ARM MKE1xF MCU Replatform
ARM MKE1xF MCU
Technical Manual

Team: Nathan Hong, Derek Lung, Japsimran Singh, Bevin Tang
Advisor: Paul Hummel
Last Updated: June 15, 2018
Special thanks to Dr. Hummel, Nick Mah, Hunter Borlik, Alvin Ha, Chris Adams and the entire Cal Poly Racing FSAE team.
**Table of Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>6</td>
</tr>
<tr>
<td>Requirements</td>
<td>7</td>
</tr>
<tr>
<td>Board</td>
<td>7</td>
</tr>
<tr>
<td>MCU</td>
<td>7</td>
</tr>
<tr>
<td>Software</td>
<td>7</td>
</tr>
<tr>
<td>System Overview</td>
<td>8</td>
</tr>
<tr>
<td>Hardware Design</td>
<td>9</td>
</tr>
<tr>
<td>Overall System Overview</td>
<td>9</td>
</tr>
<tr>
<td>MCU System Overview</td>
<td>10</td>
</tr>
<tr>
<td>Analog and Digital Power</td>
<td>11</td>
</tr>
<tr>
<td>CAN</td>
<td>11</td>
</tr>
<tr>
<td>USB</td>
<td>12</td>
</tr>
<tr>
<td>Connector</td>
<td>12</td>
</tr>
<tr>
<td>BOM</td>
<td>12</td>
</tr>
<tr>
<td>Anti-Aliasing Filter</td>
<td>14</td>
</tr>
<tr>
<td>Background</td>
<td>14</td>
</tr>
<tr>
<td>Design Considerations</td>
<td>14</td>
</tr>
<tr>
<td>Solution</td>
<td>14</td>
</tr>
<tr>
<td>Schematic</td>
<td>15</td>
</tr>
<tr>
<td>Results</td>
<td>16</td>
</tr>
<tr>
<td>MCUXpresso IDE</td>
<td>18</td>
</tr>
<tr>
<td>Flashing the board</td>
<td>18</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>19</td>
</tr>
<tr>
<td>MCUXpresso Config</td>
<td>20</td>
</tr>
<tr>
<td>Choosing Pinouts</td>
<td>22</td>
</tr>
<tr>
<td>Choosing Clocks</td>
<td>24</td>
</tr>
<tr>
<td>CAN</td>
<td>26</td>
</tr>
<tr>
<td>Usage</td>
<td>26</td>
</tr>
<tr>
<td>CANLIB::Init()</td>
<td>26</td>
</tr>
<tr>
<td>CANLIB::CAN_GET_TX_HANDLE(CANLISTNER* callback)</td>
<td>26</td>
</tr>
<tr>
<td>CANLIB::CAN_TX_Message(CANHandle handle, const CAN_DATA_PACK* frame)</td>
<td>26</td>
</tr>
<tr>
<td>CANLIB::flexcan_callback(CAN_Type *base, flexcan_handle_t *handle, status_t status, uint32_t result, void *userData)</td>
<td>26</td>
</tr>
<tr>
<td>CANLIB::find_free_MB(CAN_Type* base)</td>
<td>26</td>
</tr>
</tbody>
</table>
Sample Code 26

**Timer** 28

Usage 31
- Timer::GetStaticClass() 31
- Timer::Init(uint32_t period, bool * update) 31
- Timer::TimerStart() 31
- Timer::TimerPause() 31
- Timer::TimerResume() 31
- Timer::Update(); 31
- BOARD_FTM_HANDLER(); 31

**ADC** 32

ADCManger Library:
- ADCManager::Init() 37
- ADCManager::ADCAvailable() 37
- ADCManager::StartRead() 37
- ADCManager::Finished() 37

ADCManger Callback Interface:
- Pinout: 39
- Clock Configuration: 39
- Calibration Data: 40

Troubleshooting 41

**UART** 42

Usage 42
- UARTManager::GetStaticClass() 42
- UARTManager::Init() 42
- UARTManager::writeString() 42
- UARTManager::writeInteger() 42

UART Serial Messaging:
- On Linux: 42
- On Windows: 42

Pinout 42

**SPI** 43

Signal Descriptions 43
Parameter Register (LPSPIx_PARAM) 44
Control Register (LPSPIx_CR) 44
Configuration Register (LPSPIx_CFGRO) 46
SPi Master 47
## Function Overview

Definitions 47  
Configuration 48

## SPI Slave

Function Overview 49  
Definitions 50  
Configuration 51

## Sources

- Kinetis KE1xF Sub-Family Reference Manual 52  
- SDK Download 52
Abstract

After Cal Poly Racing’s electrical team began to hit the technical limits of the ADC and other I/O features of the current 8-bit Atmel AT90 microcontroller unit, it became clear that an upgrade was due. This replatforming project takes the functionalities of the old, 8-bit architecture, and aims to provide a 32-bit version using the ARM MKE1xF MCU. With the idea of having a working PCB as a stretch goal, the scope of the library development was limited to enable base functionality. Thus, the only libraries developed were for the Timer, ADC, SPI, UART, and CAN. Additionally, this document discusses the software and hardware development processes, as well as details on how to use specific components of the newly developed MCU platform. With this upgrade, the platform should be capable of supporting a diverse feature set to meet the needs of many future projects to come.
Requirements

**Board**
- Select 5V tolerant MCU
- Strain Gauge Amplifier circuit
- Analog Anti-Aliasing Filtering circuit

**MCU**
- Minimum of 8 Analog Sensor inputs
- 1 Mbps CAN Bus speed
- ADC
- 10 Bit minimum accuracy
- 1.5 kHz minimum individual channel sampling
- USB/Serial

**Software**
- New library implementation for 32-bit MCU
- CAN support
- USB/Serial
- I2C/SPI
- ADC
System Overview

The main functionality of the MCU is to operate as a data acquisition platform that serves to acquire external sensor voltage inputs between 0 to 5V. The platform can package sensor data and transmit them to other devices over CAN. It will also act as a layer of abstraction from the lower level codebase that directly interacts with the MCU. This abstracted interface allows individuals that may not be familiar with the Kinetis MCU to quickly pick up, learn, and use the platform.
Hardware Design

In order to deliver a functional prototype a PCB implementing the 64LQFP version of the Kinetis MKE16F was designed. This design implemented a few major components:

- FTDI for USB
- JTAG Programmer Connection
- CAN Transceiver
- ADC Channels
- Vias for probing GPIO Pins
- LDO for Analog components
- LDO for Digital components

Overall System Overview

Figure 1 shows the overall system overview which contains four major blocks: Power, MCU, USB, and CAN
**MCU System Overview**

NXP offers some advice on integrating their MCUs into schematics. Using a combination of their datasheet/reference manual as well as the already created development board schematic we were able to correctly design the overall MCU schematic.
**Analog and Digital Power**

Because this board contains both analog and digital systems, we chose to separate both the analog and digital power and grounds to eliminate noise issues that arise in mixed signal designs.

![Figure 3: Board Power Schematic](image)

**CAN**

Previously in Cal Poly Racing designs, the MCP2551 was used as a CAN transceiver. However, recently Microchip phased this chip out and now recommends the newer MCP2561 transceiver. For component availability, we chose to use the newer MCP2561 chip in our design. The only major noticeable difference with this chip is lower current consumption.

![Figure 4: CAN Communication Schematic](image)
**USB**

A commonly used FTDI chip was chosen for UART communication. A USB Micro B connector was used instead of the USB Mini B connectors the team used in the past. This was motivated by more USB Micro cable availability.

![Figure 5: USB Communication Schematic](image)

**Connector**

A AMP superseal connector was chosen for creating the PCB to harness connection. This is a commonly used connector on team projects as it provides both a TPA for contacts and is waterproof.

![Figure 6: AMP Superseal Connector Layout](image)
## BOM

Bill of materials referenced with the Altium Schematic Design

<table>
<thead>
<tr>
<th>Comment</th>
<th>Description</th>
<th>Designator</th>
<th>Footprint</th>
<th>LibRef</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1μF</td>
<td>0.1μF Capacitor</td>
<td>C1, C2, C3, C13, C14, C15, C18, C20</td>
<td>CAP SMT 0603</td>
<td>Capacitor 0.1μF 25V 0603</td>
<td>8</td>
</tr>
<tr>
<td>100 pF</td>
<td>100 pF Capacitor</td>
<td>C16</td>
<td>CAP SMT 0603</td>
<td>Capacitor 100 pF 25V 0603</td>
<td>1</td>
</tr>
<tr>
<td>0.22μF</td>
<td>0.22μF Capacitor</td>
<td>C17, C19</td>
<td>CAP SMT 0603</td>
<td>Capacitor 0.22μF 25V 0603</td>
<td>2</td>
</tr>
<tr>
<td>NUP2105LT1G</td>
<td>Dual Line CAN Bus Protector, 3-Pin SOT-23, Pb-Free, Tape and Reel</td>
<td>D1</td>
<td>ONSC-SOT-23-3-318-08_V</td>
<td>CMP-1063-0016-4-1</td>
<td>1</td>
</tr>
<tr>
<td>APG1608CGKC/T</td>
<td>LED</td>
<td>DS1, DS3, DS4</td>
<td>GREEN LED</td>
<td>Green LED</td>
<td>3</td>
</tr>
<tr>
<td>APG1608SUKC/T</td>
<td>LED</td>
<td>DS2</td>
<td>RED LED</td>
<td>RED LED</td>
<td>1</td>
</tr>
<tr>
<td>ZX62R-B-5P</td>
<td>Connector; USBMicro-B, Reverse type, SMT; 5 Position; Right Angle</td>
<td>J1</td>
<td>HIRO-ZX62R-B-5P_V</td>
<td>CMP-0031-0000-1-1</td>
<td>1</td>
</tr>
<tr>
<td>HDR2x5</td>
<td>2x5 Jtag</td>
<td>J3</td>
<td>HDR 2x5</td>
<td>HDR2x5</td>
<td>1</td>
</tr>
<tr>
<td>1k</td>
<td>1k Resistor</td>
<td>R1, R2, R3, R4</td>
<td>RES SMT 0805</td>
<td>Resistor 1k 1% 0805</td>
<td>4</td>
</tr>
<tr>
<td>10k</td>
<td>10k Resistor</td>
<td>R9</td>
<td>RES SMT 0805</td>
<td>Resistor 10k 1% 0805</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>120 Resistor</td>
<td>R10</td>
<td>RES SMT 0805</td>
<td>Resistor 120 1% 0805</td>
<td>1</td>
</tr>
<tr>
<td>470</td>
<td>470 Resistor</td>
<td>R11, R12</td>
<td>RES SMT 0805</td>
<td>Resistor 470 1% 0805</td>
<td>2</td>
</tr>
<tr>
<td>FT232RL-Reel</td>
<td>USB UART Asynchronous Serial Data Transfer Chip, SSOP-28, Tape and Reel</td>
<td>U1</td>
<td>SSOP-28_L</td>
<td>CMP-0078-0002-9-1</td>
<td>1</td>
</tr>
<tr>
<td>MKE16F512VLH16</td>
<td>MKE15F MCU</td>
<td>U7</td>
<td>MKE16F512VLH16</td>
<td>MKE16F512VLH16</td>
<td>1</td>
</tr>
<tr>
<td>MCP2561-E/ SN</td>
<td>High-Speed CAN Transceiver, 8-Pin SOIC, Extended Temperature</td>
<td>U8</td>
<td>SOIC-SN8_N</td>
<td>CMP-0186-0003-8-2</td>
<td>1</td>
</tr>
<tr>
<td>LM340MP-5.0</td>
<td>Series 3-Terminal Positive Regulators, 4-pin SOT-223</td>
<td>U9, U10</td>
<td>DCY0004A_M</td>
<td>CMP-0062-0176-5-3</td>
<td>2</td>
</tr>
<tr>
<td>TE Superseal 26 POS</td>
<td>Connector</td>
<td>X1</td>
<td>TE_SUPERSEAL_26</td>
<td>TE Superseal 26 POS</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 1: Bill of Materials
Anti-Aliasing Filter

Background

This project focused on the collection of data from shock pot sensors. These sensors give off a reading proportional to the frequency at which the car's suspension moves the electrical contact within the sensor. This data is helpful in monitoring the condition and use of the suspension.

The purpose of the filter is to eliminate the frequency of road noise so that the PCB does not have to waste resources processing data that will not be used; this noise typically sits in the 800 - 1300 Hz range, while the events the team cares about (like when the car takes a turn) sit in the 0 - 200 Hz range.

Design Considerations

Ultimately, a lowpass filter is necessary to meet the team’s needs. Over the years, many innovations of filters have been created, but all of them have their special cases -- often excelling at a specific task at the cost of another. For the team's purposes, the configuration chosen should perform with 0 gain and minimal phase shift while in the operating frequency (up to about 700 Hz).

Solution

To hit these requirements, the team designed a 2nd order Sallen Key lowpass filter. Sallen-Key filters are designed to achieve great performance before their cutoff frequency. By using a Sallen-Key design with a high quality factor, the filter can achieve 0 gain and phase shift for lower frequencies.

The tradeoff for the Sallen-Key's performance is its behavior near the cutoff frequency; at the cutoff, a quality factor of 3 equates to about a 10db gain. Additionally, as the signal reaches a decade within the cutoff frequency, the nice 0 phase shift deteriorates before completely flipping at the cutoff frequency -- changing the phase shift from 0 to -180 degrees.

The filter was designed to have relatively uniform phase under the range of the incoming sensor data (0-200Hz). To do this Q is set to be high (3). This created a resonant peak that amplifies signals around 700Hz. This gain must be dealt with digitally after sampling. Sampling must be done at or above about 2.2 kHz to ensure aliasing does not occur.

Filter Type: Sallen-Key
Cutoff Freq: 700 Hz
-3dB Freq: ~1.1 kHz
Quality Factor: 3
Figure 7: LTspice Schematic of newly designed Sallen-Key filter

Figure 8: 2nd order lowpass Sallen-Key filter simulated behavior

Op Amp Chosen: LT1057 Dual and Quad, JFET Input Precision High Speed Op Amp

 +/- 10V Regulator: MAX680
**Results**

The following oscilloscope screenshots depict comparisons of the input signal and the output signal using the filer. Channel 1 is the output (top, orange) and channel 2 in the input.

*Figure 9: Filter output comparison at 100 Hz*

*Figure 10: Filter output comparison at 200 Hz*
From the figures, one can tell that the phase shift is negligible for all frequencies ranging within the operating frequency (100 - 1000Hz). However, once outside that range, the phase drastically changes (by -180 degrees).
**MCUXpresso IDE**


**Flashing the board**

The board can be flashed using the Kinetis IDE software and the P&E UMultilink.

Build the program as an `.axf` file and save it. This can be done by selecting the build icon's drop down menu and change it from debug to release.

Select the *Program Flash* icon.

Select the USB1-Multilink Universal probe.
Under program flash memory, select the compiled .axf file. This file should be created when the release build was created. Use the browse button to locate it.

Flash the program to the board by selecting ok. The board should be flashed with the program and will begin to run.

**Troubleshooting**

**Probe not detected:**
If your probe is not detected when trying to flash it might not be enabled. Navigate to Windows -> Preferences -> MCUXpresso IDE -> Debug Probe Discovery and enable the SEGGER J-Link probes.
MCUXpresso Config

The MCUXpresso tool can easily configure the pinout of the Kinetis chipset.

- **Launch** the *MCUXpresso Config Tools v4.0* program

  ![Screenshot of the MCUXpresso Config Tools v4.0 application]

- Create a new configuration for processor, board, or kit.
  - **Select** *File -> New...*

  ![Screenshot of the new configuration window in the MCUXpresso Tools v4.0 program]
Select Create new configuration for processor, board, or kit -> Next

Select Processors -> Kinetis E -> New MKE16F256xxx16 configuration -> Name the configuration to preference -> Choose the MKE16F256VLH16-LQFP64 package -> Finish
The program should now display something similar to this

![Image: MCUXpresso Configuration Pinout]

**Figure 13: MCUXpresso Configuration Pinout**

**Choosing Pinouts**

Depending on the peripherals the user wants to use, different pins can be routed to different peripherals.

- Select the peripherals the user wants to enable. *(Red)*
- Select all the required pins for the peripherals. *(Blue)*
- After all the pins needed are selected, the code that enable those pins will be in the generated code preview. The user can simply copy and overwrite their pin_mux.c and pin_mux.h file or use the export button on the top right to push the code to the project folder. *(Green)*

![Image: MCUXpresso Configuration Pinout Selection]

**Figure 14: MCUXpresso Configuration Pinout Selection**
Some pins may be shared between multiple peripherals. If a single pin is selected for multiple peripherals then it will become red. Resolve this issue by selecting an alternate pin so that there is no conflict between peripherals.

Figure 15: Generated Code Preview for Pin Mux
Choosing Clocks

If a specific clock speed is needed for a peripheral, then using the clock tool can easily enable that clock.

- Switch the view from peripheral to clock by selecting the Clock Config tab.
- Choose the tab for clock diagram.
- On the top toolbar, select Run Mode to RUN.
- On the top toolbar, select Functional Group to be BOARD_BootClockInit.
- On the top toolbar, select the SCG Mode to be the desired clock.
- Select the peripheral and enable them to run on the specified clock of the SCG Mode.

Figure 16: Peripheral Clock
If the clock needs to be divided, find the clock signal and the corresponding clock divider. Select the clock divider module and enable it. Then select the divider ratio and the clock configuration code will be generated.

After the desired configuration is create, export the generated code by using the export button which will overwrite the clock_config files.

![Figure 17: Peripheral Clock Divider](image)
CAN

Usage

CANLIB::Init()
Initializes the CAN controller by calling the NXP CAN libary to get the default configuration, setup the clock source, and make the CAN transfer handle.

To use, call the static class instance of the function: CANLIB::StaticClass().Init();

CANLIB::CAN_GET_TX_HANDLE(CANLISTNER* callback)
Returns the transmit handle that will be used to send the transmit the CAN message

To use, call the static class instance of the function:

```
CANLIB::StaticClass().CAN_GET_TX_HANDLE(nullptr);
```

CANLIB::CAN_TX_Message(CANHandle handle, const CAN_DATA_PACK* frame)
CAN transmit function that completes the data transfer. Requires a CAN handle and a frame which consists of the CAN id, DLC, and payload.

To use, call the static class instance of the function:

```
CANLIB::StaticClass().CAN_TX_Message( handle, const_cast<const
CANLIB::CAN_DATA_PACK*>(&test))
```

CANLIB::flexcan_callback(CAN_Type* base, flexcan_handle_t* handle, status_t status, uint32_t result, void *userData)
C based callback function that checks whether a message was transmitted or received and calls the corresponding C++ based callback function.

CANLIB::find_free_MB(CAN_Type* base)
Function that finds a free CAN Message Object and returns it. Used by the CAN_TX_Message() function.

Sample Code

```c
#define CAN_TESTING

#include "board.h"
#include "peripherals.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "MKE18F16.h"
#include "timer.h"
#include "Globals.h"

#include "ADCManager.h"
```
#include "SensorManager.h"

extern void timer1_init();

int main() {
    // Init board hardware.
    BOARD_InitBootPins();
    BOARD_InitBootClocks();

    // Init FSL debug console.
    BOARD_InitDebugConsole();

    // Setup CAN
    CANLIB::StaticClass().Init();

    CANLIB::CAN_DATA canTmp = {};

    // Define sample data
    int16_t data = 1010;

    ((uint16_t *) & canTmp.bytes)[0] = data;

    // Create sample CAN message object
    CANLIB::CAN_DATA_PACK test;

    test.id = 0xC5;
    test.dlc = 2;
    test.frame_data = canTmp;

    CANLIB::CANHandle handle = CANLIB::StaticClass().CAN_Get_TX_Handle(nullptr);

    while (true) {
        // Toggle pins for testing
        if (led == 0) {
            GPIO_PortClear(BOARD_LED_RED_GPIO, 1 U << BOARD_LED_RED1_GPIO_PIN);
            led = 1;
        } else {
            GPIO_PortSet(BOARD_LED_RED_GPIO, 1 U << BOARD_LED_RED1_GPIO_PIN);
            led = 0;
        }

        CANLIB::StaticClass().CAN_Transmit(handle, const_cast <
            const CANLIB::CAN_DATA_PACK * >( & test));

        printf("Sending CAN message\n");

        Globals::StaticClass()._delay(300);
    }
}


**Timer**

The Kinetis chipset has a Flex timer that supports a 16-bit counter. The timer can interrupt every 1ms and is in charge of handling the timing for all the sensors.

**Figure 18: FTM Status and Control Registers from KE1xFP100M168SF0RM Reference Manual**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-28</td>
<td>Reserved field read only field which fields is always 0.</td>
</tr>
<tr>
<td>27-24</td>
<td>Filter Prescaler</td>
</tr>
<tr>
<td><strong>FLTPS</strong></td>
<td>0000 Divide by 1</td>
</tr>
<tr>
<td></td>
<td>0001 Divide by 2</td>
</tr>
<tr>
<td></td>
<td>0010 Divide by 3</td>
</tr>
<tr>
<td></td>
<td>0011 Divide by 4</td>
</tr>
<tr>
<td></td>
<td>0100 Divide by 5</td>
</tr>
<tr>
<td></td>
<td>0101 Divide by 6</td>
</tr>
<tr>
<td></td>
<td>0110 Divide by 7</td>
</tr>
<tr>
<td></td>
<td>0111 Divide by 8</td>
</tr>
<tr>
<td></td>
<td>1000 Divide by 9</td>
</tr>
<tr>
<td>Bit Range</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>23-16</td>
<td>PWEMEN7 - PWEMEN0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>15-10</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>TOF</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>TOIE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RF</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>RIE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>Reload interrupt is disabled.</td>
<td>1 Reload interrupt is enabled.</td>
</tr>
<tr>
<td>5 CPWMS</td>
<td>Center-Aligned PWM Select</td>
</tr>
<tr>
<td>Selects CPWM mode. This mode configures the FTM to operate in Up-Down Counting mode. This field is write protected. It can be written only when MODE[WPDIS] = 1.</td>
<td>0 FTM counter operates in Up Counting mode. 1 FTM counter operates in Up-Down Counting Mode.</td>
</tr>
<tr>
<td>4-3 CLKS</td>
<td>Clock Source Selection</td>
</tr>
<tr>
<td>Selects one of the three FTM counter clock sources</td>
<td>00 No clock selected. This in effect disables the FTM counter. 01 FTM input clock 10 Fixed frequency clock 11 External clock</td>
</tr>
<tr>
<td>2-0 PS</td>
<td>Prescale Factor Selection</td>
</tr>
<tr>
<td>Selects one of 8 divisions factors for the clock source selected by CLKS. The new prescaler factor affects the clock source on the next FTM input clock cycle after the new value is updated into the register bts,</td>
<td>000 Divide by 1 001 Divide by 2 010 Divide by 4 011 Divide by 8 100 Divide by 16 101 Divide by 32 110 Divide by 64 111 Divide by 128</td>
</tr>
</tbody>
</table>

Table 2: FTM Status and Control Registers Field Descriptors from KE1xFP100M168SF0RM Reference Manual
Usage

Timer::GetStaticClass()
Get an instance of the flex timer. The user does not need to declare the object to use it.

Timer::Init(uint32_t period, bool * update)
Initialize the timer with a period in milliseconds. The timer will count based on the FTM timer. Once the count overflows, an interrupt is called and the timer is reset.

Timer::TimerStart()
Start the timer by enabling FTM interrupt.

Timer::TimerPause()
Pause the timer by disabling FTM interrupt

Timer::TimerResume()
Re-enable the timer by enabling FTM interrupt.

Timer::Update();
Function that updates the user that the allotted time has passed. Reset the timer count and restart the timer.

BOARD_FTM_HANDLER();
The flex timer interrupt handler. This function is called when the FTM counter overflows the amount of ticks needed to trigger a 1ms interrupt. The interrupt function will proceed to call the update function and reset the timer.

Clock Configuration:

The Flex timer is configured to use the system phase-locked loop (SPLL) clock running at 96 MHz. FTM0 uses the SPLL DIV1 with a divider ratio of 1 so the timer runs at 96 MHz.

![Figure 19: Flex Timer Clock Select](image)
The 64LQFP package supports up to 16 external analog input channels on ADC0 and support up to 11 channels on ADC1 and ADC2. The ADC function at a 8 to 12-bit resolution with the ability to take multiple sample counts to return accurate and precise readings.

The ADC is configured through changing certain bits stored in registers. The status and control register 1 holds information that informs the user if data is ready to be read and allows the user to configure the ADC to be interrupt based and also what channel to read.

**Figure 20: ADC Status and Control Register 1 from KE1xFP100M168SF0RM Reference Manual**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>Reserved field read only field which fields is always 0.</td>
</tr>
</tbody>
</table>
| 7     | Conversion Complete Flag  
        This is a read-only field that is set each time a conversion is completed.  
        0 Conversion is not completed.  
        1 Conversion is completed.  
        COCO is cleared when one of the following is true:  
        - The respective SC1 channel register is written.  
        - The respective data channel is read. |
| 6     | Interrupt Enable  
        Enabling this bit enable conversion complete interrupts. When COCO is high and AIEN is high, an interrupt is asserted.  
        0 Conversion complete interrupt is disabled.  
        1 Conversion complete interrupt is enabled. |
Table 3: ADC Status and Control Register 1 Field Descriptors from KE1xFP100M168SF0RM Reference Manual

The ADC configuration register can be modified to configure the respective ADC. This register allows the user to configure the clock divider, what resolution to read at, and which internal clock to use.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-7</td>
<td>Reserved field read only field which fields is always 0.</td>
</tr>
</tbody>
</table>
| 6-5   | Clock Divide Select  
2 bits determine which divide rationis used by the ADC to generated the internal clock ADCK  
00 divide input by 1  
01 divide input by 2  
10 divide input by 4  
11 divide input by 8 |
| 4     | Reserved field read only field which fields is always 0. |
| 3-2   | Conversion mode selection  
Selects the ADC resolution |
The second configuration register allows the user to change the sample time between 2 to 256. The ADC will sample the input that amount of times and average them as the result. More samples will give a more accurate ADC reading but will take more time and the opposite will be faster but less accurate.

Table 4: ADC Configuration Register 1 Field Descriptors from KE1xFP100M168SF0RM Reference Manual

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00-0</td>
<td>ADC conversion format</td>
</tr>
<tr>
<td>00 8-bit conversion</td>
<td>01 12-bit conversion              10 10-bit conversion 11 Reserved</td>
</tr>
<tr>
<td>1-0</td>
<td>ADICLK</td>
</tr>
<tr>
<td>Input clock select</td>
<td>Selects the input clock source to generate the internal clock. The input clock will be divided by the selected ADIV. 00 Alternate clock 1 (ADC_ALTCLK1) 01 Alternate clock 2 (ADC_ALTCLK2) 10 Alternate clock 3 (ADC_ALTCLK3) 11 Alternate clock 4 (ADC_ALTCLK4)</td>
</tr>
</tbody>
</table>

The second configuration register allows the user to change the sample time between 2 to 256. The ADC will sample the input that amount of times and average them as the result. More samples will give a more accurate ADC reading but will take more time and the opposite will be faster but less accurate.

Table 5: ADC Configuration Register 2 Field Descriptors from KE1xFP100M168SF0RM Reference Manual

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>Reserved field read only field which fields is always 0.</td>
</tr>
<tr>
<td>7-0</td>
<td>SMPLTS</td>
</tr>
<tr>
<td>Sample Time Select</td>
<td>Selects a sample time of 2 to 256 ADCK clock cycles. Note that a sample time of 1 is not supported. Lower sample time will maximize conversion speed for lower impedance inputs and higher sample times will increase the accuracy of high impedance inputs.</td>
</tr>
</tbody>
</table>
The averaged ADC value will be stored in the data result registers and is read only.

<table>
<thead>
<tr>
<th>Conversion mode</th>
<th>D11</th>
<th>D10</th>
<th>D9</th>
<th>D8</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-bit single-ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Unsigned right-justified</td>
</tr>
<tr>
<td>10-bit single-ended</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit single-ended</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 23: ADC Data Result Register Description from KE1xFP100M168SF0RM Reference Manual*

The kinetis comes with a driver file that lets the user modify the ADC register without having to directly change the bits.

**There are two structures that allow the user to easily configure the ADC:**

```c
typedef struct _adc12_config {
    adc12_reference_voltage_source_t referenceVoltageSource; /* Set the reference voltage source. */
    adc12_clock_source_t clockSource;         /* Set the input clock source to converter. */
    adc12_clock_divider_t clockDivider;       /* Set the divider of input clock source. */
    adc12_resolution_t resolution;            /* Set the sample resolution mode. */
    uint32_t sampleClockCount;                /* Set the sample clock count. Add its value may improve the stability of the conversion result. */
    bool enableContinuousConversion;          /* Enable continuous conversion mode. */
} adc12_config_t;

typedef struct _adc12_channel_config {
    uint32_t channelNumber;                   /* Setting the conversion channel number. The available range is 0-31. See channel connection information for each chip in Reference Manual document. */
    bool enableInterruptOnConversionCompleted; /* Generate an interrupt request once the conversion is completed. */
} adc12_channel_config_t;
```

*Figure 24: Fsl_adc12 Driver Structures from fsl_adc12.h*

Creating an instance of the `adc12_config` structure and changing the members of the structure as specified from ADC Field Descriptor tables will allow the user to configure the ADC without having to directly write to the memory location.

These functions are useful in initializing the ADC and configuring them to the desired state.

**ADC Configuration:**

**ADC Channel Configuration:**

```c
ADC12_GetDefaultConfig(adc12_config_t * adc12ConfigStruct)
Set the default configuration struct.

ADC12_Init(ADC_Type * base, adc12_config_t * adc12ConfigStruct)
Initialize the ADC base with the configured structure.

ADC12_SetHardwareAverage(ADC_Type * base, adc12_config_t * adc12ConfigStruct)
```
ADC12_EnableHardwareTrigger(ADC_Type * base, uint8_t bool)

ADC12_DoAutoCalibration(ADC_Type * base)

**Modifies the ADC channel registers:**

ADC12_SetChannelConfig(ADC_Type * base,
adc12_config_t * adc12ConfigStruct,
uint32_t channelGroup,
const adc12_channel_config_t * config)

**Get the ADC data value in the data register and clear the COCO flag:**

ADC12_GetChannelConversionValue(ADC_Type * base, uint32_t channelGroup)

**The MKE18F16 header file contains the overall register structures:**

```c
typedef struct {
  __IO uint32_t SCl[8];                        //**< ADC Status and Control Register 1, array offset: 0x0, array step: 0x4 */
  __IO uint32_t CFA1;                          //**< ADC Configuration Register 1, offset: 0x00 */
  __IO uint32_t CFA2;                          //**< ADC Configuration Register 2, offset: 0x4 */
  __IO uint32_t B[8];                          //**< ADC Data Result Registers, array offset: 0x48, array step: 0x4 */
  __IO uint32_t CV1;                           //**< Compare Value Register, offset: 0x8 */
  __IO uint32_t CV2;                           //**< Compare Value Register, offset: 0x8 */
  __IO uint32_t TSC2;                          //**< Status and Control Register 2, offset: 0x8 */
  __IO uint32_t TSC3;                          //**< Status and Control Register 3, offset: 0x8 */
  __IO uint32_t BASE_OFS;                      //**< BASE Offset Register, offset: 0x8 */
  __IO uint32_t OFS;                           //**< ADC Offset Correction Register, offset: 0x8 */
  __IO uint32_t USB_OFS;                       //**< USER Offset Correction Register, offset: 0x8 */
  __IO uint32_t XOF5;                          //**< ADC X Offset Correction Register, offset: 0x8 */
  __IO uint32_t YOF5;                          //**< ADC Y Offset Correction Register, offset: 0x8 */
  __IO uint32_t G;                            //**< ADC Gain Register, offset: 0x8 */
  __IO uint32_t UG;                            //**< ADC User Gain Register, offset: 0x8 */
  __IO uint32_t CLPK;                          //**< ADC General Calibration Value Register 5, offset: 0x8 */
  __IO uint32_t CLP2;                          //**< ADC Plus-Side General Calibration Value Register 5, offset: 0x8 */
  __IO uint32_t CLP;                           //**< ADC Plus-Side General Calibration Value Register 2, offset: 0x8 */
  __IO uint32_t CLPI;                          //**< ADC Plus-Side General Calibration Value Register 1, offset: 0x8 */
  __IO uint32_t CLPB;                          //**< ADC Plus-Side General Calibration Value Register 0, offset: 0x8 */
  __IO uint32_t CLPK2;                         //**< ADC Plus-Side General Calibration Value Register X, offset: 0x8 */
  __IO uint32_t CLPKL;                         //**< ADC Plus-Side General Calibration Value Register 9, offset: 0x8 */
  __IO uint32_t CLPKL_OFS;                     //**< ADC General Calibration Offset Value Register 5, offset: 0x8 */
  __IO uint32_t CLPK2_OFS;                     //**< ADC Plus-Side General Calibration Offset Value Register 5, offset: 0x8 */
  __IO uint32_t CLPK2_OFS;                     //**< ADC Plus-Side General Calibration Offset Value Register 2, offset: 0x8 */
  __IO uint32_t CLPKL_OFS;                     //**< ADC Plus-Side General Calibration Offset Value Register 1, offset: 0x8 */
  __IO uint32_t CLPKL_OFS;                     //**< ADC Plus-Side General Calibration Offset Value Register 0, offset: 0x8 */
  __IO uint32_t CLPK2_OFS;                     //**< ADC Plus-Side General Calibration Offset Value Register X, offset: 0x8 */
  __IO uint32_t CLPKL_OFS;                     //**< ADC Plus-Side General Calibration Offset Value Register 9, offset: 0x8 */
  // All other registers...
} ADC_Type;
```

*Figure 25: ADC Register Structure from MKE18F16.h*
**ADCManager Library:**
To simplify and interface the driver with the overall higher level logic, a new ADC library with simplified functions is implemented. This API allows the high level user to get ADC readings without having to manually map all the registers.

**ADCManager::GetStaticClass()**
Get an instance of the ADC Manager. The user does not need to declare the object to use it.

**ADCManager::Init()**
The initialization function configures the ADC memory registers and enables interrupts for the ADC. No channel configurations are created here.

**ADCManager::ADCAvailable()**
Check if the ADC has been initialized before reading. This simply checks if the pointer to the function is null or not. This function is not called by the user but by the internal API.

**ADCManager::StartRead()**
Sets up the channel configuration structure to begin a read.

Start reading on the specified channel using interrupts to get the updated value. Once an interrupt occurs, the interrupt will call the callback interface and begin the user specified function. Refer to the ADCManager Callback Interface section to create a proper callback.

If the ADC is not available then this function will do nothing.

**ADCManager::Finished()**
Calls the user specified function when there is an interrupt. Refer to the ADCManager Callback Interface section to create a proper callback.
**ADCManager Callback Interface:**

The ADC Manager must be used with a callback interface that the user must implement. This callback should be implemented to hold the data result. This allows the ADCManager to have a common interface for the sensors.

```cpp
class test : public ADCManagerCallbackInterface {
    public:
        bool conversionComplete = false;
        uint16_t value;
        uint8_t channel;
        void INT_Call_ADC_Finished(const uint16_t& value, uint8_t channel) {
            this->value = value;
            this->channel = channel;
            conversionComplete = true;
        }
};
```
**Pinout:**
The ADC on the MCU is able to support up to 8 channels. The external inputs can be selected using the MCUXpresso Configuration tool.

![Figure 26: ADC Pinout](image)

**Clock Configuration:**
The ADC is configured to use the system phase-locked loop (SPLL) clock running at 96 MHz. ADC0 uses the SPLL DIV1 with a divider ratio of 1 so the timer runs at 96 MHz.

![Figure 27: ADC Clock Select](image)
**Calibration Data:**
The Kinetis MCU has hardware calibration for the ADC. After running it, the ADC readings can still be off. A software calibration function has been created by taking data through 50 samples of data from 0V to 5V.

![Figure 28: Actual Voltage vs ADC Voltage](image)

Using the data, the difference between the ADC voltage output and the actual voltage can be calculated. This offset was graphed and a linear graph was created.

![Figure 29: ADC Voltage Offset of Actual Voltage per Voltage](image)
Based on this offset, the calibration formula for the offset was generated:

\[
\text{OFFSET} = 0.00295 \times \text{ADC VOLTAGE VALUE} + 0.22
\]

\[
\text{ADC CALIBRATED VALUE} = \left( \text{ADC VOLTAGE VALUE} \times 500 \right) / 4096 + \text{OFFSET}
\]

The calibrated ADC value will be read as an integer value that is in volts.

**Troubleshooting**

**No readings are detected:**
- Make sure the sensors are attached to the correct pins which are specified by board the init pins function. Use the MCUXpresso Configuration tool to ensure the correct pins are assigned to the ADC.
- Make sure the ADC is running on the specified clock from the clock configuration.
- Make sure interrupts are not disabled before calling `StartRead()`.
UART

A simple UART is implemented for serial communication. The user can access two functions that allow them to write strings and integers to serial. The UART runs at a baud rate of 115200.

Usage

UARTManager::GetStaticClass()
Get an instance of the UART manager.

UARTManager::Init()
Initialize the UART to interrupt once every 1ms.

UARTManager::writeString()
A function that takes a const char * and writes a string through serial.

UARTManager::writeInteger()
A function that takes an integer and writes each digit through serial.

UART Serial Messaging

On Linux:
Open a bash terminal and run screen /dev/ttyACM0 115200. Messages should be written through serial to the screen. The MCU might be on a different port than ACM0 so configure it correctly.

On Windows:
Use a serial communication program like putty and connect to the serial device.

Pinout

Figure 30: UART Pinout
SPI

SPI does not needed to be implemented for the ARM MCU platform. Implementation for SPI is made available in the case that it will be needed in the future for additional interfacing. Therefore a stand alone interrupt based implementation of *Master* and *Slave* will be included in the code base separate from the main ARM MCU platform library. It is designed to be easy to customize and modular as use for SPI is currently undefined.

Signal Descriptions

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCK</td>
<td>Serial clock. Input in slave mode, output in master mode.</td>
<td>VO</td>
</tr>
<tr>
<td>PCS[0]</td>
<td>Peripheral Chip Select. Input in slave mode, output in master mode.</td>
<td>VO</td>
</tr>
<tr>
<td>PCS[1] / HREQ</td>
<td>Peripheral Chip Select or Host Request. Host Request pin is selected when HREN=1 and HRSEL=0. Input in either slave mode or when used as Host Request, output in master mode.</td>
<td>VO</td>
</tr>
<tr>
<td>SOUT / DATA[0]</td>
<td>Serial Data Output. Can be configured as serial data input signal. Used as data pin 0 in quad-data and dual-data transfers.</td>
<td>VO</td>
</tr>
<tr>
<td>SIN / DATA[1]</td>
<td>Serial Data Input. Can be configured as serial data output signal. Used as data pin 1 in quad-data and dual-data transfers.</td>
<td>VO</td>
</tr>
</tbody>
</table>

*Table 6: SPI Signal Descriptions*
**Parameter Register (LPSPIx_PARAM)**

Address: Base address + 4h offset

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved</td>
<td>This field is reserved. This read-only field is reserved and always has the value 0.</td>
</tr>
<tr>
<td>15-8</td>
<td>RXFIFO</td>
<td>Receive FIFO Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The number of words in the receive FIFO is $2^\text{RXFIFO}$.</td>
</tr>
<tr>
<td></td>
<td>TXFIFO</td>
<td>Transmit FIFO Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The number of words in the transmit FIFO is $2^\text{TXFIFO}$.</td>
</tr>
</tbody>
</table>

**Control Register (LPSPIx_CR)**

Address: Base address + 10h offset

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>15-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>DGEN</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>DOZEN</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RST</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MEN</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 31: Parameter register field descriptions**

**Figure 32: Control Register field descriptions**
## LPSPx.CR field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-10 Reserved</td>
<td>This field is reserved. This read-only field is reserved and always has the value 0.</td>
</tr>
</tbody>
</table>
| 0 RRF | Reset Receive FIFO  
0 No effect.  
1 Receive FIFO is reset. |
| 8 RTF | Reset Transmit FIFO  
0 No effect.  
1 Transmit FIFO is reset. |
| 7-4 Reserved | This field is reserved. This read-only field is reserved and always has the value 0. |
| 3 DBGEN | Debug Enable  
0 Module is disabled in debug mode.  
1 Module is enabled in debug mode. |
| 2 DOZEN | Doze mode enable  
Enables or disables Doze mode  
0 Module is enabled in Doze mode.  
1 Module is disabled in Doze mode. |
| 1 RST | Software Reset  
Reset all internal logic and registers, except the Control Register. Remains set until cleared by software.  
0 Master logic is not reset.  
1 Master logic is reset. |
| 0 MEN | Module Enable  
0 Module is disabled.  
1 Module is enabled. |

*Table 7: Control Register field descriptions*
# Configuration Register (LPSPIx_CFGRO)

**Address:** Base address + 20h offset

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31–10</td>
<td>Reserved. This field is reserved.</td>
</tr>
<tr>
<td></td>
<td>This read-only field is reserved and always has the value 0.</td>
</tr>
</tbody>
</table>
| 9 | RDMO | Receive Data Match Only  
When enabled, all received data that does not cause DMF to set is discarded. Once DMF is set, the RDMO configuration is ignored. When disabling RDMO, clear RDMO before clearing DMF to ensure no receive data is lost.  
0  | Received data is stored in the receive FIFO as normal.  
1  | Received data is discarded unless the DMF is set.  |
| 8 | CIRFIFO | Circular FIFO Enable  
When enabled, the transmit FIFO read pointer is saved to a temporary register. The transmit FIFO will be emptied as normal, but once the LPSPi is idle and the transmit FIFO is empty, the read pointer value will be restored from the temporary register. This will cause the contents of the transmit FIFO to be cycled through repeatedly.  
0  | Circular FIFO is disabled.  
1  | Circular FIFO is enabled.  |
| 7–3 | Reserved | This field is reserved.  
This read-only field is reserved and always has the value 0. |
| 2 | HRSEL | Host Request Select  
Selects the source of the host request input. When the host request function is enabled with the LPSPi_HREQ pin, the LPSPi_PCS[1] function is disabled.  
0  | Host request input is pin LPSPi_HREQ.  
1  | Host request input is input trigger.  |
| 1 | HRPOL | Host Request Polarity  
| | 

*Figure 33: Configuration register bit*
Table 8: Configuration Register field descriptions

### SPI Master

**Function Overview**

The master is set to do the following

1. Initialize
   ```c
   void master_init(lpspi_master_config_t masterConfig)
   ```
2. Populate TX buffer for data transfer
   ```c
   void master_transfer_set_up(void)
   ```
3. Enable interrupts
   ```c
   static inline status_t EnableIRQ(IRQn_Type interrupt)
   ```
4. Send/Receive data through ISR
   ```c
   void LPSPSI_MASTER_IRQHandler(void)
   ```
5. Check for data transfer errors
   ```c
   void master_check_transfer(void)
   ```
6. Terminate
   ```c
   void LPSPSI_Deinit(LPSPSI_Type *base)
   ```
Definitions

```c
/* Definitions
 *--------------------------------------------------------------------------*/
#define LPSPI_MASTER_BASEADDR LPSPI0 // peripheral LPSPI0 base pointer
#define LPSPI_MASTER_CLOCK KCLK_Lpspi0 // LPSPI0 specific clock name
#define LPSPI_MASTER_CLK_FREQ (CLOCK_GetIpFreq(LPSPI_MASTER_CLOCK))

#define LPSPI_MASTER_CLOCK_SOURCE KCLK_IpsrcFircAsync
// Clock options
// KCLK_IpsrcNameOrExt = BU, /* Clock is off or external clock is used. */
// KCLK_IpsrcSysOscAsync = IU, /* System oscillator async clock. */
// KCLK_IpsrcFircAsync = IU, /* Slow IRC async clock. */
// KCLK_IpsrcPllAsync = IU, /* Fast IRC async clock. */
// KCLK_IpsrcSysPllAsync = IU /* System PLL async clock. */

#define LPSPI_MASTER_PCS_PER_INPT LPSPI_Pcs2
// PCS (Peripheral Chip Select) options
// LPSPI_PcsB = BU, /* PCSB */
// LPSPI_Pcs1 = IU, /* PCS1 */
// LPSPI_Pcs2 = IU, /* PCS2 */
// LPSPI_Pcs3 = IU /* PCS3 */

#define LPSPI_MASTER_PCS_PER_TRANSFER LPSPI_MasterPcs2
// Master PCS (Peripheral Chip Select) signal
// LPSPI_MasterPcsB = BU << LPSPI_MASTER_PCS_SHIFT, /* LPSPI master transfer use PCSB signal */
// LPSPI_MasterPcs1 = IU << LPSPI_MASTER_PCS_SHIFT, /* LPSPI master transfer use PCS1 signal */
// LPSPI_MasterPcs2 = IU << LPSPI_MASTER_PCS_SHIFT, /* LPSPI master transfer use PCS2 signal */
// LPSPI_MasterPcs3 = IU << LPSPI_MASTER_PCS_SHIFT, /* LPSPI master transfer use PCS3 signal */

#define LPSPI_MASTER_IRQ LPSPI0_IRQ // serial peripheral interface 0 (LPSPI0) interrupt
#define LPSPI_MASTER_IRQHandler LPSPI0_IRQHandler // IRQ function address
#define TRANSFER_SIZE 0x4 // transfer data size of 4 bytes
#define TRANSFER_BAUDRATE 500000U // transfer baudrate
```

Figure 34: Definitions and their alternate configuration options that are used in SPI master
Configuration

```c
// set clock source for LPSPI and get master clock source
CLOCK_SetSrc(LPSPI_MASTER_CLOCK, LPSPI_MASTER_CLOCK_SOURCE);

/* master config */
masterConfig.baudRate = TRANSFER_BAUDRATE;
masterConfig.bitsPerFrame = 8;

// polarity
masterConfig.cpol = LPSPI_ClockPolarityActiveHigh;
// LPSPI_ClockPolarityActiveHigh = 0U; /* CPOL-0. Active-High LPSPI clock (Idle low)*/
// LPSPI_ClockPolarityActiveLow = 1U; /* CPOL-1. Active-Low LPSPI clock (Idle high)*/

// phase
masterConfig.cpha = LPSPI_ClockPhaseFirstEdge;
// LPSPI_ClockPhaseFirstEdge = 0U; /* CPHA-0. Data is captured on the leading edge of the SCK and changed on the following edge.*/
// LPSPI_ClockPhaseSecondEdge = 1U; /* CPHA-1. Data is changed on the leading edge of the SCK and captured on the following edge.*/

// endian-ness
masterConfig.direction = LPSPI_MODEFirst;
// LPSPI_MODEFirst = 0U; /* Data Transfers start with most significant bit.*/
// LPSPI_MODELast = 1U; /* Data Transfers start with least significant bit.*/

masterConfig.portDelay = 1000000000U; /* masterConfig.baudRate;*/
masterConfig.bitDelay = 1000000000U; /* masterConfig.baudRate;*/
masterConfig.DataTransferDelay = 1000000000U; /* masterConfig.baudRate;*/
masterConfig whichPcs = LPSPI_MASTER_PCS_FOR_INIT;

// set which edge for PC to be active on
masterConfig.pcsActiveHighLow = LPSPI_PcsActiveHigh;
// LPSPI_PcsActiveHigh = 1U; /* Pcs Active High (Idle low) */
// LPSPI_PcsActiveLow = 0U; /* Pcs Active Low (Idle high) */

// set pin direction
masterConfig.pinConfig = LPSPI_SDIInAndSDOOut;
// LPSPI_SDIInAndSDOOut = 0U; /* LPSPI SDI input, SDO output. */
// LPSPI_SDOInAndSDIOut = 1U; /* LPSPI SDI input, SDO output. */
// LPSPI_SDIOutAndSDIIn = 2U; /* LPSPI SDO input, SDI output. */
// LPSPI_SDIInAndSDIOut = 3U; /* LPSPI SDI input, SDO output. */

masterConfig.dataOutConfig = LPSPI_DataOutWrited;
// LPSPI_DataOutWrited = 0U; /* Data out retains last value when chip select is de-asserted */
// LPSPI_DataOutWrited = 1U; /* Data out is tristated when chip select is de-asserted */

// set master clock
LPSPI_MasterInit(LPSPI_MASTER_BASEADDR, masterConfig, LPSPI_MASTER_CLK_FREQ);
```

Figure 35: Master SPI configuration and alternate configuration options

**SPI Slave**

**Function Overview**

The slave is set to do the following

1. Initialize
   ```c
   void slave_init(lpspi_slave_config_t slaveConfig)
   ```

2. Populate TX buffer for data transfer
   ```c
   void slave_transfer_set_up(void)
   ```

3. Enable interrupts
   ```c
   static inline status_t EnableIRQ(IRQn_Type interrupt)
   ```

4. Send/Receive data through ISR
   ```c
   void LPSPI_SLAVE_IRQHandler(void)
   ```
5. Check for data transfer errors
   void slave_check_transfer(void)
6. Terminate
   void LPSPI_Deinit(LPSPI_Type *base)

Definitions

```c
/* Definitions
 ************************************************************/
#define LPSPI_SLAVE_BASEADDR LPSPI0 // peripheral LPSPI0 base pointer
#define LPSPI_SLAVE_IRQN LPSPI0_IRQn // serial peripheral interface 0 (LPSPI0) interrupt
#define LPSPI_SLAVE_IRQHandler LPSPI0_IRQn // IRQ function address

#define LPSPI_SLAVE_PCS_FOR_INIT kLPSPI_Pcs2
   // PCS options
   // kLPSPI_Pcs0 = 0U, /* PCS[0] */
   // kLPSPI_Pcs1 = 1U, /* PCS[1] */
   // kLPSPI_Pcs2 = 2U, /* PCS[2] */
   // kLPSPI_Pcs3 = 3U /* PCS[3] */

#define LPSPI_SLAVE_PCS_FOR_TRANSFER kLPSPI_SlavePcs2
   // Slave PC options
   // kLPSPI_SlavePcs0 = 0U << LPSPI_SLAVE_PCS_SHIFT, /* LPSPI slave transfer use PCS0 signal */
   // kLPSPI_SlavePcs1 = 1U << LPSPI_SLAVE_PCS_SHIFT, /* LPSPI slave transfer use PCS1 signal */
   // kLPSPI_SlavePcs2 = 2U << LPSPI_SLAVE_PCS_SHIFT, /* LPSPI slave transfer use PCS2 signal */
   // kLPSPI_SlavePcs3 = 3U << LPSPI_SLAVE_PCS_SHIFT, /* LPSPI slave transfer use PCS3 signal */

#define LPSPI_SLAVE_CLOCK_NAME kCLOCK_Lpspi0 // LPSPI0 specific clock name

#define LPSPI_SLAVE_CLOCK_SOURCE kCLOCK_IpsrcFlrcAsync
   // kClock options
   // kCLOCK_IpsrcNoneOrExt = 0U, /* Clock is off or external clock is used. */
   // kCLOCK_IpsrcSysOscAsync = 1U, /* System Oscillator async clock. */
   // kCLOCK_IpsrcsIrCAsync = 2U, /* Slow IRC async clock. */
   // kCLOCK_IpsrcFircAsyn = 3U, /* Fast IRC async clock. */
   // kCLOCK_IpsrcSySrPllAsync = 6U /* System PLL async clock. */

#define TRANSFER_SIZE 64U // transfer data size of 8 bytes
```

Figure 36: Definitions and their alternate configuration options that are used in SPI slave
Configuration

```c
CLOCK_Set_SRC1((LPSPI_SLAVE_CLOCK_NAME, LPSPI_SLAVE_CLOCK_SOURCE));

    // polarity
    slaveConfig.cpol = LPSPI_ClockPolarityActiveHigh;
    // LPSPI_ClockPolarityActiveHigh = 0, /* CPOL=0, Active-High LPSPI clock (i.e., low) */
    // LPSPI_ClockPolarityActiveLow = 1, /* CPOL=1, Active-Low LPSPI clock (i.e., high) */

    // phase
    slaveConfig.cpha = LPSPI_ClockPhaseFirstEdge;
    // LPSPI_ClockPhaseFirstEdge = 0, /* CPHA=0, Data is captured on the leading edge of the SCK and changed on the following edge */
    // LPSPI_ClockPhaseSecondEdge = 1 /* CPHA=1, Data is changed on the leading edge of the SCK and captured on the following edge */

    // end
    slaveConfig.direction = LPSPI_MSBFirst;
    // LPSPI_MSBFirst = 0, /* Data transfers start with most-significant-bit */
    // LPSPI_LSBFirst = 1 /* Data transfers start with least significant bit */

    slaveConfig.address = LPSPI_SLAVE_ADDRESS);

    // set which line for PC to be active on
    slaveConfig.pcsActiveHigh = LPSPI_PCSActiveHigh;
    // LPSPI_PCSActiveHigh = 0, /* PCS Active High (i.e., low) */
    // LPSPI_PCSActiveLow = 1 /* PCS Active Low (i.e., high) */

    // set pin direction
    slaveConfig.pinCfg = (LPSPI_SDINSDOUT);
    // LPSPI_SDINSDOUT = 0, /* LPSPI SDO input, SSI output */
    // LPSPI_SDINSDOUT = 1 /* LPSPI SDO input, SSI output */
    // LPSPI_SDINSDOUT = 2 /* LPSPI SDO input, SSI output */
    // LPSPI_SDINSDOUT = 3 /* LPSPI SDO input, SSI output */

    slaveConfig.dataOutConfig = LPSPIDataOutResetValue;
    // LPSPIDataOutResetValue = 0, /* Data out retains last value when CSn select is de-asserted */
    // LPSPIDataOutResetValue = 1 /* Data out is tri-stated when CSn select is de-asserted */

LPSPI_SlaveInit(LPSPI_SLAVE_BASEADDR, &slaveConfig);
```

**Figure 37: Slave SPI configuration and alternate configuration options**
Sources

Github Repository
https://github.com/CalPolyFSAE/Kinetis-ARM-MCU.git

Kinetis KE1xF Sub-Family Reference Manual

SDK Download
https://mcuxpresso.nxp.com/en/welcome