505403 M-Flex 4.3-Inch Color Display User Guide

CREATED:	C. PAWLAK	DATE:	10/06/2020
CHECKED:		DATE:	
APPROVED:		DATE:	
ECN:	13494E	DATE:	10/06/2020

Contents

1. Software Installation 2
2. Creating a Project in TouchGFX Designer 3
3. Opening an Existing Project in TouchGFX Designer 5
4. Programming the Display Module 6
5. Running the Simulator in TouchGFX Designer 9
5.1 Customizing Simulated Button Press Detection 10
6. Hardware Buttons 12
6.1 Customizing Button Press Detection 13
7. API Functions and Driver Functions 14
7.1 LEDs 14
7.2 Backlights 16
7.3 EEPROM 18
7.4 Inputs and Outputs 20
8. Tasks 23
9. Semaphores 24
10. CAN 25
10.1 CAN Rx 25
10.2 CAN Tx 29
11. Clock 31
12. Changing Application Software Identifiers 33
13. Troubleshooting Tips 35
Appendix: Giving a Button the Ability to Navigate to More Than One Possible Screen 36

1. Software Installation

Listed below is the PC software necessary to create application software for the 505403 M-FLex 4.3-Inch Color Display. Follow the directions beside each software item listed to install that piece of software.

Note: The use of asterisks (*) in file and folder names indicate varying text depending on the version of the software installed.

1. **TouchGFX Designer** is used to design the GUI (graphical user interface) for the screens displayed on the M-Flex display. Double-click on TouchGFX-*.*.*.msi to begin installing the software. Follow the directions on the screen to complete installation.

2. The **Marlin TouchGFX Template** enables TouchGFX Designer to generate applications that are compatible with the M-Flex display hardware. Copy 43inchMarlinTemplate-*.*.*.tpa to C:\TouchGFX*.*.*\app\packages. The exact directory path may vary depending on where TouchGFX Designer is installed.

3. The **Marlin Programming Tool** is used in conjunction with the USB-CAN dongle to download the user's application to the M-Flex display. Open the MarlinProgToolSetup_*_*_*_Basic folder and double-click MarlinProgToolSetup_Basic.msi to begin installing the software. Follow the directions on the screen to complete installation.

    3.1. Depending on where the Programming Tool was installed, the user guide should be located at C:\Program Files (x86)\Marlin Technologies\MarlinProgTool_Basic\UserGuide_*.pdf. Open the user guide for reference in the next step.

4. Install the driver for the appropriate **USB-CAN dongle,** as directed in the Programming Tool User guide.

WARNING: Using versions of software other than provided may result in errors and unintended functionality due to software and/or hardware incompatibility.

2. Creating a Project in TouchGFX Designer

Follow the steps below to create a project in TouchGFX Designer, in which screens for the user application can be designed.

Note: The use of asterisks (*) in file and folder names indicate varying text depending on the version of the software installed.

1. Double-click on C:\TouchGFX*.*.*\designer\TouchGFXDesigner-*.*.*.exe to start TouchGFX Designer. The exact file path may vary depending on where TouchGFX Designer was installed.

2. Click on the *MY APPLICATIONS* tab of the pop-up window. See Figure 1.

3. Hover over the window below *APPLICATION TEMPLATE*. Click *CHANGE* when that text appears, as shown in Figure 1.

Figure 1

4. Click on the *Marlin Technologies* tab as shown in Figure 2.

5. A box for the *M-FLEX 4.3" Color Display* template should now be displayed. Check the version of the template as indicated in Figure 2. If it is not the desired version proceed to step 6. Otherwise, skip to step 9.

6. Click the "i" icon (see Figure 2) to bring up a window that allows the template version to be changed.

Figure 2

7. To see the list of available template versions, click the up/down arrows beside the version number, as shown in Figure 3. If the up/down arrows are not available, or the desired version is not shown, contact Marlin Technologies. Otherwise, select the desired version from the drop-down menu.

8. Click the X in the upper-right corner of the window (see Figure 3) to close it.

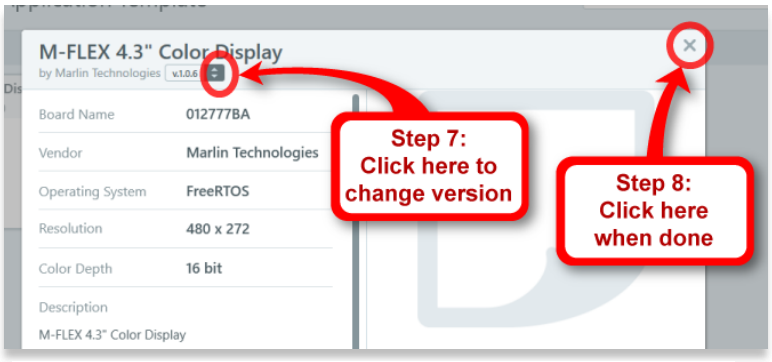

Figure 3

9. Click on the box for the *M-FLEX 4.3" Color Display* template (see Figure 4).

10. Then click *Select* (see Figure 4). The *Select* button will not become active until the template box is clicked in the previous step.

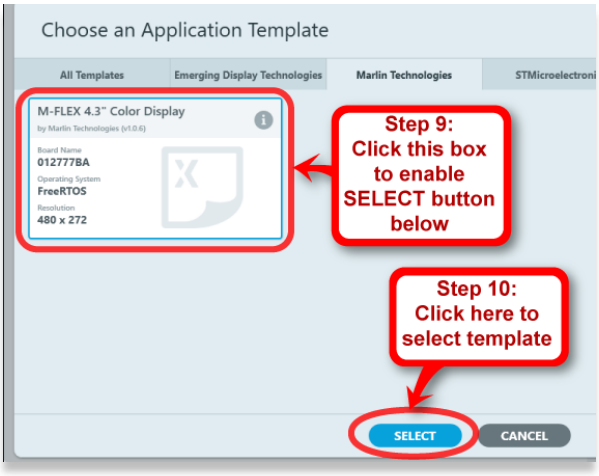

Figure 4

11. The window with the *MY APPLICATIONS* tab should now be displayed again, as shown in Figure 5. Change the application name as desired. This will create a folder of the same name in the location of the application directory, which will be set up in step 12.

12. Change the application directory as desired (see note in Figure 5.). Leave *UI TEMPLATE* set to a *Blank UI*.

13. Click *CREATE* (see Figure 5). A progress bar is displayed while the project is being created. Afterwards, the screen editor is displayed. See touchgfx.com for information about designing screens using TouchGFX Designer.

Figure 5

3. Opening an Existing Project in TouchGFX Designer

When an application has is created in TouchGFX Designer, a project file with the extension **.touchgfx** is created within the *TouchGFX* folder inside the project folder. For example, in the provided sample application software *MarlinDemo4_3Inch*, the project file is located at … \ MarlinDemo4_3Inch \ TouchGFX \ MarlinDemo4_3Inch.touchgfx. In Windows File Explorer, double-clicking the .touchgfx file will open the project in TouchGFX Designer.

Another way to open the project is to launch TouchGFX first, and then use the *SEARCH FOR RECENT APPLICATIONS* tool in the pop-up window, as shown circled in Figure 6 below, to navigate to the .touchgfx file.

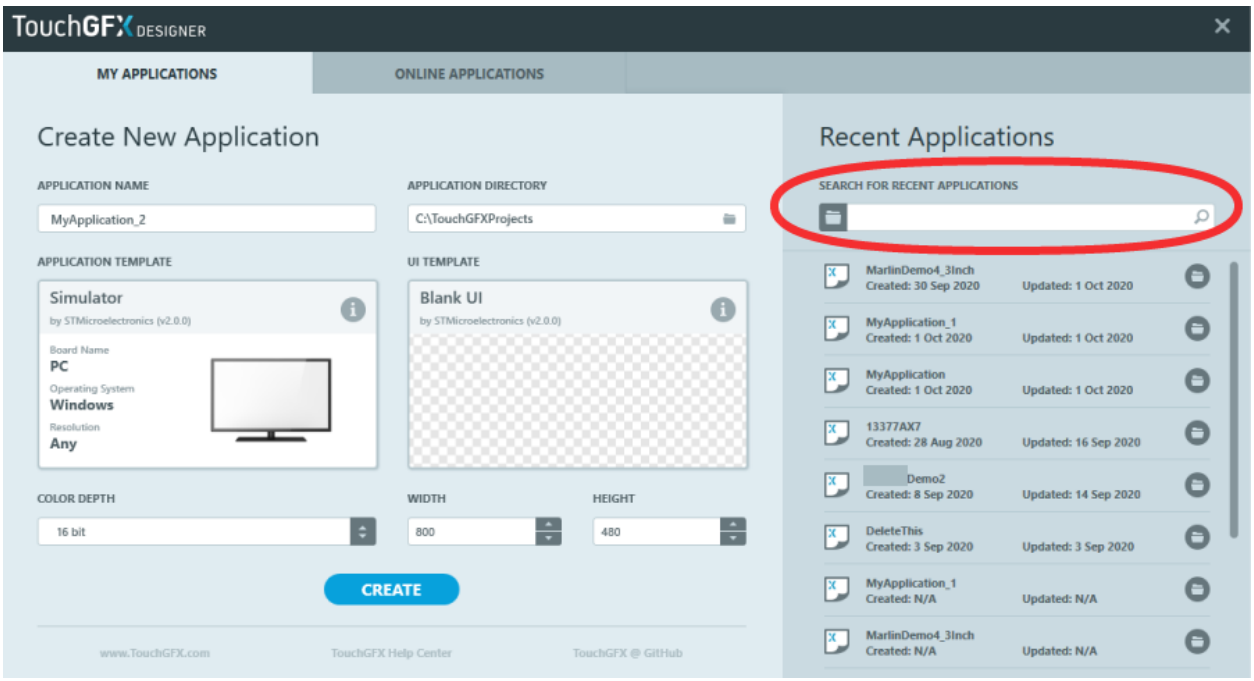

Figure 6

If the was project was opened recently, it may appear under *Recent Applications* in the pop-up window (below the circled selection in Figure 6). Simply click on the project to open it.

4. Programming the Display Module

Once a project has been created in TouchGFX Designer, application software can be built and then programmed onto the M-Flex display module by following the steps below

1. If the application software has already been built and an .s19 file already exists, skip to step 3. Otherwise, open the project in TouchGFX Designer (if not opened already). See section 3. Opening an Existing Project in TouchGFX Designer, as needed.

2. In TouchGFX Designer, click *Run Target* (as denoted in Figure 7) to build the application and generate the .s19 file that the Programming Tool will use to program the display. The status of the build is displayed at the bottom of the TouchGFX Design window (as indicated in Figure 7). When the build is complete, the build status is shown as "Run Target | Failed," even if the build is successful. This is expected. A failure is indicated because TouchGFX Designer attempts (and is unable) to program the display. Programming will be carried out in a later step.

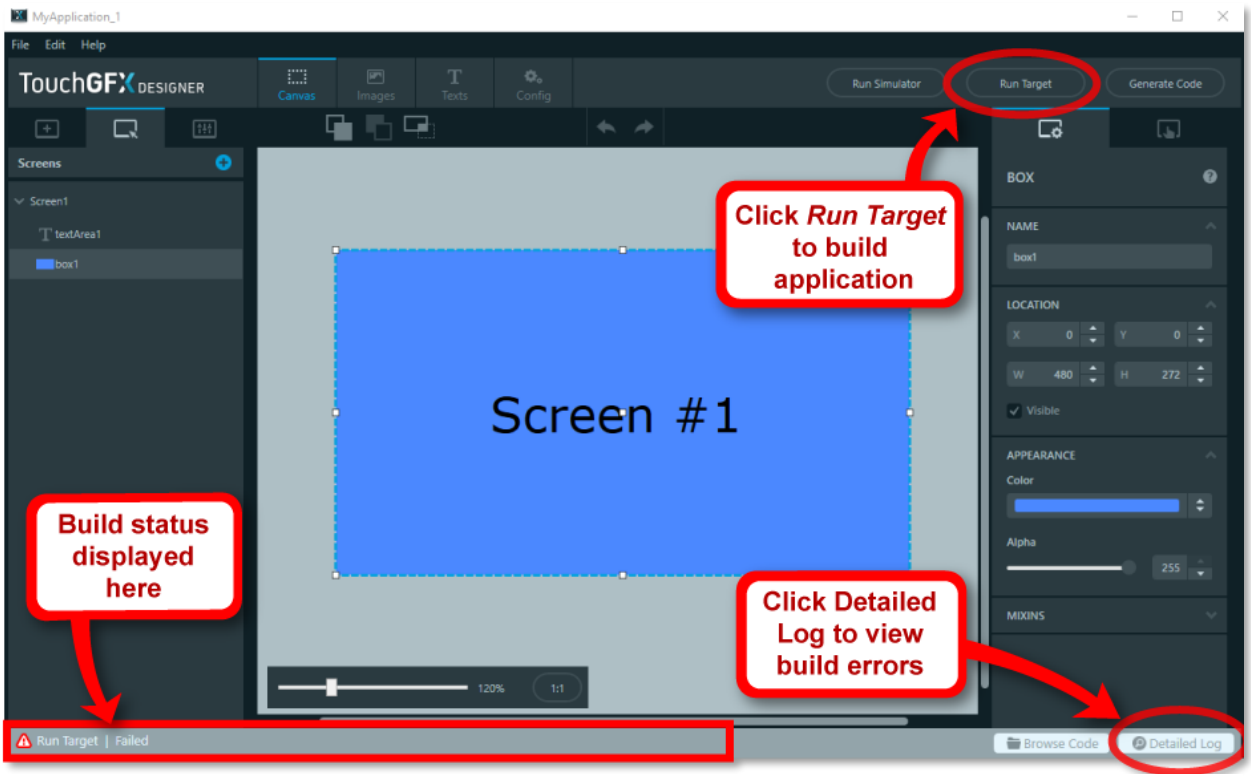

Figure 7

If the application builds successfully, the file *update_format.s19* will be created (or, if it already exists, modified with the current time and date) in the *build* folder in the project directory. The outcome of the build will also be indicated near the bottom of the detailed log. The detailed log is accessed by clicking the *Detailed Log* in the lower right corner of the TouchGFX Designer window (as shown in Figure 7). For a successful build, the log indicates that *update_format.s19* has been created (or modified) and displays the program's checksum. Otherwise, the log will indicate which errors prevented a successful build. A quick way to check if a build successful is to look for the checksum text in the log, as shown in Figure 8:

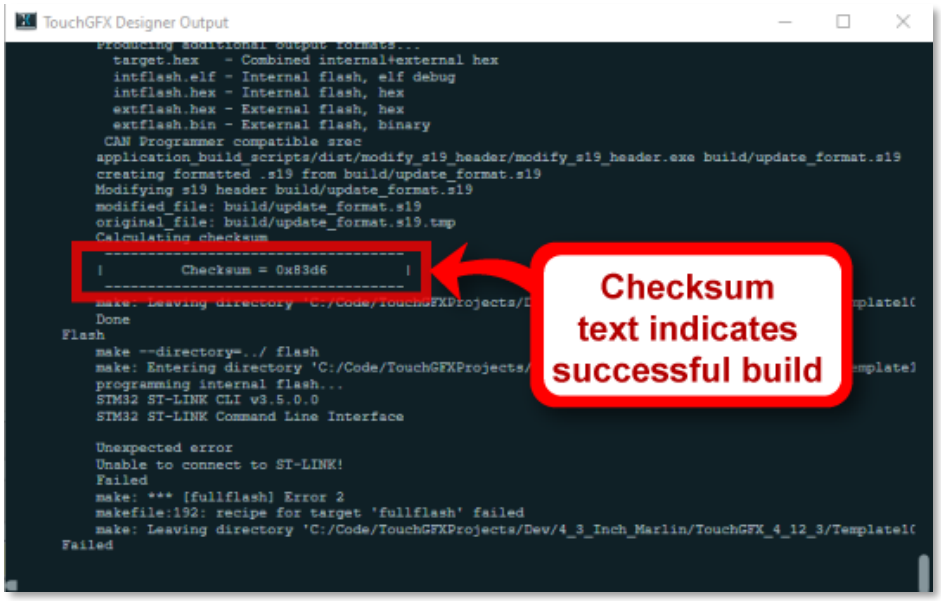

Figure 8

3. Connect the USB-CAN dongle to the PC and the CAN bus.

4. Power and CAN lines will need to be connected to the J1 connector on the back of the display modules. See Figure 9 shows the pin numbers on J1. Connect power and CAN lines as follows:
   4.1. Connect CAN LO to pin 7 of J1 and CAN HI to pin 8 of J1.
   4.2. Connect battery voltage (12V) to pin 1 of J1, and ground to pin 2 of J1.

   See the module's outline drawing for further details regarding the J1 connector pins.

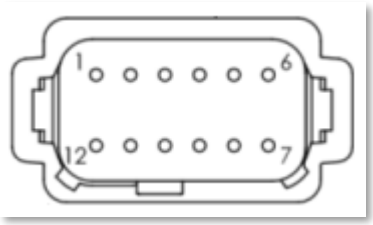

Figure 9: Connector J1

5. With the display module powered on, launch the Marlin Programming Tool application, as directed in the Programming Tool User Guide. The tool should automatically connect to the M-Flex display, indicating a module has been found. If not, click the *Search* button (as denoted in

Figure 10) to attempt another connection. If the Programming tool continues to indicate "No Controllers Found," check the set-up. Make sure the baud rate is set to 250 Kbps. See the Programming Tool User Guide for further information.

6. Click the *Browse* button (as denoted in Figure 10) and select *update_format.s19* (located in the *Build* folder of the TouchGFX project).

7. Click the *Program* button (as denoted in Figure 10) to download the application to the display module.

Figure 10

5. Running the Simulator in TouchGFX Designer

Some screen functionality can be tested without programming the display module. This is possible using TouchGFX Designer's simulator, which is activated by clicking the *Run Simulator* button in the upper-right corner of window, as indicated in Figure 11.

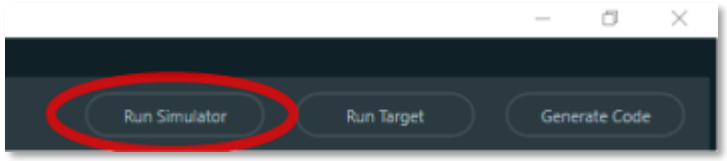

Figure 11

Once the simulator is running, button presses on the module can be simulated by pressing keys 1 through 8 on the PC keyboard. (Note: This does not include keys 1 through 8 on the numeric keypad of the PC keyboard.) The module buttons and their corresponding PC keys are indicated in Figure 12.

Figure 12

Other hardware functionality (e.g. CAN messages, screen backlight, button backlights, LEDs, inputs, and outputs) *cannot* be simulated. For any code written involving such functionality, be sure to use "#ifndef SIMULATOR" to avoid simulator compile errors. For example, see the highlighted text in the code below:

```
void LEDsView::displayRgbGreenLevel()
{
        rgbGreenBar.setValue(rgbGreenLevel);
        rgbGreenBar.invalidate();

        Unicode::snprintf(rgbGreenValBuffer, RGBGREENVAL_SIZE, "%u", rgbGreenLevel);
        rgbGreenVal.invalidate();

#ifndef SIMULATOR
        setColorLedRGB(rgbRedLevel * 10,rgbGreenLevel * 10,rgbBlueLevel * 10);
#endif
}
```


The line of code between the highlighted text turns on LEDs in an actual application, but it would cause a compile error when *Run Simulator* is clicked. The #if/#endif statement prevents that code from compiling the code for the simulator.

5.1 Customizing Simulated Button Press Detection

Simulated button-press detection is handled in **sampleKeys()**, which is defined in code_generation_resources\FrontendApplication.hpp:

```
        virtual void sampleKeys(){
                if(GetKeyState('1') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<0));
                }
                else if(GetKeyState('2') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<1));
                }
                else if(GetKeyState('3') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<2));
                }
                else if(GetKeyState('4') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<3));
                }
                else if(GetKeyState('5') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<4));
                }
                else if(GetKeyState('6') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<5));
                }
                else if(GetKeyState('7') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<6));
                }
                else if(GetKeyState('8') & 0x8000){
                        FrontendApplicationBase::handleKeyEvent((1<<7));
                }
                return;
        }
```

The highlighted instances of text above represent each of the PC keyboard keys that are used to simulate button presses on the display module. Each instance corresponds to a similarly numbered button in Figure 12. If the buttons are used to trigger simple actions, like screen transitions, the code above will likely suffice.

However, for more elaborate functionality, it may be necessary to modify sampleKeys(). Consider the case where the down-button moves a cursor downwards through a list of menu items, and keeping the button held down causes the cursor to rapidly repeat the downwards movement until the button is

released. With the definition of sampleKeys() above, holding down the down-key will move the cursor too rapidly for the operator to keep track of.

For the sample application *MarlinDemo4_3Inch,* a wait-time was added to ensure the cursor moves at a more reasonable rate. See the use of `KEY_WAIT` and `tickCounter` below:

```
void FrontendApplication::sampleKeys()
{
        if(GetKeyState('1') & 0x8000)
        {
                if(tickCounter++ > KEY_WAIT)
                {
                        tickCounter = 0;
                        FrontendApplicationBase::handleKeyEvent((1<<0));
                }
        }
        else if(GetKeyState('2') & 0x8000)
        {
                if(tickCounter++ > KEY_WAIT)
                {
                        tickCounter = 0;
                        FrontendApplicationBase::handleKeyEvent((1<<1));
                }
        }
   .
   .
   .
}
```

Note: in *MarlinDemo4_3Inch* sample code, the definition of sampleKeys() was moved from FrontendApplication.hpp to FrontendApplication.cpp (both files in the same folder).

The resulting functionality is only a simplified version of the *actual* button-press-detection functionality. The actual functionality is handled in KeySampler.cpp, which is discussed in section *5.1 Customizing Button Press Detection*. For the *MarlinDemo4_3Inch* sample application, the code changes above are sufficient in most cases, though some rapid key presses may be missed, for example. If a more accurate simulation of button presses is needed, change the code in sampleKeys() to more closely resemble the code in KeySampler.cpp.

6. Hardware Buttons

To make use of the display module's buttons, select the *Interactions* tab near the upper-right corner of the TouchGFX Designer window (see Figure 13) and click the *Add Interaction* button.

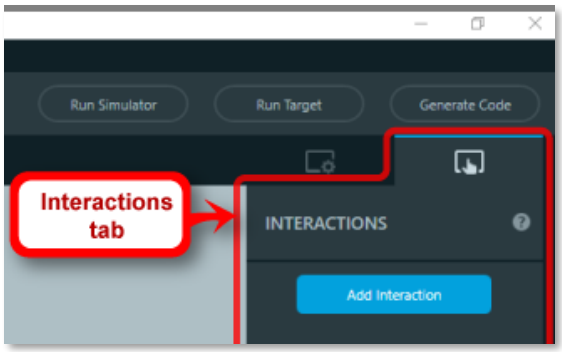

Figure 13

A new interaction and its settings should now be displayed. Go to the *Trigger* pull-down menu and select *Hardware button is clicked*, as shown in Figure 14. The next setting, *Choose button key* (also shown in Figure 14), ties the interaction to a given button on the display module. Each button is represented by a unique number, as shown in Figure 15. For example, if the interaction is intended to be executed when the UP button is pressed, select *16* from the drop-down menu under *Choose button* key (as was done in Figure 14).

Now choose an item from the *Action* pull-down menu to determine what the button press should do. Figure 14, for example, shows how the user would configure the *Action* setting to so that a button press would change screens. With the *Change screen option* selected, a pull-down menu appears which includes all of the user-defined screens. (If the screen does not already exist, it cannot be selected from the pull-down menu.)

Button interactions like the one in Figure 14 configure is limited to linking a given button to only one possible screen. On a main menu screen, such as the one implemented in the sample application software *MarlinDemo4_3Inch*, there may be a need for an Enter button that navigates to *various* possible screens, depending on what menu item the cursor is on. One way to accomplish this is detailed in the *Appendix: Giving a Button the Ability to Navigate to More Than One Possible Screen.*

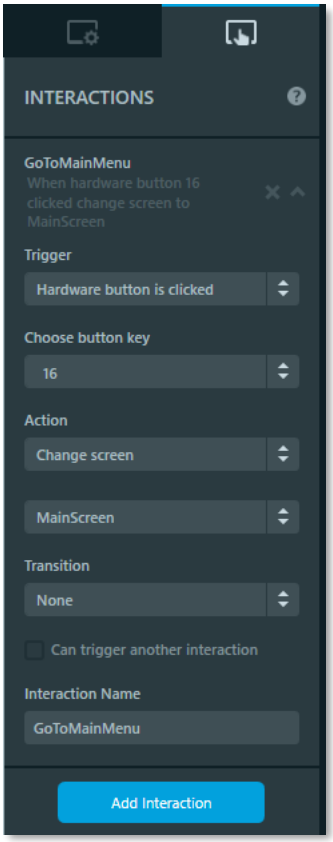

Figure 14

Figure 15

6.1 Customizing Button Press Detection

Button functionality can be fine-tuned in the source code by modifying **\Src\touchGFXUserCode\KeySampler.cpp**. Here debounce logic for the buttons can be adjusted. This logic can also be modified, for example, to notify the rest of the code whether a key has merely been pressed, or whether it has been pressed *and released*.

It can also be modified to signal when multiple buttons are pressed simultaneously. Note that the buttons in Figure 15 are numbered non-consecutively; i.e. 1, 2, 4, 8… instead of 1,2,3,4… That is because each button corresponds to a bit of the byte `assembled_mask`, as shown in KeySampler::sampler() below:

```
bool KeySampler::sample(uint8_t& key)
{
	uint8_t assembled_mask =
		(((HAL_GPIO_ReadPin(SW1_GPIO_Port, SW1_Pin)==GPIO_PIN_RESET)?1:0)<<0) |
		(((HAL_GPIO_ReadPin(SW2_GPIO_Port, SW2_Pin)==GPIO_PIN_RESET)?1:0)<<1) |
		(((HAL_GPIO_ReadPin(SW3_GPIO_Port, SW3_Pin)==GPIO_PIN_RESET)?1:0)<<2) |
		(((HAL_GPIO_ReadPin(SW4_GPIO_Port, SW4_Pin)==GPIO_PIN_RESET)?1:0)<<3) |
		(((HAL_GPIO_ReadPin(SW5_GPIO_Port, SW5_Pin)==GPIO_PIN_RESET)?1:0)<<4) |
		(((HAL_GPIO_ReadPin(SW6_GPIO_Port, SW6_Pin)==GPIO_PIN_RESET)?1:0)<<5) |
		(((HAL_GPIO_ReadPin(SW7_GPIO_Port, SW7_Pin)==GPIO_PIN_RESET)?1:0)<<6) |
		(((HAL_GPIO_ReadPin(SW8_GPIO_Port, SW8_Pin)==GPIO_PIN_RESET)?1:0)<<7);

	if(assembled_mask != most_recent_detected_mask){
		handle_changed_mask(assembled_mask);
	}
	else if(assembled_mask && is_debounced()){
		return handle_debounced_mask(key);
	}

	return false;
}
```

Note that when a GUI is imported into TouchGFX Designer from an existing application, KeySampler.cpp is not copied over. The code must be copied over manually if such button functionality is to be retained.

7. API Functions and Driver Functions

Much of the display module's hardware functionality can be controlled using the API (application programming interface) functions in Src\bsp.c, while other functionality can be controlled with driver functions in hardware-specific files. Many such functions are demonstrated in the sample application software *MarlinDemo4_3Inch*.

7.1 LEDs

Two types of LEDs – red and RGB – are located to the sides of the hardware button as shown in Figure 16 through Figure 19. They can be turned on at different duty cycles to achieve various levels of brightness. Turning on one type of LED will turn on that type of LED (at the same brightness) on both the left and right side of the display module. For example, it is not possible to turn on the red LED on the left side of the display without also turning on the red LED on the right side.

Figure 16: All LEDs off

Figure 17: Red-only LEDs on

Figure 18: RGB LEDs on (blue internal LED only)

Figure 19: Both Red-only and RGB LEDs on

The red, green, and blue internal LEDs within the RGB LEDs can be turned on individually at different duty cycles, making a wide spectrum of colors available. Both the red and RGB LEDs can be turned on at the same time, with different colors visible, as shown in Figure 19. The RGB LED located at the top of the LED light pipe, while the red LED is located at the bottom.

Use **setRedLedIntensity()** (Src\bsp.c) to turn on the red LED:

```
void setRedLedIntensity (uint32_t duty)
```

where `duty` ranges from 0 to 1000, representing 0 to 100% duty cycle.

Use **setColorLedRGB ()** (Src\bsp.c) to turn on the RGB LED:

```
void setColorLedRGB(uint32_t R_duty, uint32_t G_duty, uint32_t B_duty)
```

where `R_duty`, `G_duty`, and `B_duty` each range from 0 to 1000, representing 0 to 100% duty cycle.

The individual internal LEDs of the RGB LED can also be turned on with **setColorLedRed()**, **setColorLedGreen()**, and **setColorLedBlue()** (all in Src\bsp.c):

```
void setColorLedRed(uint32_t duty)
void setColorLedGreen(uint32_t duty)
void setColorLedBlue(uint32_t duty)
```

where `duty` ranges from 0 to 1000, representing 0 to 100% duty cycle. Note that calling any of these functions for a given internal LED does not turn off the other internal LEDs.

When adding LED API function calls to a file, be sure to add `#include "bsp.h"` to the top of that file (or its header file), or else there may be compile errors.

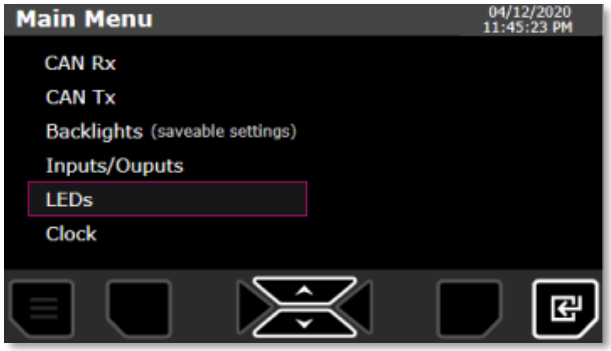

Figure 20

The use of LED API functions above are demonstrated in the sample application software *MarlinDemo4_3Inch*. The LEDs screen (see Figure 21) is available to the user via the *LEDs* item on the Main Menu (see Figure 20).

With *the MarlinDemo4_3Inch* project opened in TouchGFX Designer, refer to the *LEDs* screen in the *Screens* tab on the left-hand side of the window. The associated source code files, which makes use of the LED API functions, are located in MarlinDemo4_3Inch\TouchGFX\gui:

- \src\leds_screen\LEDsView.cpp
- \include\gui\leds_screen\LEDsView.hpp

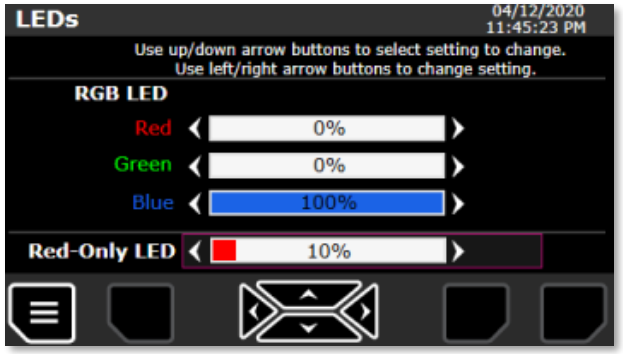

Figure 21

7.2 Backlights

The LCD backlight, as well as the backlights for the buttons, can be turned on at different duty cycles to achieve various levels of brightness. The button backlights are controlled in tandem. That is, a given button backlight cannot be set to a brightness level different from the other backlight buttons. See Figure 22 and Figure 23 for views of the button backlights off and fully on, respectively.

Figure 22: Button backlights off

Figure 23: Button backlights fully on

Use **setKeypadBacklightIntensity()** (Src\bsp.c) to adjust button backlights:

```
void setKeypadBacklightIntensity (uint32_t duty)
```

where `duty` ranges from 0 to 1000, representing 0 to 100% duty cycle.

Use **setLcdBacklightIntensity()** (Src\bsp.c) to adjust LCD backlight:

```
void setLcdBacklightIntensity (uint32_t duty)
```

where `duty` ranges from 0 to 1000, representing 0 to 100% duty cycle.

When adding backlight API function calls to a file, be sure to add `#include "bsp.h"` to the top of that file (or its header file), or else there may be compile errors.

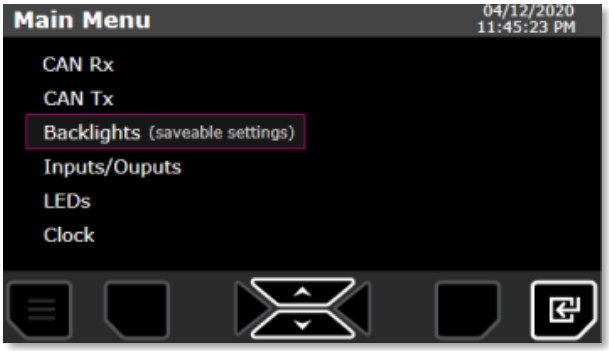

Figure 24

The backlight-related API functions above are demonstrated in the sample application software *MarlinDemo4_3Inch*. The *Backlights* screen (see Figure 25) is available to the operator via the *Backlights (savable settings)* item on the Main Menu (see Figure 24).

With the *MarlinDemo4_3Inch* project opened in TouchGFX Designer, refer to the *Backlights* screen in the *Screens* tab on the left-hand side of the window. The associated source code files, which make use of the backlight API functions, are located in MarlinDemo4_3Inch\TouchGFX\gui:

- src\backlights_screen\BacklightsView.cpp
- include\gui\backlights_screen\BacklightsView.hpp

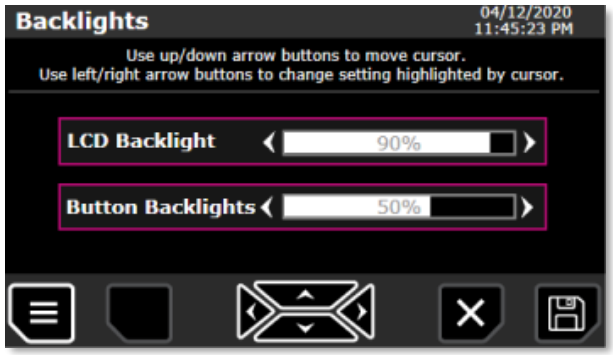

Figure 25

Note: See the section 7.3 EEPROM for information on how the settings on the *Backlights* screen are saved.

7.3 EEPROM

The display module contains a 512Kb EEPROM (25LC512). Driver functions for interfacing with the EEPROM are located in Src\applicationCode**25lc512t.c**. Communication with the EEPROM is via SPI (serial peripheral interface), so many of the driver functions require a pointer to an SPI object (SPI_HandleTypeDef) as an input parameter. For these functions, pass the following object by reference, which is defined in Src**spi.c**:

```
SPI_HandleTypeDef hspi2;
```

Before accessing the EEPROM, **EE_InitSpi()** must be called:

```
void EEP_InitSpi(SPI_HandleTypeDef* HSPI)
```

When a project is created in TouchGFX Designer, a call to EEP_InitSpi() is automatically included in main() (Src\main.cpp):

```
EEP_InitSpi(&hspi2);
```

Note how hspi2 (mentioned above) is passed by reference.

Use the following functions to read/write data to the EEPROM, or check the EEPROM status:

```
uint8_t EEP_ReadStatus(SPI_HandleTypeDef* HSPI);
uint8_t EEP_ReadByte(SPI_HandleTypeDef* HSPI, uint16_t Address);
void EEP_ReadBlock(SPI_HandleTypeDef* HSPI, uint16_t Address, uint8_t* pData, uint16_t Size);
void EEP_EraseChip(SPI_HandleTypeDef* HSPI);
void EEP_WriteByte(SPI_HandleTypeDef* HSPI, uint16_t Address, uint8_t Data);
```

where

- `*HSPI` should be `hspi2` passed by reference,
- `Address` is the EEPROM address (starting at 0)
- `*pData` points to a block of data to be read or written to
- `Data` is a byte of data to write
- `Size` is the number of bytes of data to be read/written.

The sample application software *MarlinDemo4_3Inch* uses the EEPROM-related code above to store the backlight settings available on the *Backlights* screen (see section 7.2 Backlights above). Figure 26 demonstrates the layers of code involved in storing these settings on the EEPROM.

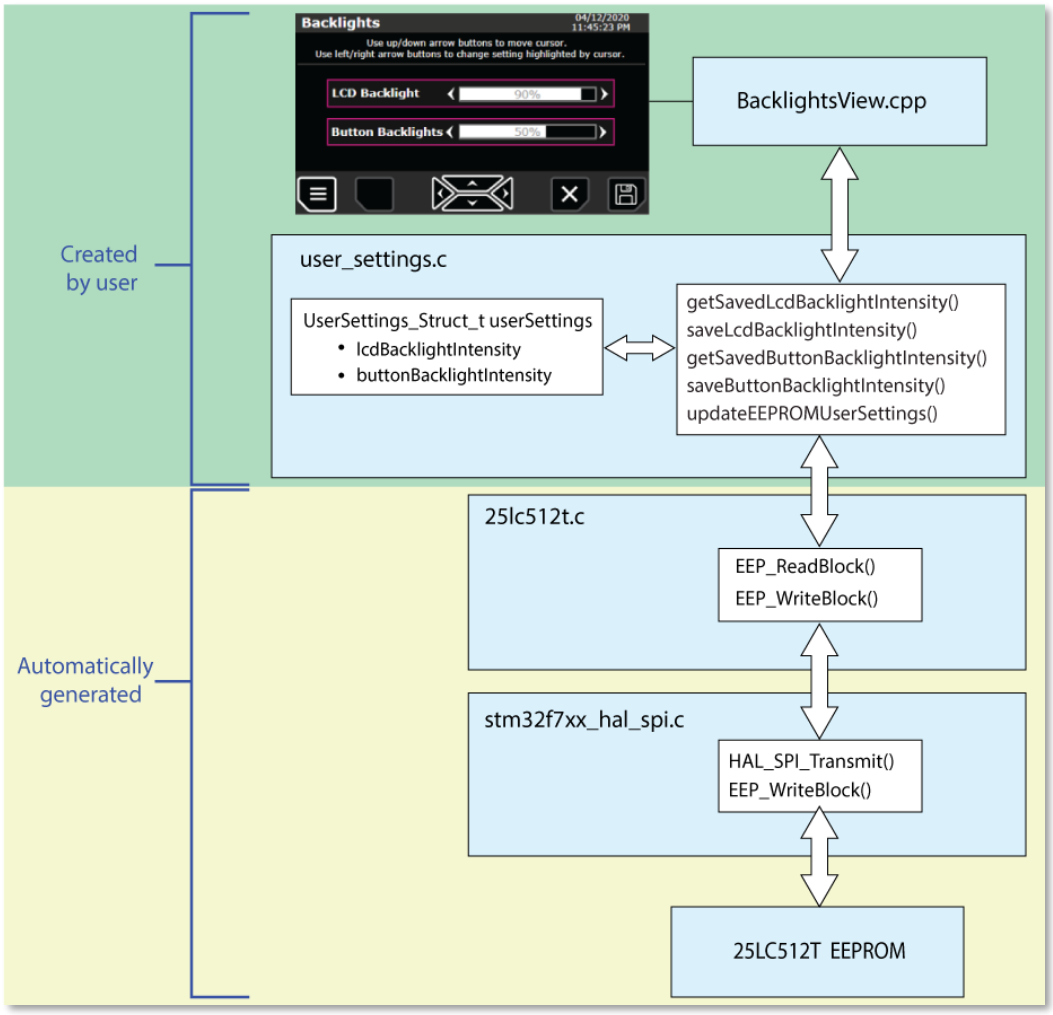

Figure 26

The settings stored in the EEPROM are stored locally (on the microcontroller) in the struct *userSettings* (Src\user_settings.c):

```
static UserSettings_Struct_t userSettings;
```

The struct is initialized with **initUserSettings()**, which is called in main():

```
void initUserSettings(void);
```


Individual settings within the struct *userSettings* are written to or read from settings using the following functions:

```
uint8_t getSavedLcdBacklightIntensity(void);
void saveLcdBacklightIntensity(uint8_t);
uint8_t getSavedButtonBacklightIntensity(void);
void saveButtonBacklightIntensity(uint8_t);
```

These functions are called from other files, including BacklightsView.cpp, where the settings from the *Backlights* screen can be changed (and saved) by the operator (see section 7.2 Backlights).

The settings are written from *userSettings* to the EEPROM with **updateEEPROMUserSettings():**

```
void updateEEPROMUserSettings();
```

The settings are read from the EEPROM and into *userSettings* with **readEEPROMUserSettings()**:

```
void readEEPROMUserSettings();
```

updateEEPROMUserSettings() and readEEPROMUserSettings(), in turn, call the functions in 25lc512t.c, described above, to access the EEPROM.

7.4 Inputs and Outputs

The input and output pins on the J1 connector (see Figure 27), located on the back of the display module, are configured as followed:

- **Digital Input 1:** J1-10
- **Digital Input 2:** J1-9
- **Digital Input 3:** J1-3
- **Digital Input 4:** J1-4
- **External 5V sensor:**
  - **5V:** J1-6,
  - **GND:** J1-5
- **PWM Output 1:** J1-11
- **PWM Output 2:** J1-12

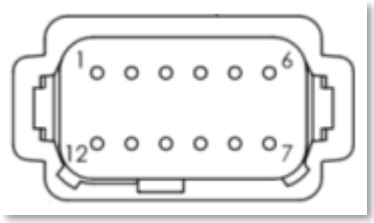

Figure 27: Connector J1

See the outline drawing for the display module for details regarding J1 connector pins.

The four digital input statuses are read using the following function from Drivers\STM32F7xx_HAL_Driver\Src**stm32f7xx_hal_gpio.c**:

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

which returns 0 for off and 1 for on.

More specifically:

- For Digital Input 1 (J1-10), use **HAL_GPIO_ReadPin(GPIOH, GPIO_PIN_6)**
- For Digital Input 2 (J1-9), use **HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_15)**
- For Digital Input 3 (J1-3), use **HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_12)**
- For Digital Input 4 (J1-4), use **HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_13)**

Use **enableExt5vPower()** (Src\bsp.c) to turn on the external 5V sensor (J1-6):

```
void enableExt5vPower(void)
```

Use **disableExt5vPower()** (Src\bsp.c) to turn off the external 5V sensor (J1-6):

```
void disableExt5vPower (void)
```

Use **setOutput_1()** (Src\bsp.c) to set the duty cycle of PWM Output 1 (J1-11):

```
void setOutput_1(uint32_t duty)
```

where `duty` ranges from 0 to 1000, representing 0 to 100% duty cycle.

Use **setOutput_2()** (Src\bsp.c) to set the duty cycle of PWM Output 2 (J1-12):

```
void setOutput_1(uint32_t duty)
```

where `duty` ranges from 0 to 1000, representing 0 to 100% duty cycle.

When adding I/O-related API functions mentioned above to a file, be sure to add `#include "bsp.h"` to the top of that file (or its header file), or else there may be compile errors. When using "`HAL_GPIO_ReadPin()`," `#include main.h` can be added to the file (or its header file).

The I/O-related API functions and driver functions above are used in the sample application software *MarlinDemo4_3Inch*. The *Inputs/Outputs* screen (see Figure 29) is available to the operator via the *Inputs/Outputs* item on the Main Menu (see Figure 28).

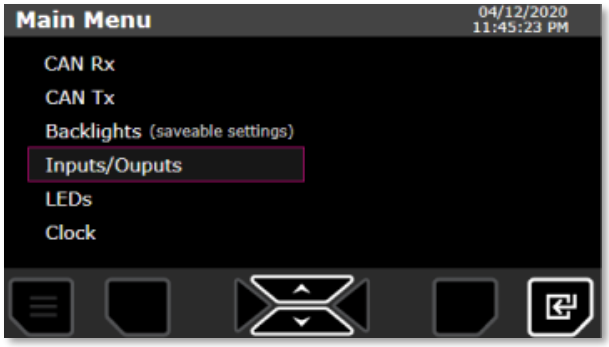

Figure 28

With *the MarlinDemo4_3Inch* project opened in TouchGFX Designer, refer to the *InputsOutputs* screen in the *Screens* tab on the left-hand side of the window. The associated source code files, which makes use of the I/O-related API and driver functions, are located in MarlinDemo4_3Inch\TouchGFX:

- \src\inputsoutputs_screen\ InputsOutputsView.cpp
- \include\gui\inputsoutputs_screen\ InputsOutputsView.hpp

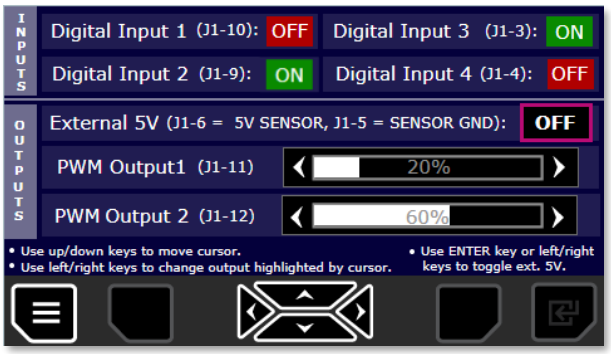

Figure 29

8. Tasks

When a TouchGFX project is created, certain tasks are already set up in the code. The implementation of these tasks can be used as a basis for user-defined tasks as well. The general structure for these tasks, with the text *task_name* used as a placeholder for the actual task names, is presented below. For user-defined tasks, simply replace *task_name* with the desired task name.

Tasks are initialized in main() (Src\main.cpp):

```
task_name_task_init(osPriorityNormal, 128);
```

Tasks are to be defined in their own files in **Src\tasks** and **Inc\tasks**. It is a good practice to include the name of the task in the file name as well as in the names of related functions and objects. A task file, **Src\tasks*task_name*_task.c**, would be created and would contain the following:

```
task_name_task_init(osPriorityNormal, 128);

void task_name_task_init(osPriority priority, uint32_t stack_size)
{
        osThreadDef_t thread_cfg = {"task_name_task", task_name_task, priority, 0, stack_size};
        task_name_task_handle = osThreadCreate(&thread_cfg, NULL);
        return;
}

static void task_name_task(void const *arg)
{
        for(;;)
        {
                //user code goes here

                osDelay(20);   // 20 milliseconds until task executed again
        }
        return;
}
```

Examples of tasks utilized in the sample application software *MarlinDemo4_3Inch* are discussed in section 10.1 CAN Rx and section 10.2 CAN Tx.

9. Semaphores

Semaphores are to be defined in their own files in **Src\semaphores** and **Inc\semaphores**. As with tasks, it is good practice to include the name of the semaphore in the file name and the names of related functions and objects. The general structure for semaphores used in the sample application code (*MarlinDemo4_3Inch*) is presented below, with the text *semaphore_name* used as a placeholder for the actual semaphore name. This example can be used as a reference for user-defined semaphores. In Src\semaphores*semaphoreName*_sem.c there would be the following:

```
static osSemaphoreId semaphoreName_sem;

int semaphoreNameSem_Init(void)
{
        osSemaphoreDef(semaphoreName);
        semaphoreName_sem = osSemaphoreCresemaphoreNameate(osSemaphore(semaphoreName), 1);
        osSemaphoreWait(semaphoreName, osWaitForever);

        return SEMAPHORE_SUCCESS;
}

int semaphoreNameSem_Post(void)
{
        osSemaphoreRelease(semaphoreName);
        return SEMAPHORE_SUCCESS;
}

int semaphoreName_Get(uint32_t ms)
{
        return (osSemaphoreWait(semaphoreName, ms)==osOK)?SEMAPHORE_SUCCESS:SEMAPHORE_ERR;
}
```

In the sample application, semaphore initialization functions (*semaphoreName*Sem_Init()) are called within the initialization of the task the semaphore is associated with. For example:

```
static void task_name_task(void const *arg)
{
        semaphoreNameSem_Init();      //placed before infinite for-loop
        for(;;)
        {
                //user code goes here
                osDelay(20);   // 20 milliseconds until task executed again
        }
        return;
}
```

Examples of semaphores utilized in the sample application software *MarlinDemo4_3Inch* are discussed in section 10.1 CAN Rx and section 10.2 CAN Tx.

10. CAN

Marlin CAN library functions are declared in **Inc\can_bus\marlin_can_lib.h**. Two of the functions, **CAN_Init()** and **CAN_Start()**, handle CAN initialization. When a TouchGFX project is created, calls to those functions are automatically placed within main(). **CAN_QueueMessage()** and **CAN_DequeueMessage()**, are used to send and receive CAN messages, and are discussed in section 10.1 CAN Rx and section 10.2 CAN Tx below. (**CAN_GetCountWaitingMessages()** currently has no functionality.)

Note: For reference the structs for received and transmitted CAN messages, referenced in section 10.1 CAN Rx and section 10.2 CAN Tx, are as follows (from Inc\can_bus\j1939_message_defines):

```
struct can_msg{
	CAN_RxHeaderTypeDef header;
	uint8_t data[8];
};

struct can_tx_msg{
	CAN_TxHeaderTypeDef header;
	uint8_t data[8];
};

struct __attribute__((packed)) j1939_header{
	uint8_t prio_and_reserved;
	union{
		uint16_t extended;
		struct{
			uint8_t specific;
			uint8_t format;
		}pdu;
	}pgn;
	uint8_t src;
};
```

10.1 CAN Rx

A CAN Rx task is automatically created for the user when a TouchGFX project is created. That includes initialization of the task in main(), via **user_can_task_init()**. That function is defined in an automatically generated file, **Src\tasks\user_can_task.c**. The file also includes the task function **user_can_task()**, in which the user can add code to process incoming CAN messages and responses:

```
static void user_can_task(void const *arg)
```

Use **CAN_DequeueMessage()** within user_can_task() to get a message from the CAN receive buffer:


```
enum marlinJ1939_error_codes CAN_DequeueMessage(struct can_msg *msg, uint32_t ms)
```

where `ms` is a timeout value, in milliseconds, for getting a message. In the code that is automatically generated, as well as the code in the sample application software *MarlinDemo4_3Inch*, this parameter is set to osWaitForever.

If a received message requires a response, call **CAN_QueueMessage()** to send it:

```
enum marlinJ1939_error_codes CAN_QueueMessage(struct can_tx_msg *msg);
```

The sample application software *MarlinDemo4_3Inch* demonstrates simple CAN Rx functionality on its *CAN Rx* screen (see Figure 31), which is accessible via the *CAN Tx* menu item in the *Main Menu (see* Figure 30*).*

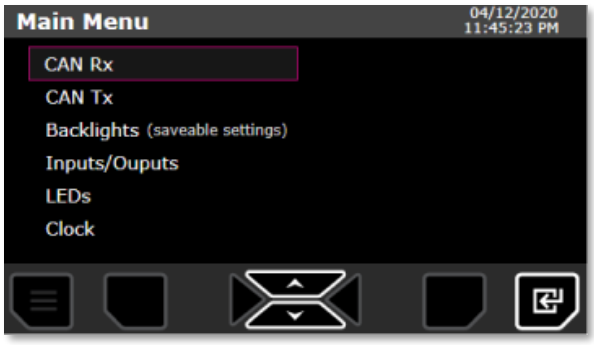

Figure 30

With *the MarlinDemo4_3Inch* project opened in TouchGFX Designer, refer to the *CAN_Rx* screen in the *Screens* tab on the left-hand side of the window. The associated source code, which makes use of the CAN-Rx-related task and semaphores, is located in MarlinDemo4_3Inch\TouchGFX\gui\:

- \src\can_rx_screen\CAN_RxView.cpp
- \include\gui\can_rx_screen\CAN_RxView.hpp

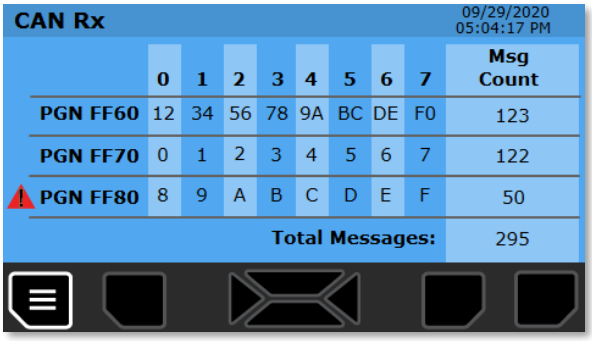

Figure 31

The screen shows the data bytes of received CAN, specifically those with PGNs 0xFF60, 0xFF70, or 0xFF80. (The source addresses in these messages does not matter, in this case.) Individual message counts and a total message count are also displayed.

The following functions were added to Src\tasks\user_can_task.c in order to display the various shown in Figure 31:


```
uint8_t* getRxMsg1()
uint8_t* getRxMsg2()
uint8_t* getRxMsg3()
uint32_t getRxMsg1Cnt()
uint32_t getRxMsg2Cnt()
uint32_t getRxMsg3Cnt()
uint32_t getTotalRxMsgCnt()
```

Each of these functions return data that is updated by the CAN Rx task function **user_can_task()** **(**user_can_task.c):

```
static void user_can_task(void const *arg).
```

The *CAN Rx* screen also flag indicates communication errors. If one of the three CAN messages displayed on the screen are not received withing 500 ms, that is considered a comm error. A comm error is represented on the screen by a red triangle with an exclamation point. One can be seen in Figure 31.

A comm error task is set up for each of the three messages displayed on the screen. See the task code in

- Src\tasks\comm_err1.c
- Src\tasks\comm_err2.c
- Src\tasks\comm_err3.c

Each of the comm error tasks utilizes its own semaphore, each of which is defined in the following files:

- Src\semaphores\comm_err1_sem.c
- Src\semaphores\comm_err2_sem.c
- Src\semaphores\comm_err3_sem.c

Each of the semaphore files contains functions similar to the ones described in section 9. Semaphores. For example, when user_can_task() receives PGN 0xFF60, it calls **comErr1Sem_Post()** to postpone flagging a comm error. Meanwhile, comm_err1_task() utilizes **commErr1Sem_Get ()** to continuously check if 500 ms has elapsed since the last PGN 0xFF60 was last received.


```
static void comm_err1_task(void const *arg)
{
    commErr1Sem_Init();

        for(;;)
        {
                if(SEM_IS_SUCCESS(commErr1Sem_Get(500)))
                {
                        commError1Active = 0;
                }
                else
                {
                        commError1Active = 1;
                }
        }
}
```

And finally, the code for the *CAN Rx* screen (TouchGFX\gui\src\can_rx_screen\CAN_RxView.cpp) determines whether to display the comm error icon (red triangle) by calling **getCommErr1Status()** to see if comm_err1_task() has flagged a comm error:

```
uint8_t getCommErr1Status()
{
        return commError1Active;
}
```


10.2 CAN Tx

As mentioned in section 10.1 CAN Rx, user_can_task can be set up to transmit messages in response to received CAN messages. However, periodic CAN messages are to be transmitted periodically, without regard to messages received, it is suggested that the user create a new task to handle such functionality. See section 8. Tasks for information about setting up a task.

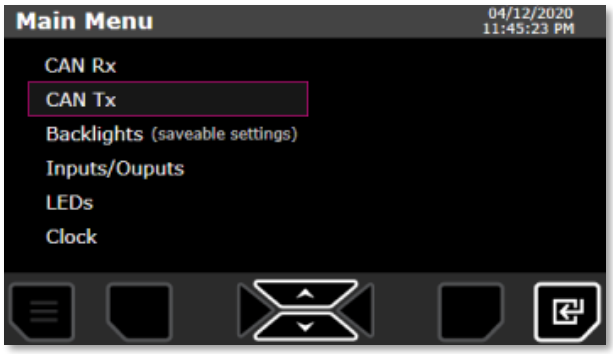

Figure 32

The user may also refer to the sample application software *MarlinDemo4_3Inch* for an example. The sample application implements simple CAN Tx functionality, as demonstrated on the *CAN Tx* screen (see Figure 33), which is accessible via the *CAN Tx* menu item in the *Main Menu* (see Figure 32).

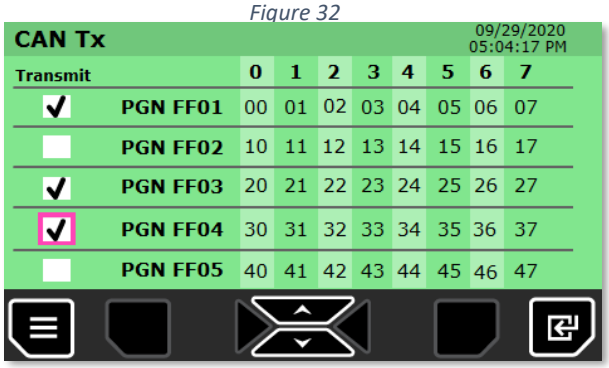

Figure 33

The user can move the cursor using the up- and down-arrow buttons. Pressing the enter button will add a checkmark to the box highlighted by the cursor, and will also trigger periodic transmission of the CAN message that is displayed in the row of the checked box.

With *the MarlinDemo4_3Inch* project opened in TouchGFX Designer, refer to the *CAN_Tx* screen in the *Screens* tab on the left-hand side of the window. The associated source code files, which makes use of the CAN-Tx-related task, are located in MarlinDemo4_3Inch\TouchGFX\gui:

- \src\can_tx_screen\CAN_TxView.cpp
- \include\gui\can_tx_screen\CAN_RxView.hpp

The sample application utilizes a CAN Tx task, which is defined in **Src\tasks\user_can_tx_task.c.** The file contains the following functions:

```
void user_can_tx_task_init(osPriority priority, uint32_t stack_size);
static void user_can_tx_task(void const *arg);
void enableCanTxMsg0(uint8_t status);
void enableCanTxMsg1(uint8_t status);
void enableCanTxMsg2(uint8_t status);
void enableCanTxMsg3(uint8_t status);
void enableCanTxMsg4(uint8_t status);
```


user_can_tx_task() utilizes **CAN_QueueMessage()** (also discussed in section 10.1 CAN Rx)**,** to send up to five CAN messages. For example, when the operator activates transmission of the PGN FF01 message, the screen code in CAN_TxView.cpp calls enableCanTxMsg0():

```
void enableCanTxMsg0(uint8_t status)
{
        canTxMsg0Enabled = status;
}
```

In turn, user_can_tx_task() checks if **canTxMsg0Enabled** has been set, and accordingly puts a PGN 0xFF01 CAN message into transmit buffer via **CAN_QueueMessage()**.

11. Clock

When a Marlin TouchGFX project is created, code is automatically generated to initialize the RTC on the microcontroller. Specifically, a call to MX_RTC_Init() (Src\rtc.c) is automatically included in main(). The function sets up an RTC object called `hrtc`:

```
RTC_HandleTypeDef hrtc;
```

To get and set the time and date, use the following functions:

```
HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTime, uint32_t Format);
HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTime, uint32_t Format);
HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef *hrtc, RTC_DateTypeDef *sDate, uint32_t Format);
HAL_StatusTypeDef HAL_RTC_GetDate(RTC_HandleTypeDef *hrtc, RTC_DateTypeDef *sDate, uint32_t Format);
```

For the first input parameter of the functions above, pass `hrtc` passed by reference. For `Format`, use `RTC_FORMAT_BIN` for binary values, `RTC_FORMAT_BCD` for binary coded decimal values. The other data types listed among the input parameters are `RTC_TimeTypeDef` and `RTC_DateTypeDef`, are defined in stm32f7xx_hal_rtc.h:

```
typedef struct
{
  uint8_t Hours;
  uint8_t Minutes;
  uint8_t Seconds;
  uint32_t SubSeconds;
  uint32_t SecondFraction;
  uint8_t TimeFormat;
  uint32_t DayLightSaving;
  uint32_t StoreOperation;
}RTC_TimeTypeDef;
```

```
typedef struct
{
  uint8_t WeekDay;
  uint8_t Month
  uint8_t Date
  uint8_t Year
}RTC_DateTypeDef;
```

See the definition of `RTC_TimeTypeDef` in stm32f7xx_hal_rtc.h for descriptions and value ranges of the struct members.

The sample application software *MarlinDemo4_3Inch* includes a screen (see Figure 35) that allows the operator to set the time and date. This screen is accessible via the *Clock* menu item in the *Main Menu* (see Figure 34)*:*

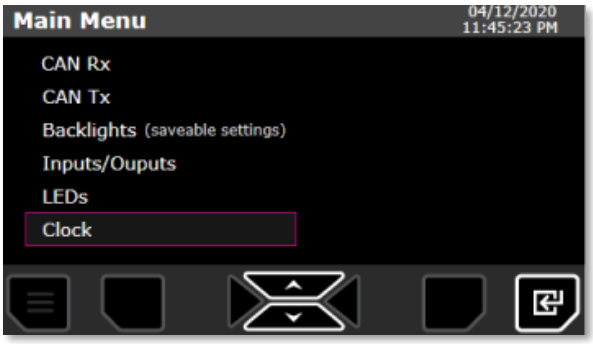

Figure 34

With *the MarlinDemo4_3Inch* project opened in TouchGFX Designer, refer to the *Clock* screen in the *Screens* tab on the left-hand side of the window. The associated source code files, which make use of the CAN-Tx-related task, are located in MarlinDemo4_3Inch\TouchGFX\gui\:

- \src\ clock_screen\ClockView.cpp
- include\gui\clock_screen\ClockView.hpp

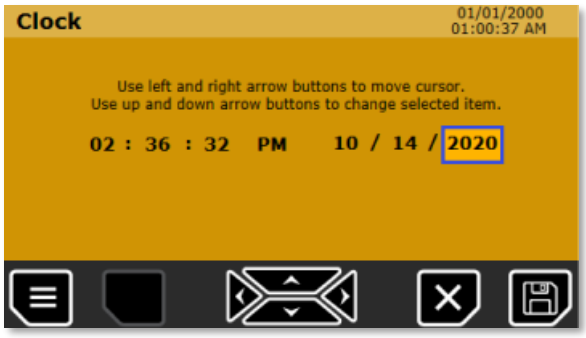

Figure 35

On this screen, the operator moves the cursor between individual parameters of the time and date, using the left- and right-arrow buttons. The time and date parameters are adjusted the up- and down-arrows. The time and date are saved to the RTC only when the operator presses the Save button, located under the disk icon displayed on the screen. The changes can be cancelled using the Cancel button, located under the X icon displayed on the screen. Pressing the Cancel button reverts the time and date parameters to the current date and time. Below is an example of how ClockView.cpp uses **HAL_RTC_GetTime()** and **HAL_RTC_GetDate()**:

```
HAL_RTC_GetTime(&hrtc, &currentTime, RTC_FORMAT_BIN);
HAL_RTC_GetDate(&hrtc, &currentDate, RTC_FORMAT_BIN);
```

And **HAL_RTC_SetTime()** and **HAL_RTC_SetDate()** (mentioned above), passing input parameters by reference:

```
HAL_RTC_SetTime(&hrtc, &newTime, RTC_FORMAT_BIN);
HAL_RTC_SetDate(&hrtc, &newDate, RTC_FORMAT_BIN);
```


12. Changing Application Software Identifiers

Below are customizable values and text in the source code that identify various aspects of the application software and display module. Such identifiers are typically be transmitted in CAN messages or displayed on the screen.

In …\Inc**bootloader_helper.h:**

- **PROJECT_NUMBER** is the part number of the display module. It is used in the J1939 address claim message (see above) and is displayed by the Marlin Programming Tool (see Figure 36).

```
#define PROJECT_NUMBER              505403
```

In …\Inc**J1939_DEF.h**:

- The following are used in name field of the **J1939 address claim message**:

```
// Application J1939 Name Field
 #define MODULE_IDENTIFIER           PROJECT_NUMBER      // typically Marlin model number
 #define MANUFACTURER_CODE           169          // use 169 for Marlin Technologies Inc
 #define ECU_INSTANCE                0            // use 0 for first/only instance
 #define FUNCTION_INSTANCE           0            // use 0 for only one function instance
 #define FUNCTION                    255          // use 255 for not available
 #define VEHICLE_SYSTEM              0            // use 0 for non-specific system
 #define VEHICLE_SYSTEM_INSTANCE     0            // use 0 for one instance
 #define INDUSTRY_GROUP              2            // use 2 for Agriculture & Forestry
 #define ARBITRARY_ADDRESS_CAPABLE   0            // use 0 for FALSE, 1 for TRUE
```

- **SW_NUMBER** is an ASCII string representing the part number for the application software.

```
#define SW_NUMBER       {'0', '1', '2', '8', '8', '7'}      // Marlin Software Number
```

- **SW_ID** is an ASCII string that indicates the revision of the application software (SW_NUMBER). It is displayed by the Marlin Programming Tool (see Figure 36).

```
#define SW_ID           {'$', 'A', 'X', '2', ' ', ' '}             // Software Revision
```

- **SOURCE_ADDRESS** is the source address of the display module, as transmitted in CAN messages. It is also displayed by the Marlin Programming Tool (see Figure 36).

```
#define SOURCE_ADDRESS              0xD0
```


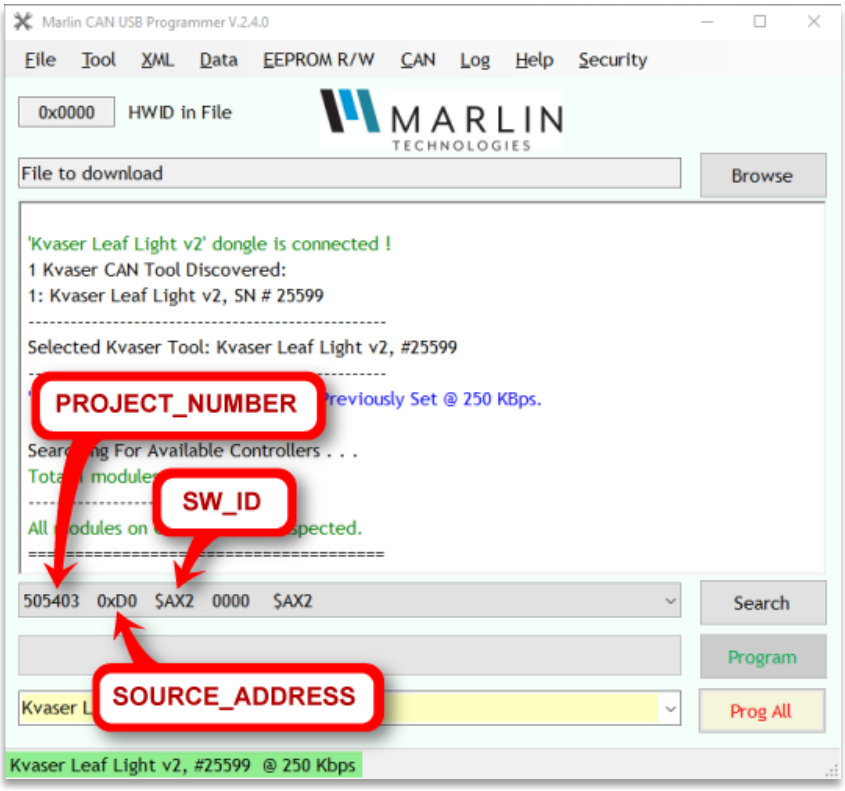

Figure 36

It is not recommended to change items in bootloader_helper.h and J1939_def.h, other than the ones mentioned above. Doing so can result in errors and unexpected functionality. For example, **MODULE_HWID** is value representing the type of hardware in the display module. It is expected to match the hardware ID in the display module's bootloader program.

```
#define MODULE_HWID                0x0299
```

If the IDs do not match, the Marlin Programming Tool will not readily program the module. In that case, the operator will need to force the module will need to be forced into bootloader mode to enable it to be programmed again. See section 13. Troubleshooting Tips for more information.

13. Troubleshooting Tips

1. Certain build errors and unexpected functionality may be resolved by deleting the *build* folder (in the application's top-level folder) prior to clicking Run Target in TouchGFX Designer. This forces a clean build of the application, in case TouchGFX Designer fails to rebuild a portion of the application that the user has modified since the previous build. Remember to back up any necessary files (such as .s19 files) before deleting the *build* folder.

2. If there is an error in the user application that prevents the display module from communicating with the Programming Tool, the display can be forced into bootloader mode and then reprogrammed. To enter bootloader mode, first power down the display module. Then power up the module while holding down the following three buttons:

Figure 37

When the display is in bootloader mode, the screen is black and the blue or red LEDs are turned on.

3. If "Hardware ID Mismatch!" is displayed by the Programming Tool, check that MODULE_HWID in Inc\bootloader_helper.h is defined as follows:

```
#define MODULE_HWID                0x0299
```

The hardware ID defined in the application program must match the hardware ID of the bootloader, which is 0x0299. The Programming Tool checks for matching hardware IDs to ensure that compatible software modules are programmed onto modules.

Appendix: Giving a Button the Ability to Navigate to More Than One Possible Screen

On the *Main Menu* screen in the sample application software *MarlinDemo4_3Inch* (see Figure 38), pressing UP and DOWN buttons moves a cursor through the list of screens, and pressing the Enter button loads the screen that is highlighted by the cursor.

If a hardware button is to trigger a transition to one of several possible screens, it will take more than a single TouchGFX "interaction" to implement that functionality. The following steps were taken to set up the Enter button functionality in the sample application:

Figure 38

1. An interaction was created in TouchGFX Designer in which the Enter button triggers, not a screen transition, but a virtual function called handleEnterButton() (see Figure 39). This virtual function is hand-coded, as described in a later step, and it ultimately handles the transition to the appropriate screen.

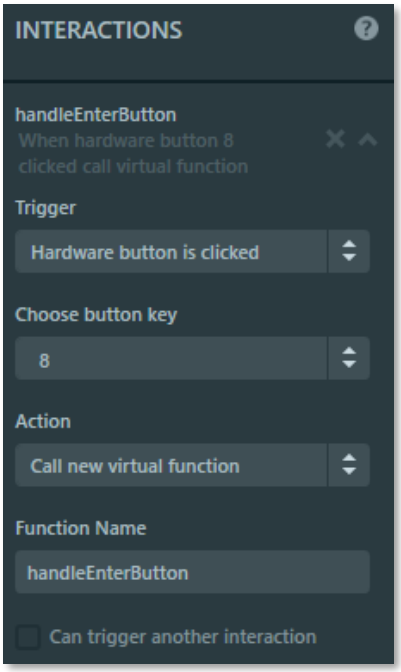

Figure 39

2. For each menu item in the Main Menu, a dummy button was created by dragging a touchscreen button from the widget panel (see Figure 40) to an area of the canvas that is outside the active screen.

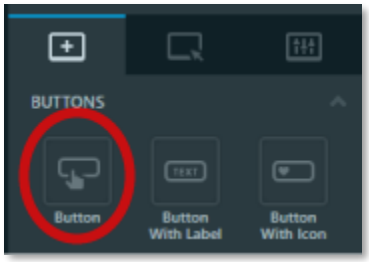

Figure 40

3. Only the names of the dummy buttons need be changed; the appearance and other properties of the buttons do not matter because the buttons will not be visible on the screen. With a dummy button selected, its name can be changed on the Settings tab (see Figure 41) located on the right side of the TouchGFX Designer window. These are "dummy" buttons because they are merely being used to generate code that can be used by the virtual function to change screens (not to mention that the Marlin display does not support touch screen buttons).

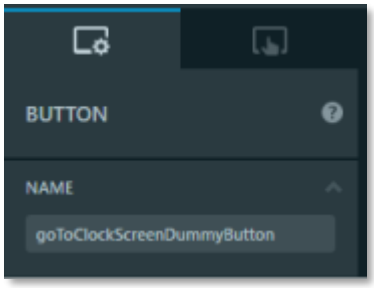

Figure 41

4. For each dummy button created, an interaction was created to link the dummy button to a screen transition. For example, see goToClockScreen under the list of interactions for the MainMenu screen. As shown in Figure 42, goToClockScreen uses goToClockScreenDummyButton to set a transition to the *Clock* screen.

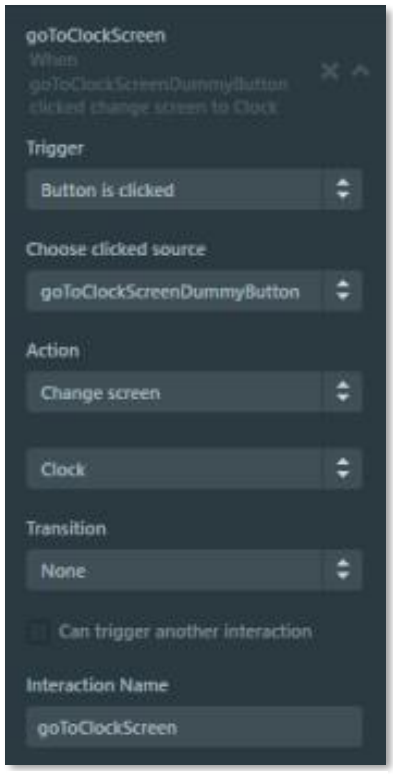

Figure 42

5. The Generate Code button was then clicked in the upper-right corner of the TouchGFX Designer window (see Figure 43).

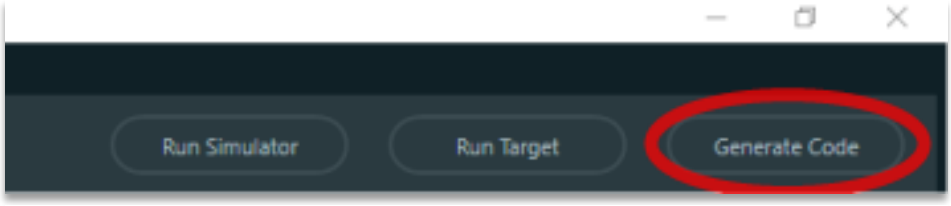

Figure 43

This generated a code function based on each the screen-transition interactions mentioned above. The functions generated are declared in MarlinDemo4_3Inch \ TouchGFX \ generated \ gui_generated \ include \ gui_generated \ common \ FrontendApplicationBase.hpp (see highlighted text below):

```
class FrontendApplicationBase : public touchgfx::MVPApplication
{
public:
    FrontendApplicationBase(Model& m, FrontendHeap& heap);
    virtual ~FrontendApplicationBase() { }

    // CAN_Rx
    void gotoCAN_RxScreenNoTransition();

    // MainMenu
    void gotoMainMenuScreenNoTransition();

    // LEDs
    void gotoLEDsScreenNoTransition();

    // Backlights
    void gotoBacklightsScreenNoTransition();

    // CAN_Tx
    void gotoCAN_TxScreenNoTransition();

    // InputsOutputs
    void gotoInputsOutputsScreenNoTransition();

    // Clock
    void gotoClockScreenNoTransition();
```

Note that the function names are derived from the screen name plus the type of screen transition defined for it: goto***<name of screen>***Screen***<screen transition type>.*** For example, for the interaction shown in Figure 42, the function goto**Clock**Screen**NoTransition** was generated. These functions will be called inside the virtual function handleEnterButton() in the next step **Note:** The function names are *not* based on the name of the interaction they are derived from, despite the

similarities between such names in this example. Be sure to get the exact function name by checking FrontendApplicationBase.hpp.

6. Next, the virtual function handleEnterButton(), called out in Figure 39, was added to the appropriate source code files. Specifically, its definition was added to MarlinDemo4_3Inch \ TouchGFX \ gui \ src \ mainmenu_screen \ **MainMenuView.cpp** as follows:

```
void MainMenuView::handleEnterButton()
{
        switch(cursorPosition)
        {
                case MENU_ITEM_CAN_RX:
                        application().gotoCAN_RxScreenNoTransition();
                        break;

                case MENU_ITEM_CAN_TX:
                        application().gotoCAN_TxScreenNoTransition();
                        break;

                case MENU_ITEM_BACKLIGHTS:
                        application().gotoBacklightsScreenNoTransition();
                        break;

                case MENU_ITEM_INPUTS_OUTPUTS:
                        application().gotoInputsOutputsScreenNoTransition();
                        break;

                case MENU_ITEM_LEDS:
                        application().gotoLEDsScreenNoTransition();
                        break;

                case MENU_ITEM_CLOCK:
                        application().gotoClockScreenNoTransition();
                        break;
                default:
                        break;
        }
}
```

Note that the functions highlighted above match those declared in FrontendApplicationBase.hpp. These function calls change effectively change the screen to the one highlighted by the cursor as the Enter button is pressed. Be sure to include the text preceding the function name to avoid compile errors.

Note: The function handleEnterButton() is able to determine which menu item that the cursor is highlighting because of the cursorPosition variable. The interactions handleUpButton and handleDownButton are set up in TouchGFX Designer to trigger virtual functions of the same names

when the UP or DOWN buttons are pressed. Those virtual functions (defined in MainMenuView.cpp) update the cursorPosition variable accordingly.

The virtual function's declaration was added to MarlinDemo4_3Inch \ TouchGFX \ gui \ include \ gui \ mainmenu_screen \ **MainMenuView.hpp** as follows:

```
class MainMenuView : public MainMenuViewBase
{
.
.
.
protected:
.
.
.
    virtual void handleEnterButton();
};
```

