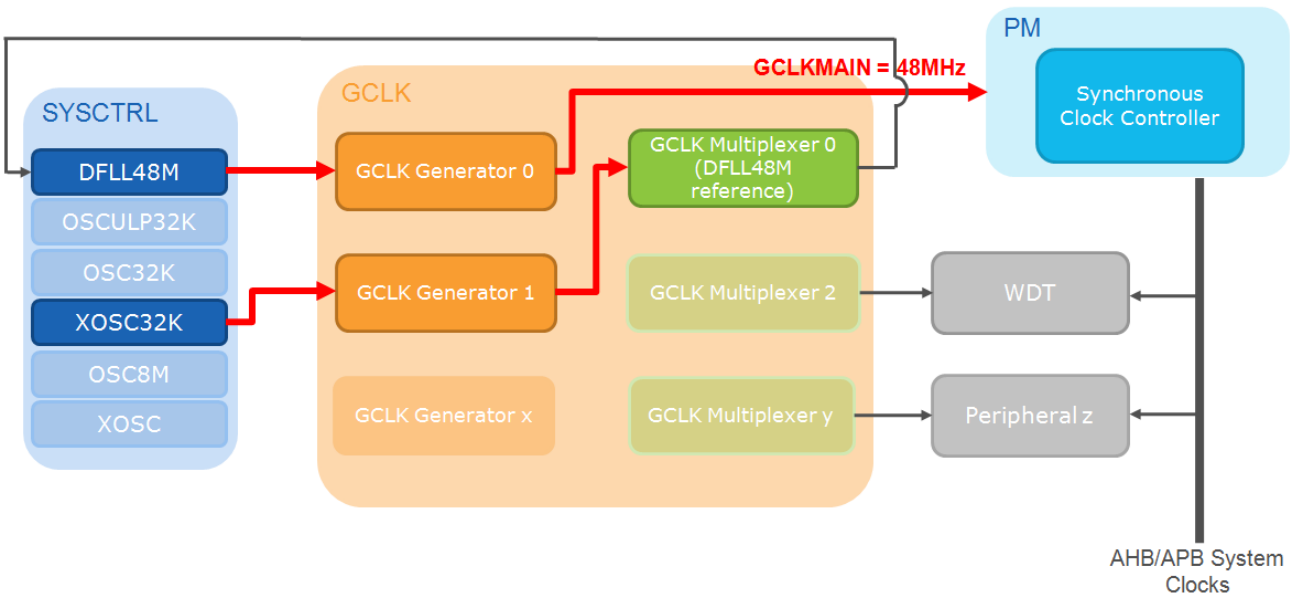
Figure 2-10. Default main() Function

```
/*
 * Include header files for all drivers that have been imported from
 * Atmel Software Framework (ASF).
 */
#include <asf.h>

int main (void)
{
    system_init();

    // Insert application code here, after the board has been initialized.

    // This skeleton code simply sets the LED to the state of the button.
    while (1) {
        // Is button pressed?
        if (port_pin_get_input_level(BUTTON_0_PIN) == BUTTON_0_ACTIVE) {
            // Yes, so turn LED on.
            port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
        } else {
            // No, so turn LED off.
            port_pin_set_output_level(LED_0_PIN, !LED_0_ACTIVE);
        }
    }
}
```

The function system_init() initializes any board specific hardware (such as buttons and LEDs) and sets up the clock system according to the configuration file conf_clocks.h. By default the CPU will be running at 8MHz from the internal RC oscillator after system_init() has been executed. Inside the while(1) loop, Button0 (SW0) status is checked and LED0 is turned on if SW0 is pressed and turned off if LED0 is not pressed.

2.3 Clock Configuration

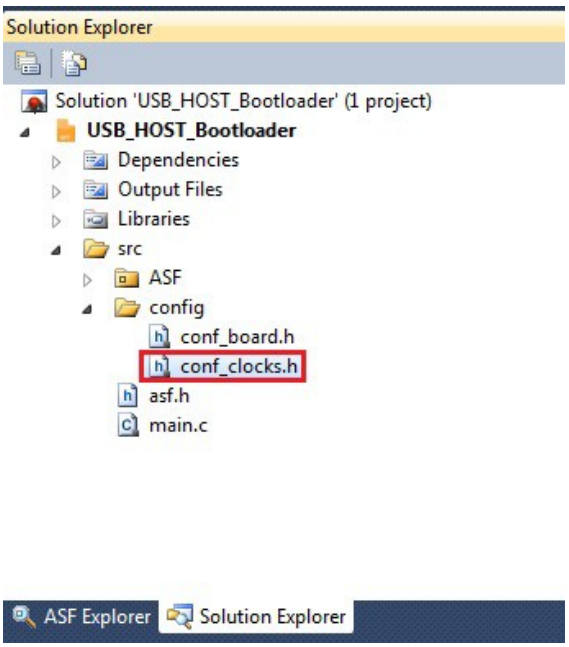
In this project, we are going to use the USB module. To meet the USB standard, a very accurate clock source is needed for the USB module. We will use the external 32kHz crystal oscillator as the reference clock source for the DFLL which in turn will be used as the clock source for both the CPU and the USB clock domains. We will configure the DFLL to output a 48MHz clock signal to support the USB module. The DFLL is configured in closed-loop mode to ensure maximum frequency stability and accuracy. The external 32kHz crystal oscillator present on the SAM D21 Xplained pro kit is used as reference clock source.



TO DO Configure the DFLL to output 48MHz and make this as the main clock by modifying existing clock settings in conf_clocks.h.

- In Solution explorer, configuration files can be found in folder src/config. The conf_clocks.h file holds the includes and defines for all the clock configuration settings of the application.

Figure 2-11. Conf_clocks.h Location in ASF Tree



Modify the existing settings in conf_clocks.h file to match the below: Configuring clocks is easy by stating true/false in the enable define and configuring simple MUL values is enough.

- Enable the 32k oscillator and set it to output a 32kHz clock: CONF_CLOCK_XOSC32K_ENABLE should be defined true

```
/* SYSTEM_CLOCK_SOURCE_XOSC32K configuration - External 32KHz crystal/clock oscillator */
#  define CONF_CLOCK_XOSC32K_ENABLE                  true
#  define CONF_CLOCK_XOSC32K_EXTERNAL_CRYSTAL        SYSTEM_CLOCK_EXTERNAL_CRYSTAL
#  define CONF_CLOCK_XOSC32K_STARTUP_TIME            SYSTEM_XOSC32K_STARTUP_65536
#  define CONF_CLOCK_XOSC32K_AUTO_AMPLITUDE_CONTROL  false
#  define CONF_CLOCK_XOSC32K_ENABLE_1KHZ_OUPUT       false
#  define CONF_CLOCK_XOSC32K_ENABLE_32KHZ_OUTPUT     true
#  define CONF_CLOCK_XOSC32K_ON_DEMAND               true
#  define CONF_CLOCK_XOSC32K_RUN_IN_STANDBY          false
```

- Configure GCLK generator 1 to use the external 32k oscillator as input: ONF_CLOCK_GCLK_1_ENABLE should be defined true CONF_CLOCK_GCLK_1_CLOCK_SOURCE should be defined to SYSTEM_CLOCK_SOURCE_XOSC32K

```
/* Configure GCLK generator 1 */
#  define CONF_CLOCK_GCLK_1_ENABLE                   true
#  define CONF_CLOCK_GCLK_1_RUN_IN_STANDBY           false
#  define CONF_CLOCK_GCLK_1_CLOCK_SOURCE             SYSTEM_CLOCK_SOURCE_XOSC32K
#  define CONF_CLOCK_GCLK_1_PRESCALER                1
#  define CONF_CLOCK_GCLK_1_OUTPUT_ENABLE            false
```

TIPS To know the define values and to find alternate options, select a define and press (ALT + G). This short- cut will help us to know the value of defines.

We are going to multiply the clock from the external 32kHz oscillator with 48000000/32768 to achieve a 48MHz clock.

- Configure the DFLL to use the output from GCLK generator 1 as input and output a 48MHz clock: CONF_CLOCK_DFLL_SOURCE_GCLK_GENERATOR should be defined to GCLK_GENERATOR_1 CONF_CLOCK_DFLL_MULTIPLY_FACTOR should be defined to (48000000/32768).

```
/* DFLL closed loop mode configuration */
#  define CONF_CLOCK_DFLL_SOURCE_GCLK_GENERATOR   GCLK_GENERATOR_1
#  define CONF_CLOCK_DFLL_MULTIPLY_FACTOR         (48000000 / 32768)
#  define CONF_CLOCK_DFLL_QUICK_LOCK              true
#  define CONF_CLOCK_DFLL_TRACK_AFTER_FINE_LOCK   true
#  define CONF_CLOCK_DFLL_KEEP_LOCK_ON_WAKEUP     true
#  define CONF_CLOCK_DFLL_ENABLE_CHILL_CYCLE      true
#  define CONF_CLOCK_DFLL_MAX_COARSE_STEP_SIZE    (0x1f / 4)
#  define CONF_CLOCK_DFLL_MAX_FINE_STEP_SIZE      (0xff / 4)
```

- Enable the DFLL and set it to operate in closed loop mode: CONF_CLOCK_DFLL_ENABLE should be defined true CONF_CLOCK_DFLL_LOOP_MODE should be defined to SYSTEM_CLOCK_DFLL_LOOP_MODE_CLOSED.

```
/* SYSTEM_CLOCK_SOURCE_DFLL configuration - Digital Frequency Locked Loop */
#  define CONF_CLOCK_DFLL_ENABLE                  true
#  define CONF_CLOCK_DFLL_LOOP_MODE               SYSTEM_CLOCK_DFLL_LOOP_MODE_CLOSED
#  define CONF_CLOCK_DFLL_ON_DEMAND               false
```

- Configure GCLK generator 0 (the main clock) to use the DFLL as input: CONF_CLOCK_GCLK_0_ENABLE should be defined true CONF_CLOCK_GCLK_0_CLOCK_SOURCE should be defined to SYSTEM_CLOCK_SOURCE_DFLL.

```
/* Configure GCLK generator 0 (Main Clock)
#  define CONF_CLOCK_GCLK_0_ENABLE                true
#  define CONF_CLOCK_GCLK_0_RUN_IN_STANDBY        false
#  define CONF_CLOCK_GCLK_0_CLOCK_SOURCE          SYSTEM_CLOCK_SOURCE_DFLL
#  define CONF_CLOCK_GCLK_0_PRESCALER             1
#  define CONF_CLOCK_GCLK_0_OUTPUT_ENABLE         false
```

- Set the CPU to use 1 wait state while reading from flash since we are using the 48MHz clock. NVM Characteristics in device datasheet has this Wait states values for various CPU clock speed. CONF_CLOCK_FLASH_WAIT_STATES should be defined to 1.

```
/* System clock bus configuration */
#  define CONF_CLOCK_CPU_CLOCK_FAILURE_DETECT     false
#  define CONF_CLOCK_FLASH_WAIT_STATES            1
#  define CONF_CLOCK_CPU_DIVIDER                  SYSTEM_MAIN_CLOCK_DIV_1
#  define CONF_CLOCK_A PBA_DIVIDER                SYSTEM_MAIN_CLOCK_DIV_1
#  define CONF_CLOCK_APBB_DIVIDER                 SYSTEM_MAIN_CLOCK_DIV_1
```

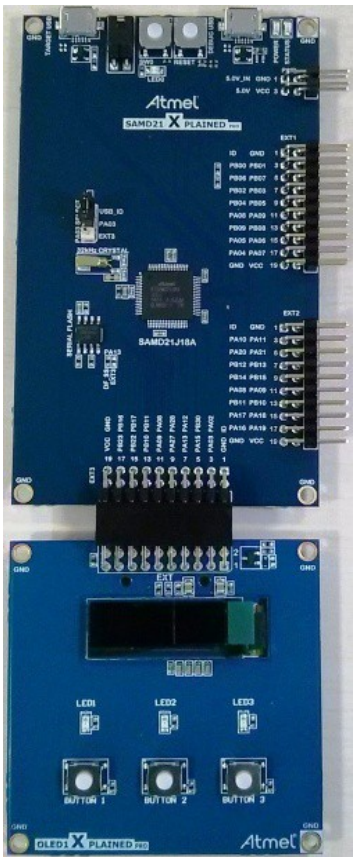
2.4 Adding OLED Display Drivers

In this section we are going to add an OLED driver with string support and print some message on OLED screen on the OLED1 Xplained Pro expansion wing. This will be used later to display messages for user information when the bootloader is running.

TO DO

Start by making sure that the OLED1 is connected to the EXT3 header on the Xplained PRO as shown in Figure 2-12:

Figure 2-12. OLED1 Xplained Wing Connected to EXT13



TO DO

Add the GFX Monochrome - System Font service using the ASF Wizard and configure the driver to connect to the display as shown in Figure 2-13, Figure 2-14, and Figure 2-15.

The following settings will set up the correct graphic controller driver and interface used to connect to the OLED.

Figure 2-13. ASF Wizard

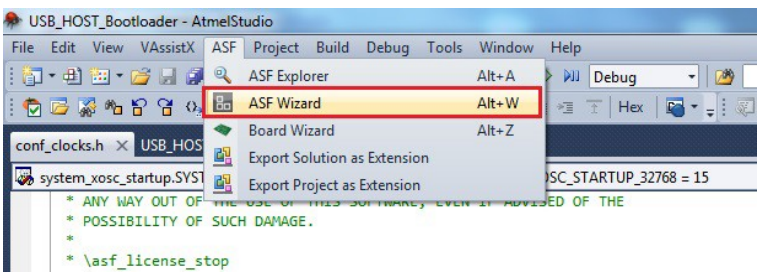


Figure 2-14. GFX Monochrome Service

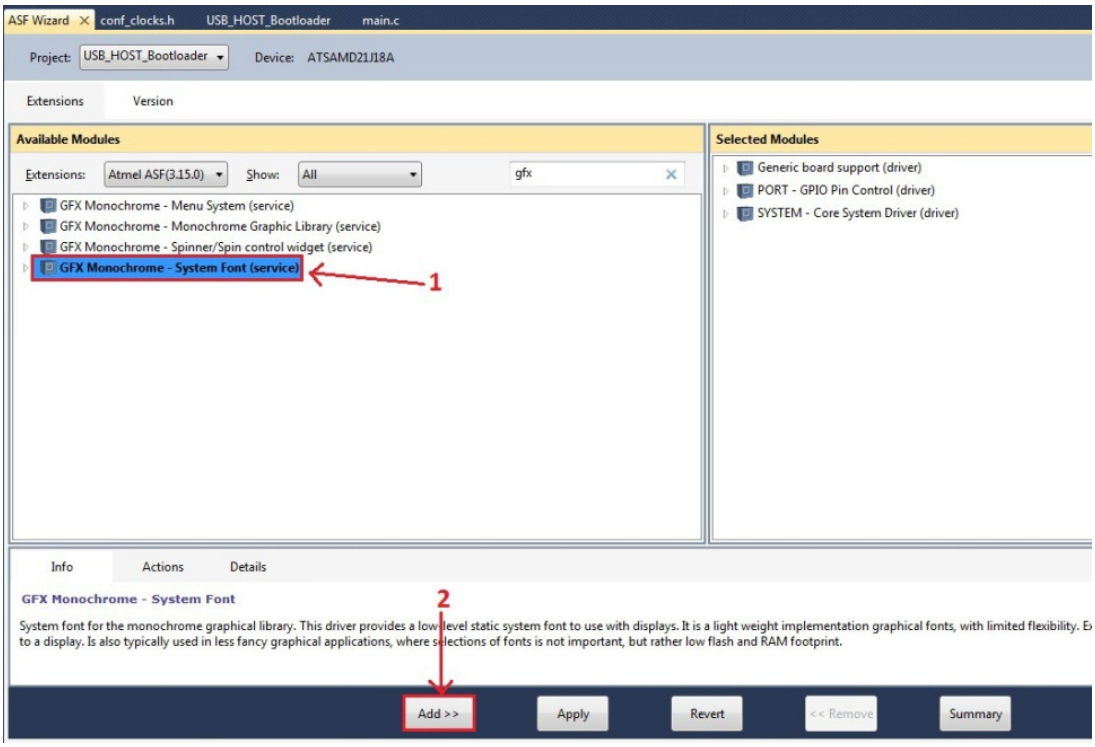


Figure 2-15. GFX Monochrome Display Driver Configuration

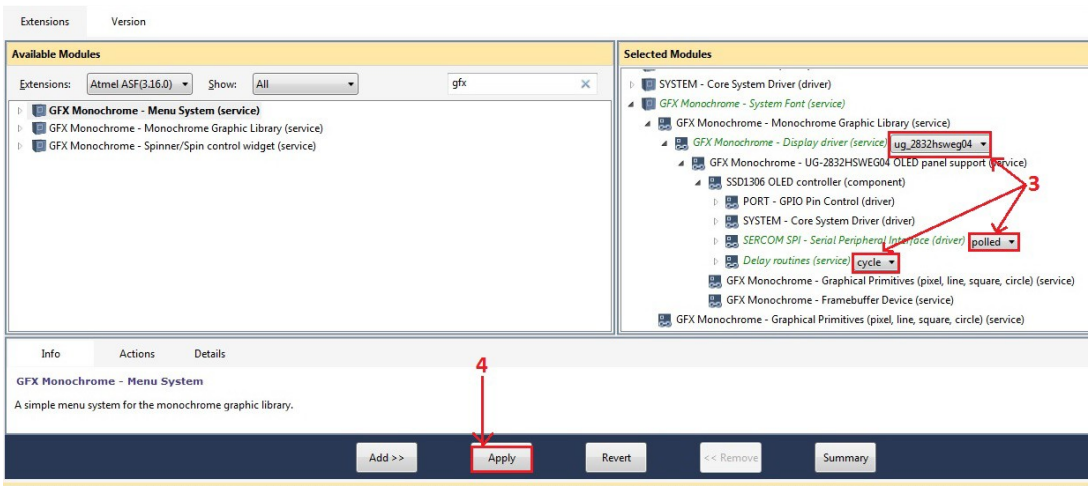
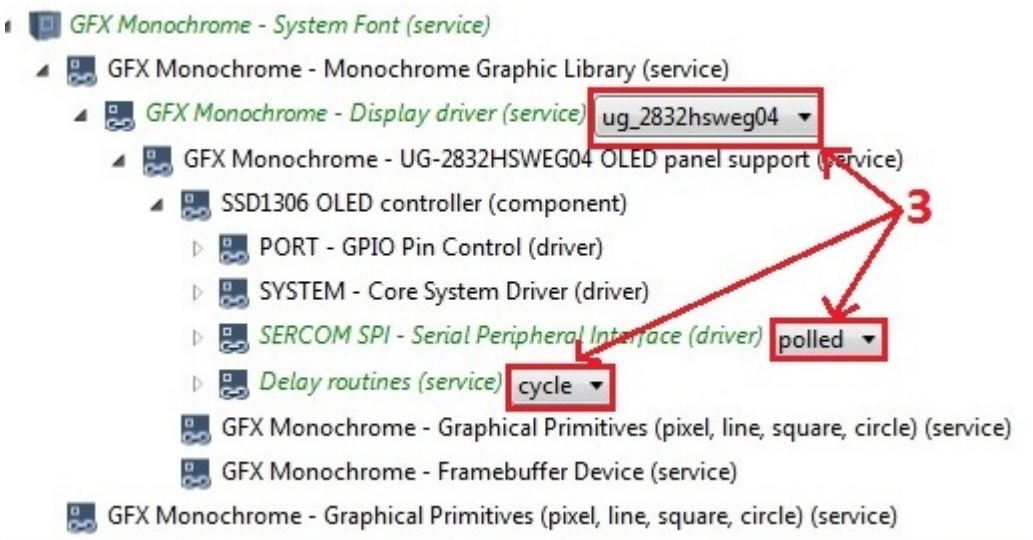


Figure 2-16. Display Driver Configurations



Once the driver has been added using the ASF Wizard, three configuration files have been added to the project under the folder src/config. The three files are named:

- conf_spi.h - Contains SPI enable and Timeout defines
- conf_ssd1306.h - Includes defines to set up the SPI interface with the screen correctly
- conf_sysfont.h - Contains bitmaps of the text font

TIPS Quick start guide and comments in driver will help in getting started with OLED display.

Comments in driver file (gfx_mono_text.h) contain code snippets to display a simple string. A quick start guide is also available on this which is really close to what we would like to do for our application. Quick Start Guide: http://asf.atmel.com/docs/3.15.0/samd21/html/asfdoc_common2_gfx_mono_font_quickstart.html.

2.5 OLED Initialization and Displaying Text

From the comments as well as from the quick start guide we can see that it is fairly easy to start using the font library for the display. The function system_init() is already included in our code, so the next logical step would be to initialize the screen. As seen in the quick start guide, this is done by:

```
gfx_mono_init();
```

This will set up the SPI interface to communicate with the OLED screen and wipe anything currently shown.

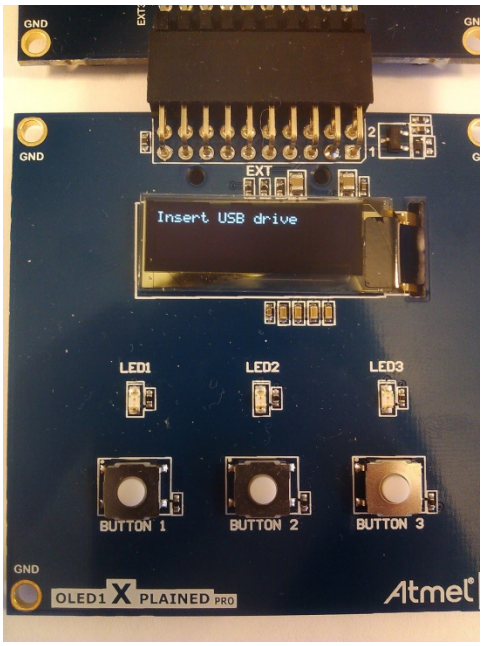
Now that the screen is initialized, it's ready to receive a string. Again we turn to the quick start guide to find out how this is done. One can see that what is needed in order to print "Insert USB Drive" on the screen is:

```
gfx_mono_draw_string("Insert USB drive", 0, 0, &sysfont);
```

TO DO After adding the above routines before the while(1), Save and Run the application.

If gfx_mono_init() and gfx_mono_draw_string() has been inserted before the while loop in the application, see Figure 2-17 for when the device is programmed:

Figure 2-17. OLED1 Xplained



RESULT Now the main function should look as shown in Figure 2-18. If required, solution project (SAMD21_USB_BOOT_ASSIGN1) is available in Resources folder.

Figure 2-18. Main Function

```
 */
#include <asf.h>

int main(void)
{

    system_init();

    //! [init_gfx]
    gfx_mono_init();
    //! [init_gfx]

    //! [drawstring_gfx]
    gfx_mono_draw_string("Insert USB drive", 0, 0, &sysfont);
    //! [drawstring_gfx]

    while (1) {
        // Is button pressed?
        if (port_pin_get_input_level(BUTTON_0_PIN) == BUTTON_0_ACTIVE) {
            //Yes, so turn LED on.
            port_pin_set_output_level(LED_0_PIN,LED_0_ACTIVE);
        } else {
            // No, so turn LED off.
            port_pin_set_output_level(LED_0_PIN,!LED_0_ACTIVE);
        }

    }
}
```

3 Assignment 2: Adding USB and File System Services

In this section we are going to add a USB host service and FAT File system service. Then set up the program to read contents from a file in a connected USB mass storage device and display the contents of the file on the OLED display.

The overview of this chapter is:

- Adding USB driver
- Accessing files
- Displaying the contents of a file

3.1 Adding USB Driver

TO DO In the ASF Wizard, add the USB Host service to the project with the settings displayed in Figure 3-1.

Figure 3-1. USB Host (Service) – Mass Storage Class (MSC)

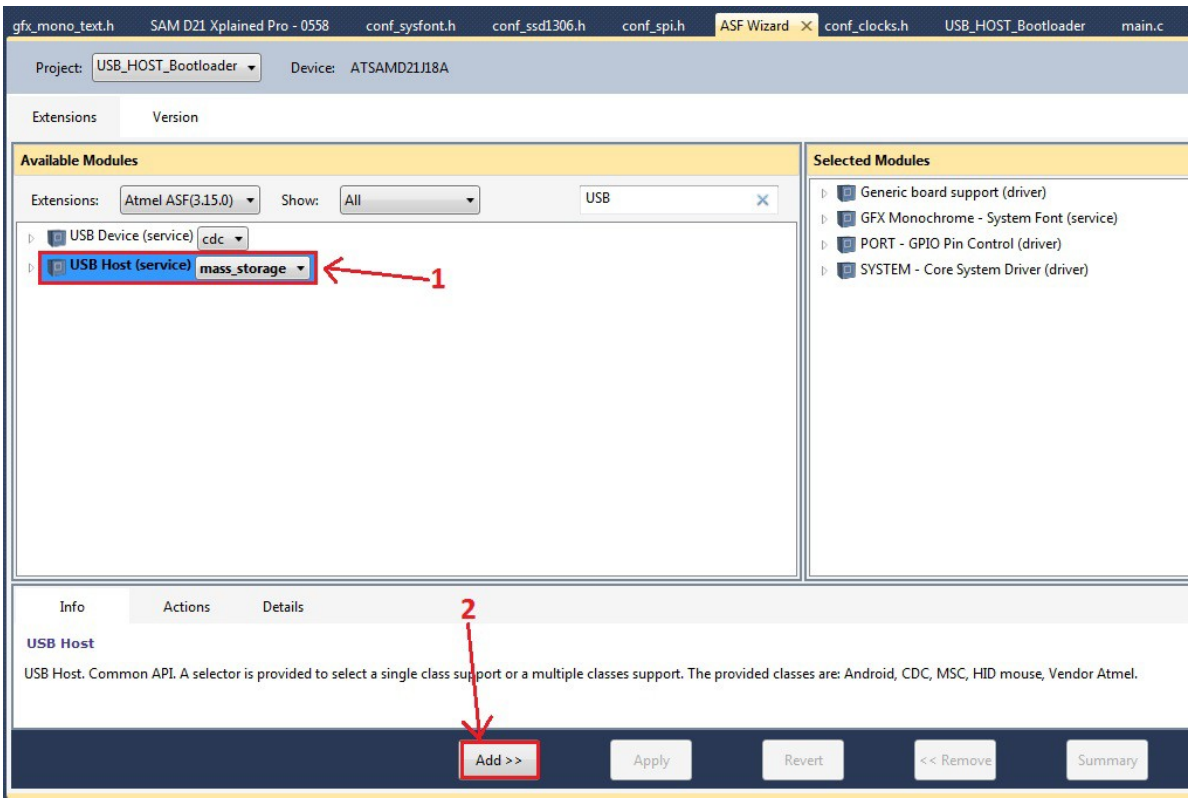
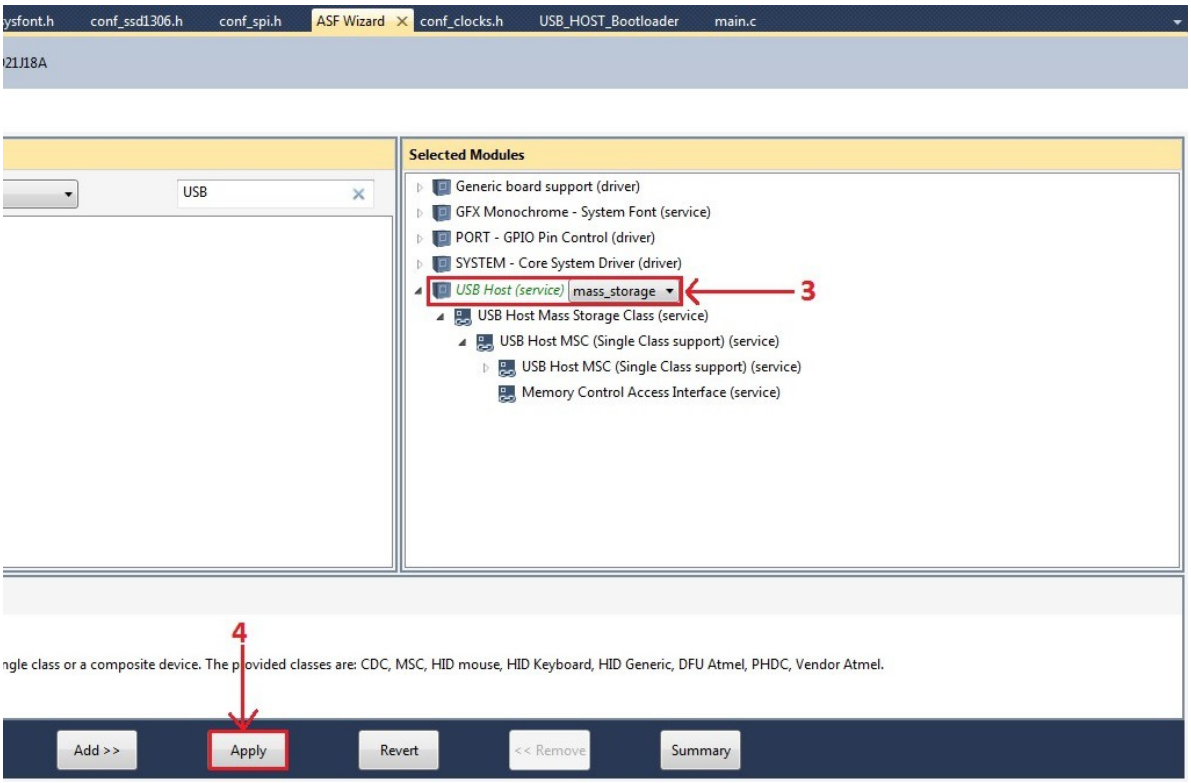


Figure 3-2. USB Host MSC (Service) – Configuration



Once the driver has been added using the ASF Wizard, some configuration files have been added to the project under the folder src/config. These two are of main interest to us:

- conf_usb_host.h - Configures the USB and sets callbacks
- conf_access.h - Configures the abstraction layer for memory interfaces

To enable the USB Host LUN and its APIs, couple of defines needs to be added in symbols section in compiler settings in project properties. Also we have to comment a line in USB LUNs Definitions in conf_access.h file to avoid compiler errors. This is a bug and this line should be removed from original file to avoid the compilation error in future.

TO DO Add the symbols in compiler settings.

- Right click the Project and select Properties -> Select Toolchain tab -> ARM/GNU C Compiler -> Symbols -> Click the icon add item and add "USB_MASS_STORAGE_ENABLE=true"
- After adding this symbol, add one more symbol "ACCESS_MEM_TO_RAM_ENABLED=true"

RESULT The symbols in compiler settings are now added with define for USB accesses. Figure 3-3 is the reference.

Figure 3-3. Compiler Settings – Symbols

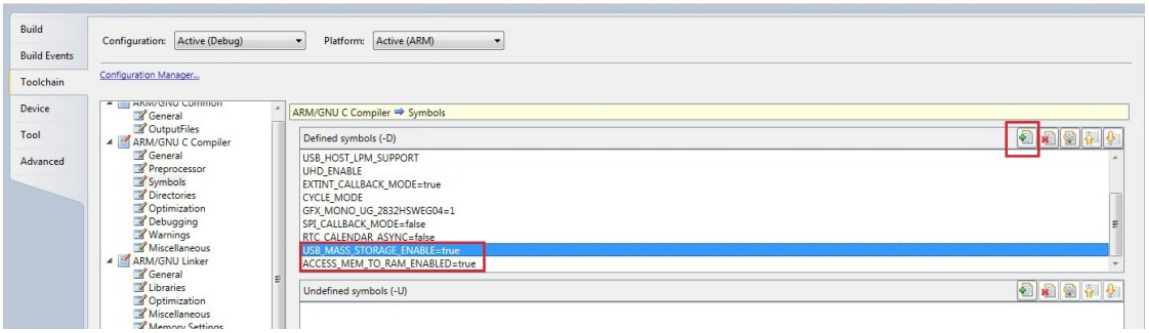
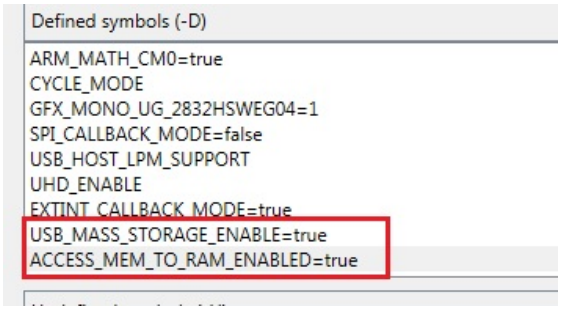


Figure 3-4. Symbols



TO DO Comment the Lun_usb_unload defines in conf_access file.

- Comment the line "#define Lun_usb_unload - NULL" in USB LUNs Definitions in conf_access file to avoid compilation errors. We can also remove this line if we wish so.

RESULT The corresponding line is now commented conf_access file to avoid compilation errors.

Figure 3-5. Line Should be Commented

```
#define Lun_5_ram_2_mem                    sd_mmc_mci_ram_2_mem_0
#define LUN_5_NAME                         "\"SD/MMC Card over MCI Slot 0\""
//! @}

/*! \name USB LUNs Definitions
 */
//! @{
#define MEM_USB                            LUN_USB
#define LUN_ID_MEM_USB                     LUN_ID_USB
#define LUN_USB_INCLUDE                    "uhi_msc_mem.h"
#define Lun_usb_get_lun()                  uhi_msc_mem_get_lun()
#define Lun_usb_test_unit_ready(lun)       uhi_msc_mem_test_unit_ready(lun)
#define Lun_usb_read_capacity(lun, nb_sect) uhi_msc_mem_read_capacity(lun, nb_sect)
#define Lun_usb_read_sector_size(lun)      uhi_msc_mem_read_sector_size(lun)
//#define Lun_usb_unload                     NULL
#define Lun_usb_wr_protect(lun)            uhi_msc_mem_wr_protect(lun)
#define Lun_usb_removal()                  uhi_msc_mem_removal()
#define Lun_usb_mem_2_ram(addr, ram)       uhi_msc_mem_read_10_ram(addr, ram)
#define Lun_usb_ram_2_mem(addr, ram)       uhi_msc_mem_write_10_ram(addr, ram)
#define LUN_USB_NAME                       "\"Host Mass-Storage Memory\""
//! @}

/*! \name Actions Associated with Memory Accesses
 *
 * Write here the action to associate with each memory access.
 *
 * \warning Be careful not to waste time in order not to disturb the functions.
 */
```

We want to add a callback to the Start of Frame Event. This event is called once every SOF is sent, and since an SOF is sent every 1ms when a USB device is connected, this works as a simple timer. We are going to use this later to measure the time used to update the firmware. A variable is updated inside this callback function and this variable is used to measure time period taken for updating the firmware.

Callback function has to be mentioned in conf_usb_host.h file. This callback function name can be of our wish or we can use the default one. Add the following line in conf_usb_host.h file.

```
#  define UHC_SOF_EVENT()           main_usb_sof_event()
```

We also have to add a prototype for the main_usb_sof_event() at the top of the conf_usb_host.h file, after the #includes, add:

```
void main_usb_sof_event(void);
```

Then declare a variable in main.c file and increment this variable in callback function defined in main.c file (outside of the main-function):

Initialize the global variable somewhere before the main-function and SOF function:

```
volatile static uint16_t main_usb_sof_counter = 0;
```

Implement the callback function after the main() function in main.c file. Increment the SOF counter variable in this function.

```
void main_usb_sof_event(void)
{
        main_usb_sof_counter++;
}
```

3.2 Accessing Files

We will need FAT file system support in the application to be able access a file in the connected mass storage memory on the USB stick. So we are going to add the FAT file system service through ASF wizard.

TO DO Add the FatFS service through the wizard, and configure it as shown in Figure 3-6 and Figure 3-7.

Figure 3-6. FatFS File System (Service)

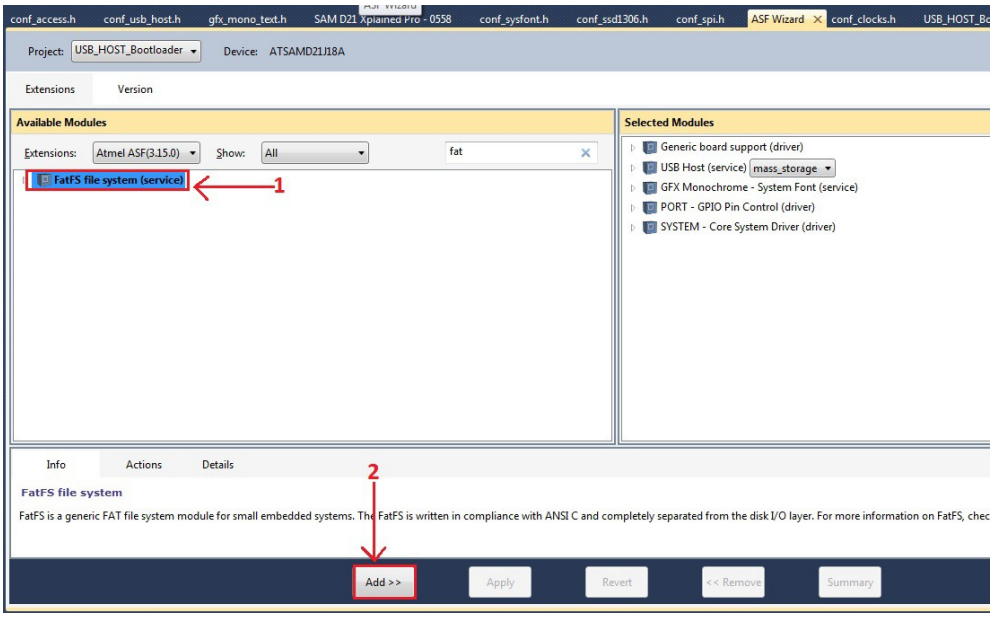
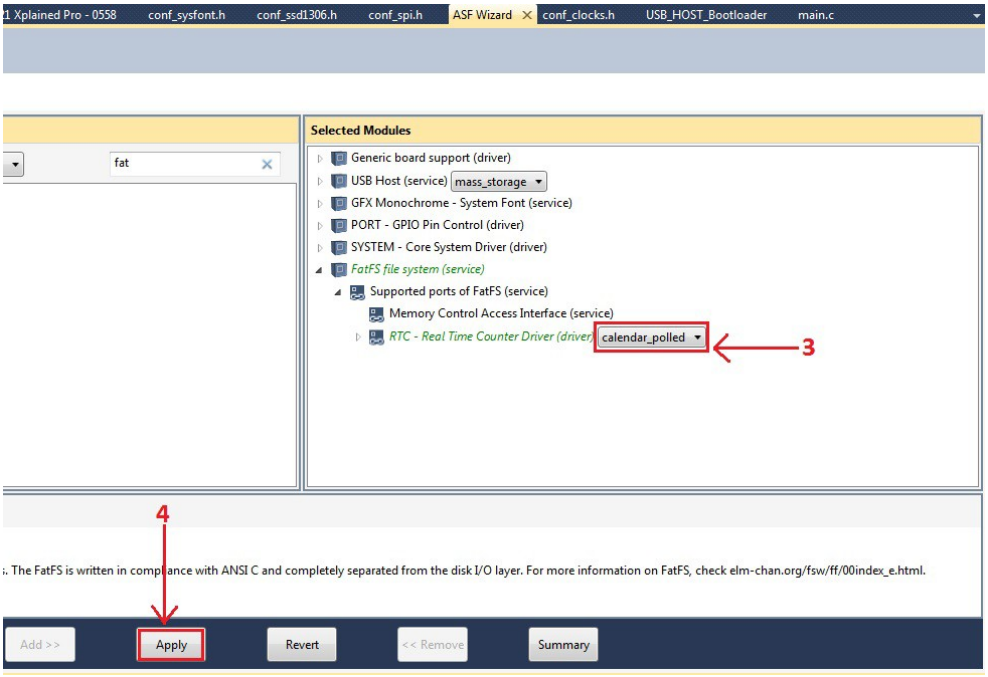


Figure 3-7. FatFS File System (Service) – Configuration



INFO

After clicking "Apply", we have accepted the FatFS File system license agreement in order to add FatFS drivers to the Project.
The asf help of USB Host Mass storage application for SAM D21 Xplained PRO board will help us to start with MSC with FAT File system. Main.c file in following ASF help link would help us better.
http://asf.atmel.com/docs/latest/common.services.usb.class.msc.host.example2.samd21_xplained_pro/html/index.html

To be able to access files on a connected USB mass storage device, we have to add some variables and definitions to the main file. We want to add this code at the top of the file after the #includes:

```
#include "string.h"
#define MAX_DRIVE       _VOLUMES
#define FIRMWARE_FILE "firmware.txt"
const char firmware_filename[] = {FIRMWARE_FILE};
/* FATFS variables */
static FATFS fs;
static FIL file_object;
```

And add this to the main-function before the while(1) loop. This function is to start the USB Host communication:

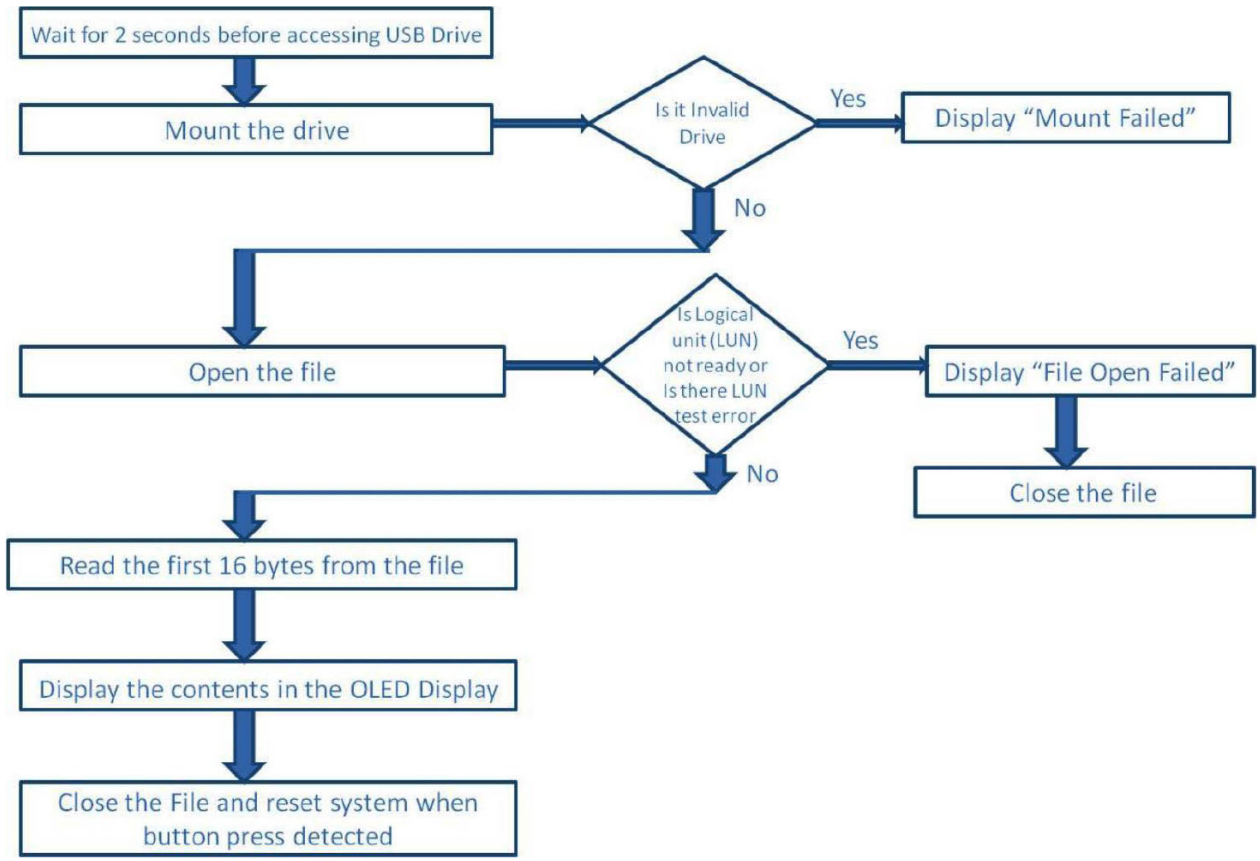
```
uhc_start();
```

3.3 Displaying the Contents of a File

TO DO Replace the existing while(1) loop in your main function with this:

- Flow chart will help in understanding the code flow. Replace the existing while(1) loop with the snippet after the flowchart.

Figure 3-8. Display Flowchart



Code snippet:

```
	while (true) {

		/* Wait 2 seconds before trying to access the USB drive */
		if (main_usb_sof_counter > 2000) {
			main_usb_sof_counter = 0;
			volatile uint8_t lun = LUN_ID_USB;

			/* Mount drive */
			memset(&fs, 0, sizeof(FATFS));

			FRESULT res = f_mount(lun, &fs);
			if (FR_INVALID_DRIVE == res) {
				gfx_mono_draw_string("Mount Failed!", 0, 0, &sysfont);
				continue;
			}

			res = f_open(&file_object,firmware_filename, FA_READ);
			if (res == FR_NOT_READY) {
				/* LUN not ready */
				gfx_mono_draw_string("File open failed!", 0, 0, &sysfont);

				f_close(&file_object);
				continue;
			}

			if (res != FR_OK) {
				/* LUN test error */
				f_close(&file_object);
				gfx_mono_draw_string("File open failed!", 0, 0, &sysfont);
				continue;
			}

			/* Get size of file */
			uint32_t fw_size = f_size(&file_object);

			uint8_t char_buffer[16];
			/* Read the first 16 bytes from USB stick into char_buffer*/
			f_read(&file_object, char_buffer, 16, NULL );
			/* Clear display and print content of file */
			gfx_mono_draw_string("", 0, 0, &sysfont);
			gfx_mono_draw_string(char_buffer, 0, 0, &sysfont);

			f_close(&file_object);

			/* Wait until push button is pressed and then reset the device */
			while (port_pin_get_input_level(BUTTON_0_PIN) != BUTTON_0_ACTIVE) {};
			NVIC_SystemReset();
		}
	}
```

When a USB device is connected main_usb_sof_counter is incremented once for each start of frame. The software waits until a device has been connected for two seconds. Then it tries to read the file firmware.txt in the root folder of the USB device, if found it first clears the display, and then displays the 16 first characters from this file to the OLED display. After that it waits until the user presses SW0 button.

TO DO

Find an empty Fat32 formatted USB thumb-drive. Create the file "firmware.txt" in the root folder of this thumb-drive, and fill it with some text. Compile the project abd Run the application and connect the thumb-drive when prompted.

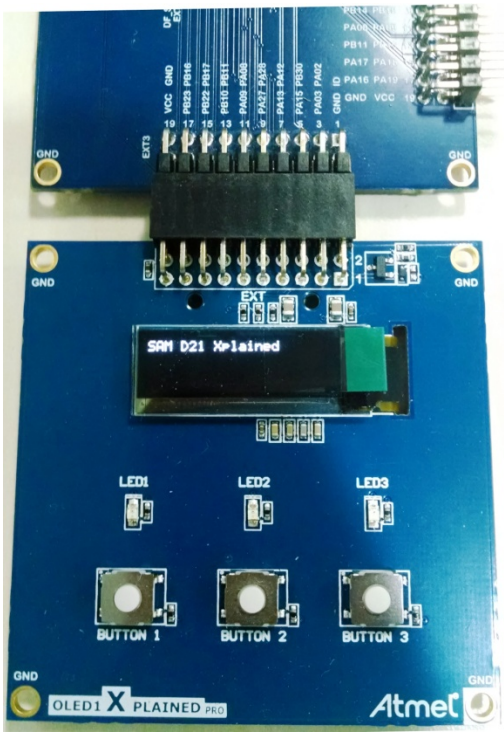
WARNING

Make sure that you connect the USB memory stick to the TARGET USB-port on the SAM D21 Xplained Pro and not to the DEBUG USB-port. Keep the Thumb-drive unconnected to TARGET USB PORT and connect it only after the device prompts.

Figure 3-9. Complete Setup

RESULT The first 16 characters of the file firmware.txt will be displayed on the OLED display. If required, solution project (SAMD21_USB_BOOT_ASSIGN2) is available in Resources folder.

Figure 3-10. Displaying the First 16 Characters



4 Assignment 3: Adding the Bootloader

Now we have to add the bootloader part to the program. The bootloader will run on startup from reset as it will be placed at the start of the program memory. The bootloader code will first check if there is a valid firmware present in the application part of the program memory and that the SW0 button is not pressed. If this is the case then the bootloader will transfer control to the application firmware.

If either the button is pressed or no valid firmware is found in the application part of program memory then the bootloader code will wait until an USB mass storage device is connected to the Target USB connector. When a USB mass storage device is connected, the firmware will scan this device for a file with the correct name. The content of any matching file will be loaded by into the MCU at a predefined position in the program memory after the bootloader.

This way, bootloader and application space will not be overlapped (the size of the bootloader should be monitored so that we could set the position inside the program memory where we want the main application to start).

The overview of this chapter is as follows:

- Updating the flash
- To enter Bootloader or Application mode on starting

4.1 Updating the Flash

We have the basic framework to read a file from a connected USB mass storage device. What we are missing is a way to write this file into the flash without writing over the bootloader, and a way to run the firmware if a valid firmware is present or run the bootloader if either the user wishes to update it or if no valid firmware is present.

We will call our firmware-file "firmware.bin", so we should change the string in main.c accordingly: change the “firmware.txt” to “firmware.bin”.

TO DO Modify the firmware file name (In main.c file) correctly to read the bin file.

```
#define FIRMWARE_FILE "firmware.bin"
```

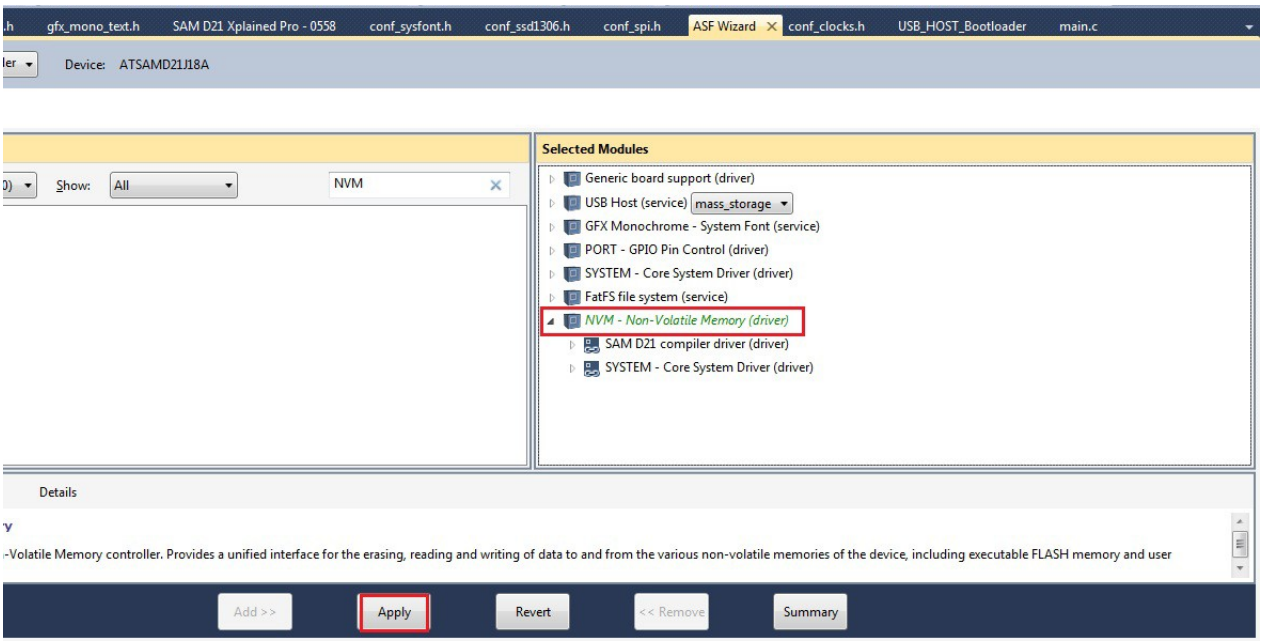
Then we need to define where our new firmware will be loaded, this must be somewhere after the bootloader code ends. Hence we do not overwrite the bootloader part. Application firmware should be located at the start of a flash- page, so we can concentrate on whole pages while writing the flash.

The Flash space we allotted for the bootloader section is 200 NVM Rows. This value will be configured in main file in upcoming steps. The value of 200 has been chosen for this specific implementation on this specific chip and should be adjusted if the size of the bootloader changes or a new chip with a different page-size is chosen. We will also set up a page-buffer. Later we will read the firmware from the USB to the page-buffer and write it into flash, one page at a time.

We need to add the Non-Volatile Memory driver in order to write to the flash of the chip. Go to the ASF wizard and add the NVM driver. There should be no configuration required in order to get this driver to work.

TO DO Add the NVM Driver through ASF Wizard.

Figure 4-1. Adding NVM Driver



TO DO Add the following lines at the start of main.c (after the #include in the top).

```
#define APP_START_ADDRESS (NVMCTRL_ROW_SIZE * 200)
uint8_t page_buffer[NVMCTRL_PAGE_SIZE];
```

We are going to write to the flash, so we should make sure that the non volatile memory controller is initialized with the correct settings. Add initialization lines after "system_init()" in main.c.

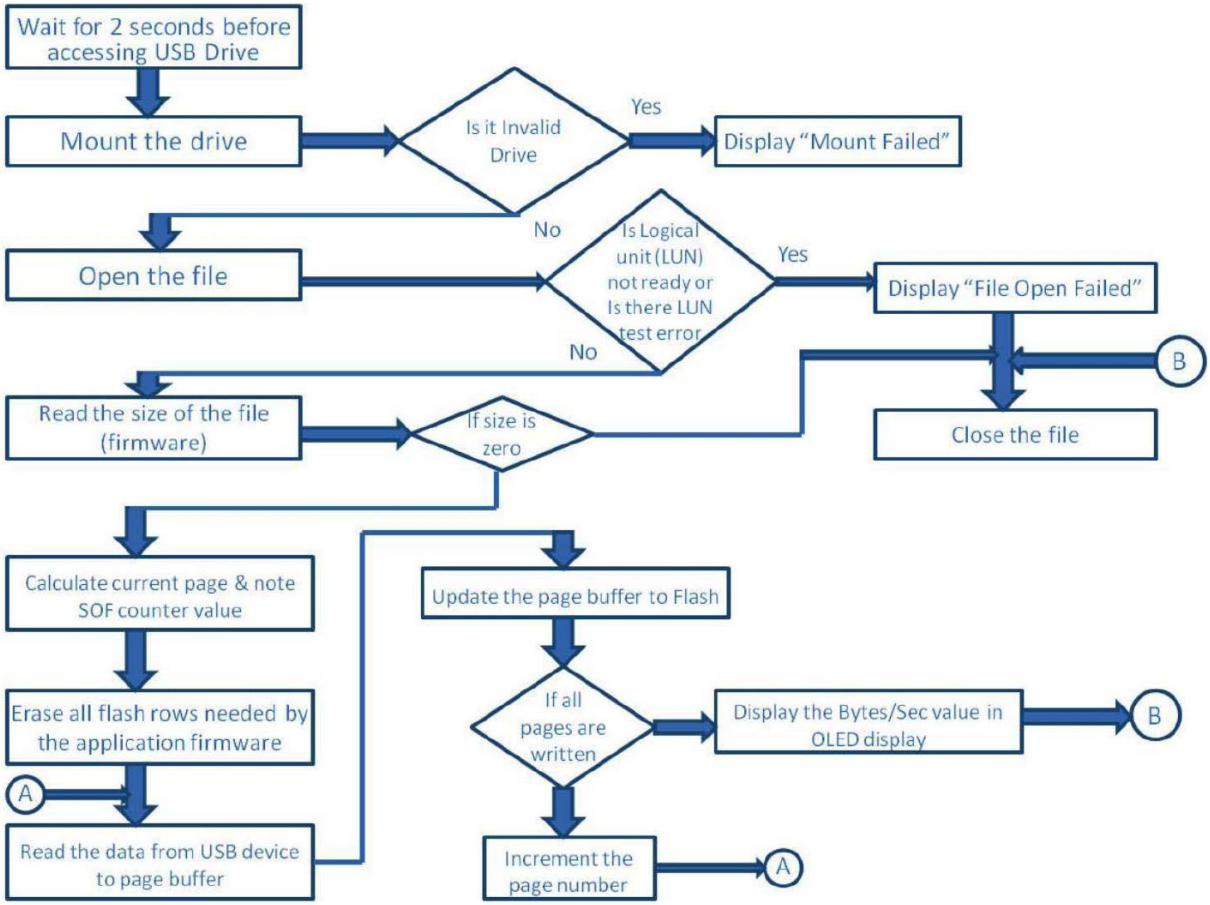
```
	struct nvm_config nvm_cfg;
	nvm_get_config_defaults(&nvm_cfg);
	nvm_set_config(&nvm_cfg);
```

We are now going to add the code that does the writing to the flash from the USB, replace the following snippet in main.c.

```
	uint8_t char_buffer[16];
	/* Read the first 16 bytes from USB stick into char_buffer*/
	f_read(&file_object, char_buffer, 16, NULL );
	/* Clear display and print content of file */
	gfx_mono_draw_string("", 0, 0, &sysfont);
	gfx_mono_draw_string(char_buffer, 0, 0, &sysfont);
```

with the following snippet (code flow is explained in the flowchart).

Figure 4-2. Bootloader Flowchart



Code snippet:

```
        UINT bytes_read = 0;
        enum status_code error_code;
        char str[70];
        if (fw_size != 0) {
               uint32_t current_page = APP_START_ADDRESS /
                            NVMCTRL_PAGE_SIZE;
               uint32_t curr_address = 0;

               /* Store start time to use in b/s calculation */
               uint32_t start_time = main_usb_sof_counter;

               /* Erase flash rows to fit new firmware */
               uint16_t rows_clear = fw_size / NVMCTRL_ROW_SIZE;
               uint16_t i;
               for (i = 0; i < rows_clear; i++) {
                      do {
                             error_code = nvm_erase_row(
                                                    (APP_START_ADDRESS) +
                                                    (NVMCTRL_ROW_SIZE * i));
                      } while (error_code == STATUS_BUSY);
               }

               do {
                      /* Read data from USB stick to the page buffer */
                      f_read(&file_object,
                             page_buffer,
                             NVMCTRL_PAGE_SIZE,
                             &bytes_read );
                      curr_address += bytes_read;

                      /* Write page buffer to flash */
                      do {
                             error_code = nvm_write_buffer(
                                                    current_page *
                                                    NVMCTRL_PAGE_SIZE,
                                                    page_buffer,
                                                    bytes_read);
                      } while (error_code == STATUS_BUSY);
                      current_page++;
               } while (curr_address < fw_size);

               /* Store end time of operation and calculate delta value */
               uint32_t done_time = main_usb_sof_counter;
               done_time -= start_time;

               /* Calculate bytes/s */
               start_time = (fw_size * 1000) / done_time;

               /* Clear display and print summary to display */
               gfx_mono_draw_string("                ", 0, 0, &sysfont);
               sprintf(str, "Written %u bytes\n%u Bytes/s!",
                             (unsigned int)fw_size,
                             (unsigned int)start_time);
               gfx_mono_draw_string(str, 0, 0, &sysfont);
        }
```

This code reads one page at a time from the file firmware.bin on a connected USB mass storage device to the buffer in memory. It then writes this buffer to flash, repeating until the whole file is written to flash.

RESULT We now have the ability to detect if a USB mass storage device is connected, see if it contains a firmware-file and to write the firmware without overwriting the code doing the writing.

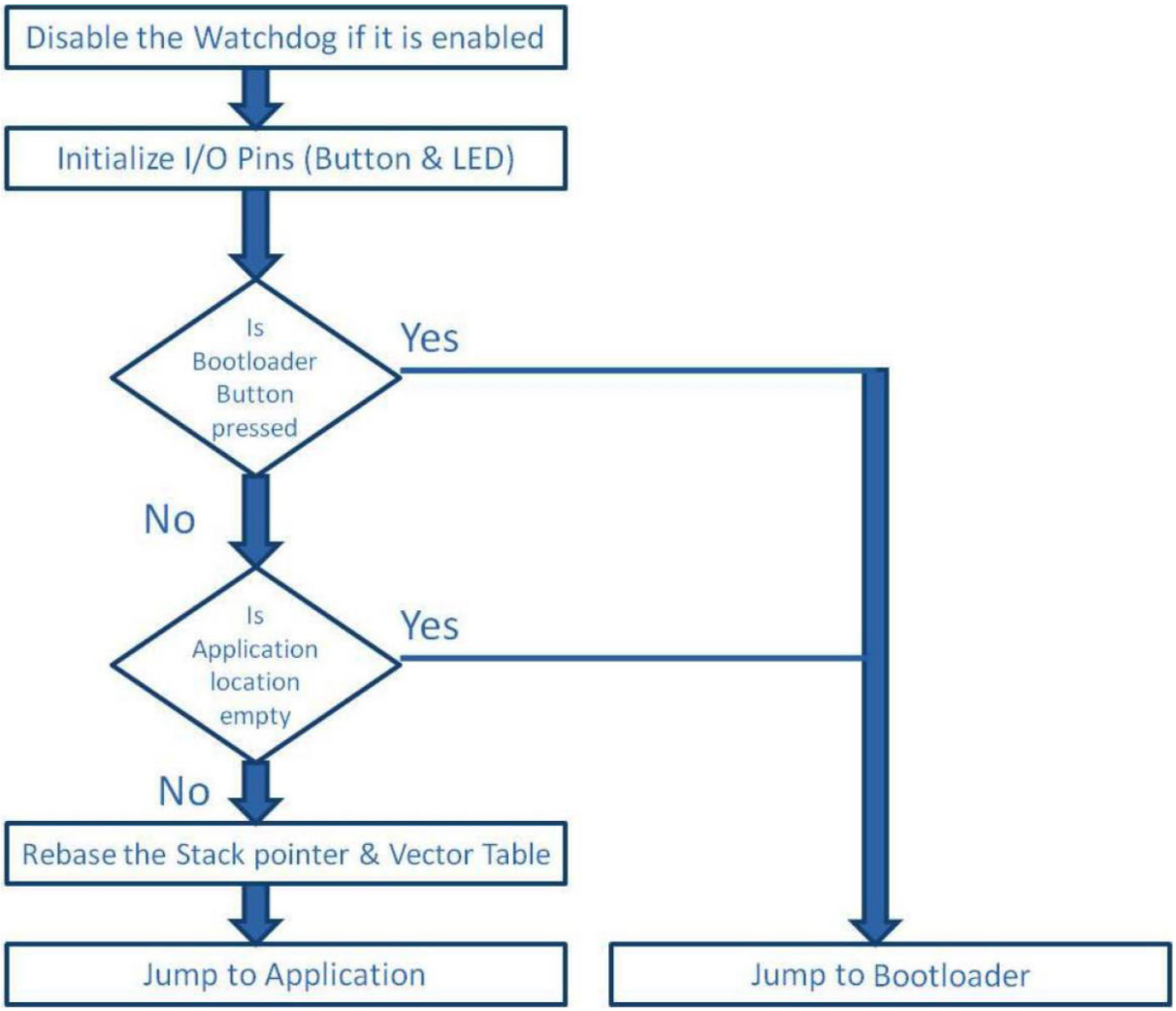
4.2 To Enter Bootloader or Application Mode on Starting

Now we are just missing the bootloader or application checking part at the start of main.

TO DO Add the following function before main().

- Add the code snippet before main() (code flow is explained in flowchart)

Figure 4-3. Boot Condition Check Flowchart



Code snippet:

```
static void check_boot_mode(void)
{
	uint32_t app_check_address;
	uint32_t *app_check_address_ptr;

	/* Check if WDT is locked */
	if (!(WDT->CTRL.reg & WDT_CTRL_ALWAYSON)) {
		/* Disable the Watchdog module */
		WDT->CTRL.reg &= ~WDT_CTRL_ENABLE;
	}

	app_check_address = APP_START_ADDRESS;
	app_check_address_ptr = (uint32_t *)app_check_address;

	board_init();

	if (port_pin_get_input_level(BUTTON_0_PIN) == BUTTON_0_ACTIVE) {
		/* Button is pressed, run bootloader */
		return;
	}

	if (*app_check_address_ptr == 0xFFFFFFFF) {
		/* No application; run bootloader */
		return;
	}

	/* Pointer to the Application Section */
	void (*application_code_entry)(void);

	/* Rebase the Stack Pointer */
	__set_MSP(*(uint32_t *)APP_START_ADDRESS);

	/* Rebase the vector table base address TODO: use RAM */
	SCB->VTOR = ((uint32_t)APP_START_ADDRESS & SCB_VTOR_TBLOFF_Msk);

	/* Load the Reset Handler address of the application */
	application_code_entry = (void (*)(void))(unsigned *)(*(unsigned *)
			(APP_START_ADDRESS + 4));

	/* Jump to user Reset Handler in the application */
	application_code_entry();
}
```

This function runs first when the chip is started, if either the button SW0 on the board is held down, or the first byte of the application-portion of the flash is empty, indicating that no firmware is present, it returns to main(), allowing the device to load a new firmware from a connected USB device.

If the button is not held down, and there is a firmware present, the code rebases the vector-table to point to the vector table of the loaded firmware, sets the stack pointer to the stack pointer of the firmware and finally transfers control to the firmware by running the reset handler of the firmware.

Add a call to this function as the very first function call in main():

```
check_boot_mode();
```

TIPS

Finally, It's good to ensure that "Use newlib-nano" check box is enabled in project properties. By default this newlib-nano is enabled in project. Newlib nano is a cut-down version of the C standard library, and using this library instead of the standard library will make the project take up much less space in the flash. As a summary, this linker option will enable the project to use more size optimized arm-gcc library

Atmel Studio → Project → Properties → ARM/ GNU Linker → General

Now we are ready to run the program. To test it, we will compile and upload our bootloader the SAM D21. Then we can copy a valid firmware called firmware.bin to a USB thumb drive and if you connect this thumb drive to the SAM D21 Xplained Pro, it should automatically program the firmware into the flash. If you then press the reset-button on the SAM D21 Xplained Pro, it should run the new firmware.

TO DO

Copy the file firmware.bin (available in Resources folder) to a USB thumb-drive. Run the program and connect the thumb-drive. Press reset when the firmware has been updated.

The new firmware will run after the chip has been reset.

TIPS

Application binary file named "firmware.bin" in Resources folder is precompiled binary file. It is being used to make this bootloader checking procedure easier. If required Section 4.3 will explain the steps to create an application binary file. Section 4.3 is not mandatory section.

If required, Solution project (SAMD21_USB_BOOT_ASSIGN3) is available in Resources folder.

Figure 4-4. Application Updated

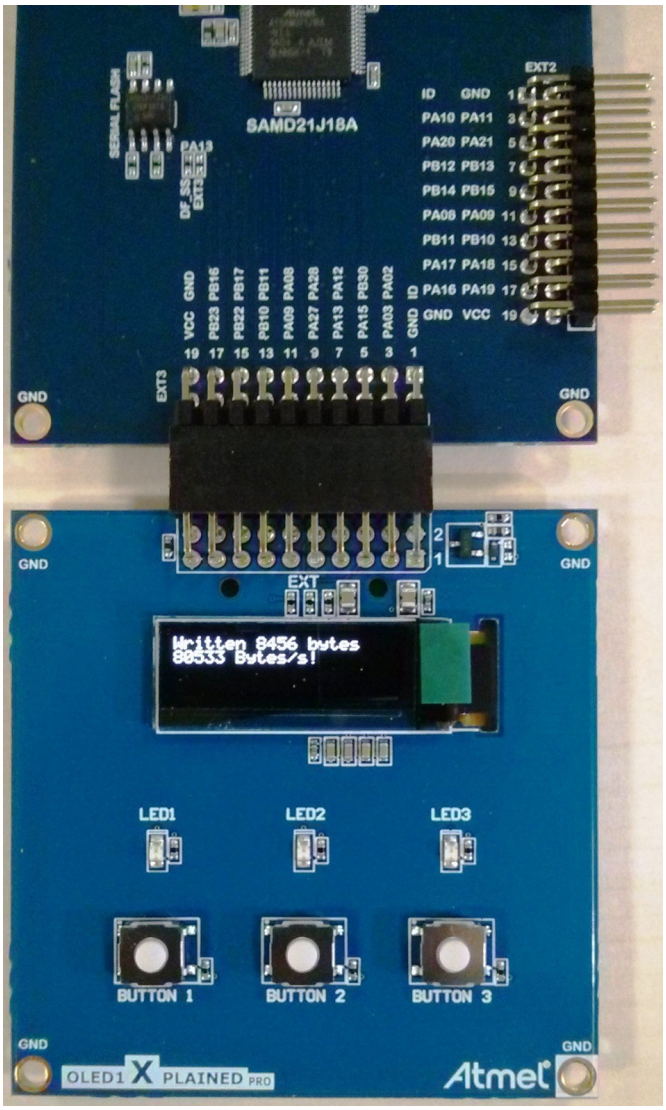
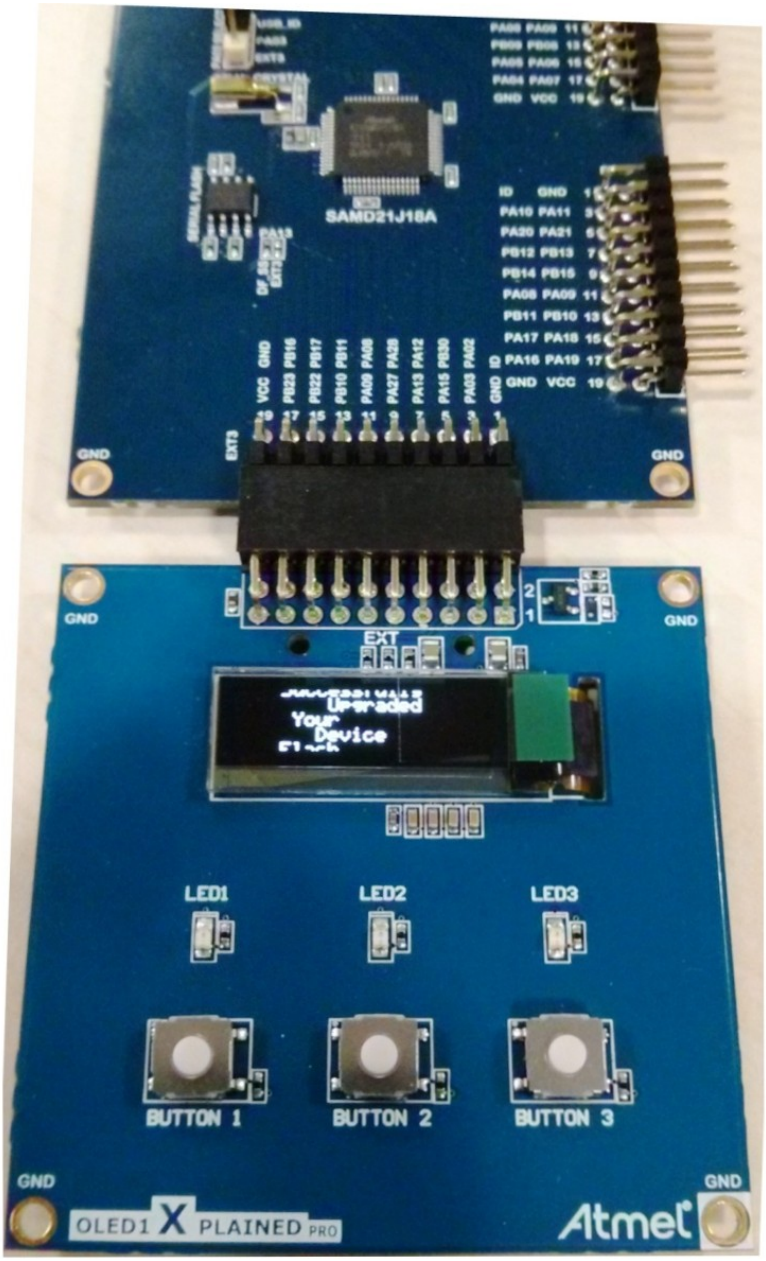


Figure 4-5. Application Running



4.3 Creating Application Binary File

This section explains the change that's needs to be done while creating application binary file that works with this bootloader. Only one change in device's linker script is required to make this binary as bootloader compatible binary.

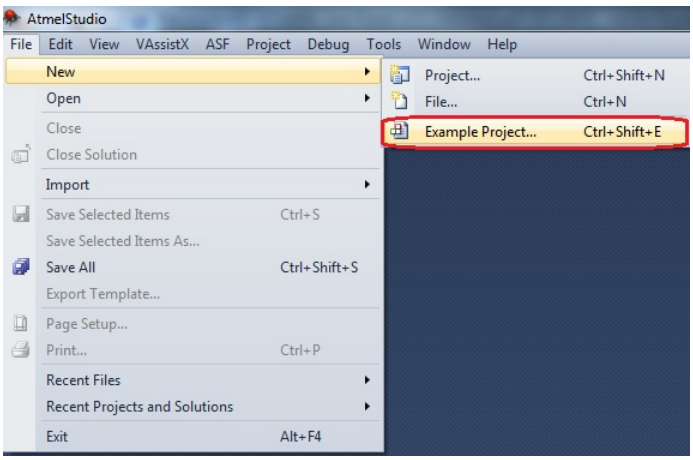
Flash address and size in linker script of this device has to be modified in order to make this as bootloader compatible binary file.

Following steps will guide in creating an ASF example project (GFX Monochrome System Font Example – SAM D21 Xplained PRO) and use this project's binary image as an application binary image. Explanation for this change is given after the procedure.

TO DO Launch Atmel Studio and open the example project (GFX Monochrome System Font Example – SAM D21 Xplained Pro) and compile the project.

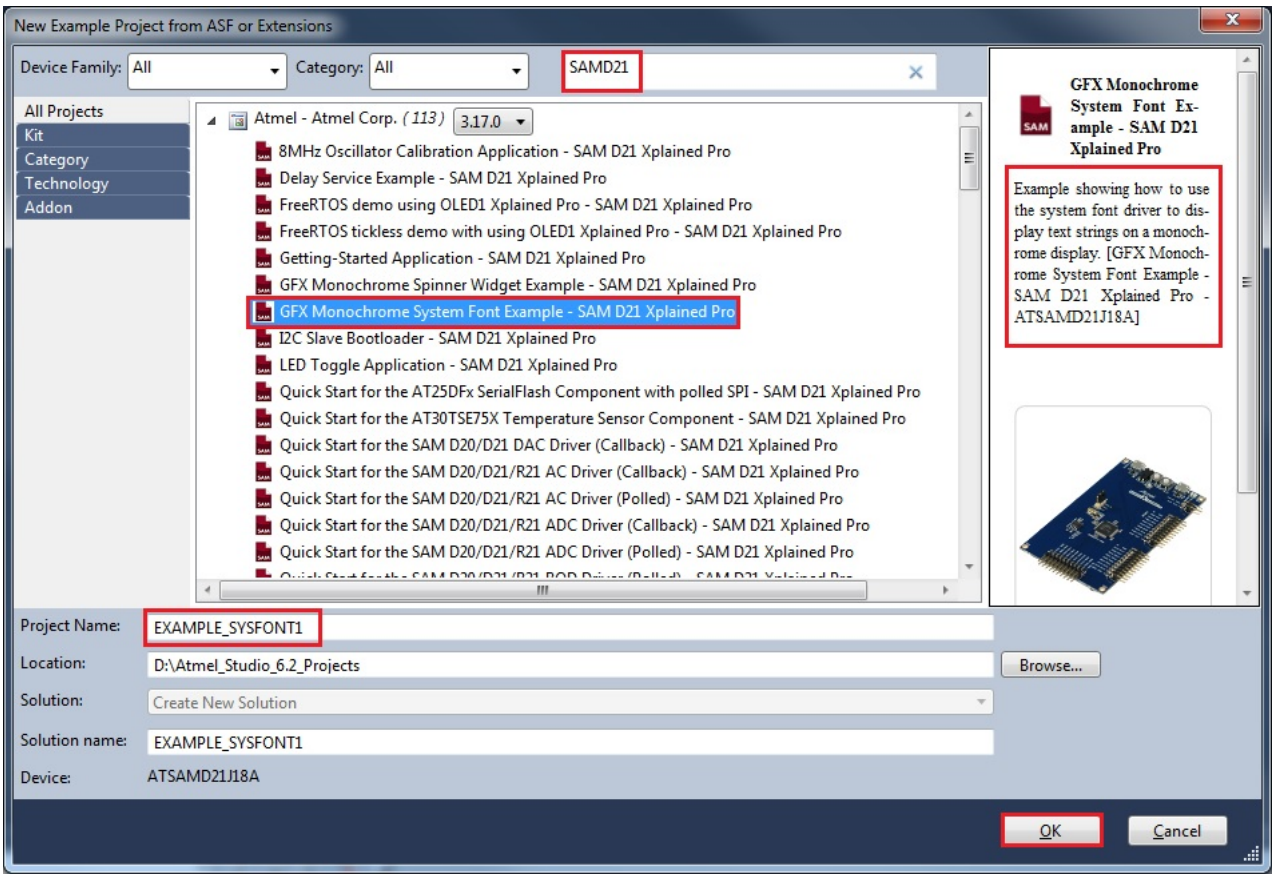
- Launch Atmel Studio -> File -> New -> Example Project from ASF -> search with SAMD21 and select the project (GFX Monochrome System Font Example – SAM D21 Xplained Pro)

Figure 4-6. New Example Project



TIPS Project name and location can be optionally changed if needed (see Figure 4-9).

Figure 4-7. GFX Monochrome Example Project for SAM D21 Xplained Pro

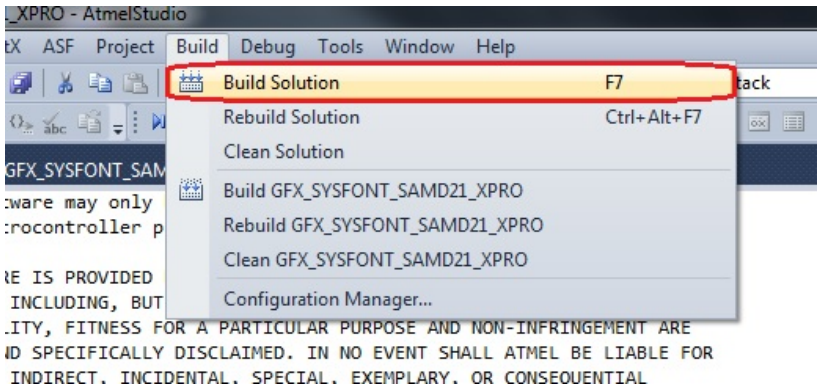


Once the project is created, compile the project and ensure that the compilation is successful. After ensuring, changes in the linker script can be made and re-compiled again.

TO DO Compile the project and ensure the successful compilation.

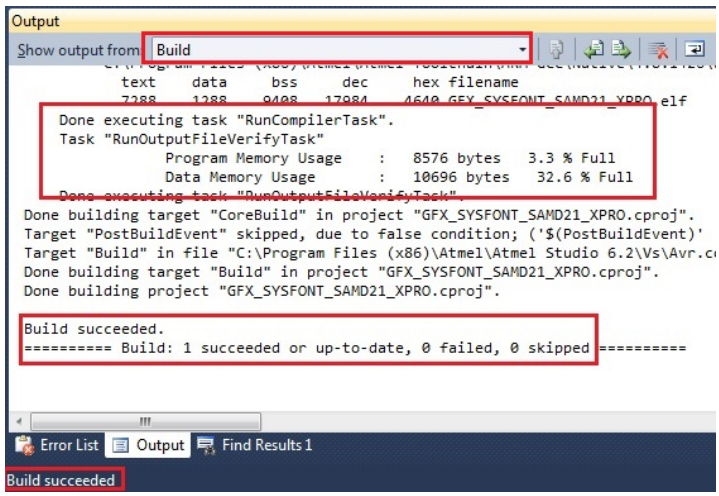
- Atmel Studio -> Build -> Build Solution or use shortcut (F7).

Figure 4-8. Build Solution



RESULT Compilation successful in Console window should be confirmed. See Figure 4-11.

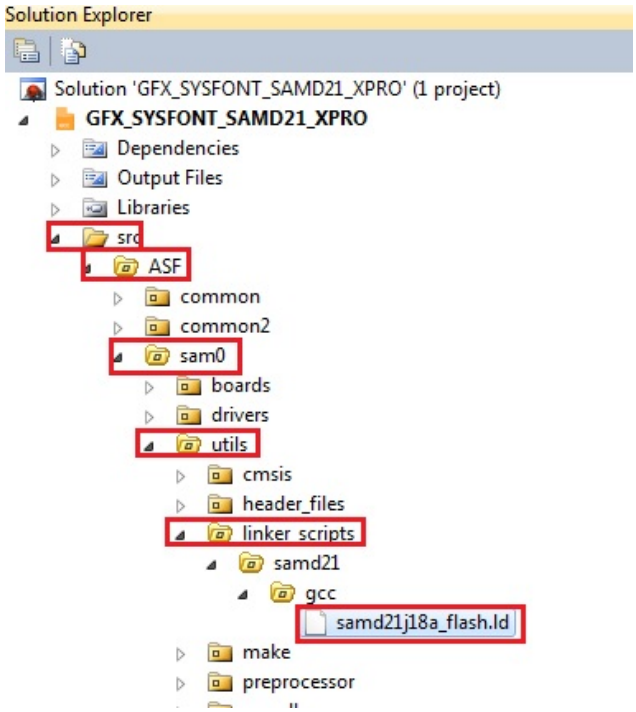
Figure 4-9. Compilation Successful



TO DO Change the linker script as mentioned and re-compile again.

- Find the linker script (samd21j18a_flash.ld) in the solution explorer and check the default value

Figure 4-10. Linker Script Location



TO DO Memory spaces definition in linker script has to be changed as follows.

- Default configuration is "`rom (rx) : ORIGIN = 0x00000000, LENGTH = 0x00040000`"
- This should be changed to "`rom (rx) : ORIGIN = 0x0000C800, LENGTH = 0x00033800`"

The linker script now looks like Figure 4-13.

Figure 4-11. Linker Script After Change

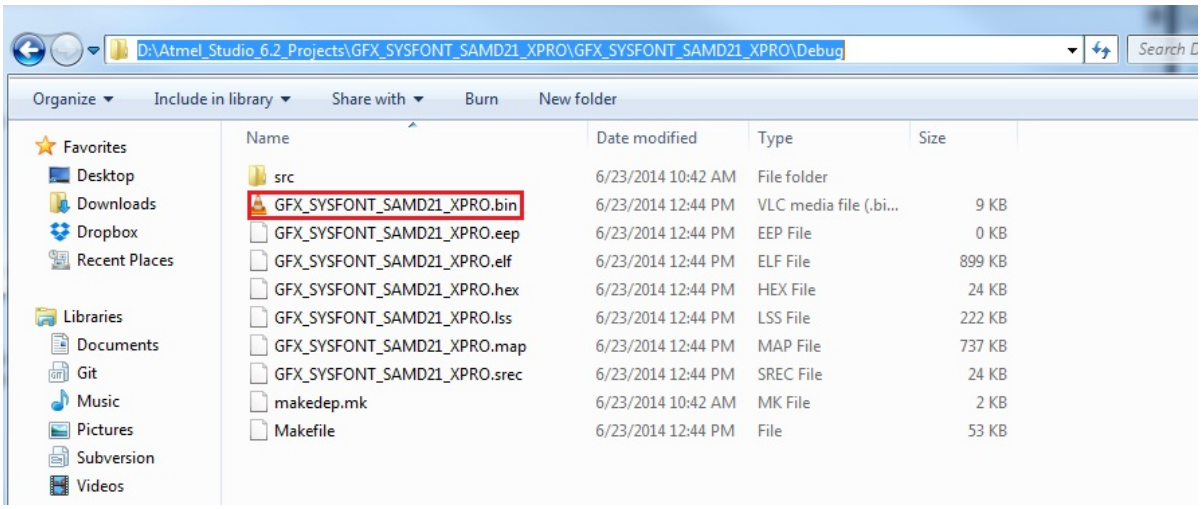
```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
SEARCH_DIR(.)

/* Memory Spaces Definitions */
MEMORY
{
  rom (rx)  : ORIGIN = 0x0000C800, LENGTH = 0x00033800
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00008000
}

/* The stack size used by the application. NOTE: you need to adjust accordin
STACK_SIZE = DEFINED(STACK_SIZE) ? STACK_SIZE : DEFINED(__stack_size__) ? __
```

TO DO Re-Compile the project and ensure the successful compilation as did before. Take the binary and rename it as "firmware.bin" in order to work with the bootloader.

- Atmel Studio -> Build -> Rebuild Solution or use shortcut (Ctrl + Alt + F7)
- Debug folder in the Project path will contain the binary image by default. Go to the debug folder path of the project and copy the binary. Paste it in the thumb drive and rename it as "firmware.bin".

  Example location is shown in Figure 4-14.

Figure 4-12. Binary Location



TIPS

Path for debug folder can be found by selecting the src folder in the Project in solution explorer -> Right click -> select “Open Folder” and this takes us to path where src folder resides. Navigating one folder-up will take us to Debug folder location.

Explanation for the change:

Application start address in this bootloader is the next address after bootloader section ends. 200 NVM Rows is the size we reserved for this bootloader in Flash memory space. So the next address (after 200 NVM Rows) has to be the first flash address for application binary.

NVM chapter in SAM D21 device datasheet can be referred for NVM memory organization.

In SAMD21J18A device, each NVM Row has four pages and each page has 64 bytes of memory. Therefore, 200 NVM Rows will contribute to (200 * 4 * 64) bytes = 51200 (0xC800) bytes of memory. Application section starts after 51200 bytes of flash memory.

The Partition of Flash space:

Bootloader section:

Size: 50Kbytes (51200 bytes).

Range: From Flash address 0x00000000 to 0x0000C7FF

Application section:

Size: 206Kbytes (256KB – 50KB).

Range: From flash address 0x0000C800 to 0x0003FFFF

By default, linker script contains:

Flash start address: 0x00000000

Flash size: 0x00040000

Linker script of Application binary should contain:

Flash start address: 0x0000C800 (starting address after 50KB)

Flash size: 0x00033800 (equivalent value of 206KB)

5 Conclusion

Here is the summary what we did.

- Checked the default project template available in Atmel Studio 6.2 for SAM D21 Xplained Pro Board
- Changed the clock configurations of the device, added drivers for OLED display and displayed a simple string
- Added USB Host, FatFS file system drivers and made a simple application to read the content of a file in a
- USB device when it is attached
- Added NVM driver and modified the main() to program the flash (Application part) with the contents of the file in the attached USB Device

6 Revision History

Doc Rev.	Date	Comments
42352A	02/2015	Initial document release.

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**