Application Note

AN_347

FT51A Test and Measurement Sample

Version 1.1

Issue Date: 2015-11-26

This document provides a guide for using the FT51A development environment to read sensors and send results to a host PC via the USB Test and Measurement Class.

Future Technology Devices International Limited (FTDI)
Unit 1, 2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777 Fax: + 44 (0) 141 429 2758
Web Site: http://ftdichip.com

Table of Contents

1 Introduction

This application note documents an example firmware project for the FT51A. The source code is available in the "examples\AN_347_Test_and_Measurement_Sample" folder of the FT51A Software Development Kit.

1.1 Overview

The Test and Measurement firmware and LabView application demonstrate a method for providing data to LabView from an FT51A device. The FT51A need only be connected to a host PC via the USB interface. The firmware responds to test and measurement class (TMC) requests and message protocol to send readings from sensors to LabView. An application written in LabView on the host PC will generate the TMC messages and process the responses. The readings from sensors are displayed on the host PC.

The TMC class firmware will decode and respond to a subset of SCPI commands which are delivered over USB by the TMC class.

The example code also includes the DFU functionality from AN_344 FT51A DFU Sample.

1.2 Features

The test and measurement example has the following features:

- Open source firmware layered on the FT51A USB Library.
- Implements a bulk IN and a bulk OUT endpoint pair.
- Reads data from a temperature sensor using the SPI Master interface.
- Converts the analogue voltage from a force sensor to a digital reading.
- Detects transitions on an analogue voltage input to make a simple heart rate monitor.

The built-in USB hub on the FT51A is enabled in this example to allow up-to 4 FT51A EVM modules to be chained together.

1.3 Limitations

The firmware does not implement a wide range of SCPI commands.

The firmware is designed for the FT51A EVM module. It can utilise the LCD for user feedback with communications on the I2C Master bus. This is disabled by default as program space is limited – the code to support it is conditionally compiled. (See section 3)

The size of the firmware code to support the TMC class and DFU class leaves very little space in the program data area for expansion.

1.4 Scope

The guide is intended for developers who are creating applications, extending FTDI provided applications or implementing example applications for the FT51A.

In the reference of the FT51A, an "application" refers to firmware that runs on the FT51A; "libraries" are source code provided by FTDI to help user, access specific hardware features of the chip.

The FT51A Tools are currently only available for Microsoft Windows platform and are tested on Windows 7 and Windows 8.1.

2 Test and Measurement Overview

The protocol and requirements for test and measurement devices are documented in the TMC specification:

http://www.usb.org/developers/docs/devclass_docs/USBTMC_1_006a.zip

The FT51A implementation is for a USB device which decodes and responds to a small number of SCPI commands.

Configuration descriptors for TMC devices have an interface descriptor containing a bulk IN and a bulk OUT endpoint. They have a USB "Application" class and a subclass identifying the device as Test and Measurement. The USB protocol indicates whether the device is a USB-488 interface or not. This firmware does not provide a USB-488 interface.

2.1 Firmware Overview

The firmware will enumerate as a TMC device to the host. This will allow LabView to access the device using SCPI commands sent over the TMC protocol.

The SCPI commands supported are "`*IDN?`" for IDeNtify, "`*MEAS:hrm?`" to return heart rate MEASurement, "`*MEAS:for?`" for force sensor and "`*MEAS:tmp?`" for temperature sensor.

This does not cover the minimum set of SCPI commands that must be supported. However, the LabView application does not require additional commands and a trade-off between spaces available in the firmware and demonstrating multiple sensors has been made.

By default the FT51A Hub function is enabled in this firmware.

2.1.1 FT51A Libraries

The TMC firmware uses the FT51A USB library, DFU library, SPI Master library, TMC library, general config library and the IOMUX library. The IOMUX library is not used in the example code but is included to allow further functionality to be added. A code module for the ADT7310 temperature sensor (via the SPI Master library) is included.

The firmware is designed for the FT51A EVM module and may be extended to use the LCD for displaying some information. If so, then it will need the I2C Master library added.

DFU functionality is implemented as described in AN_344 DFU Sample.

2.2 Labview Application Overview

A standalone LabView application and LabView source code is included with the firmware.

The user must also install the following software available from National Instruments website, which provides a good search facility:

- LabVIEW Run-Time Engine (2013 version or compatible).
  Note that different OS versions are available.
  Running the TMC application when not installed gives the following error. Click yes to be directed to the website for executable file download:

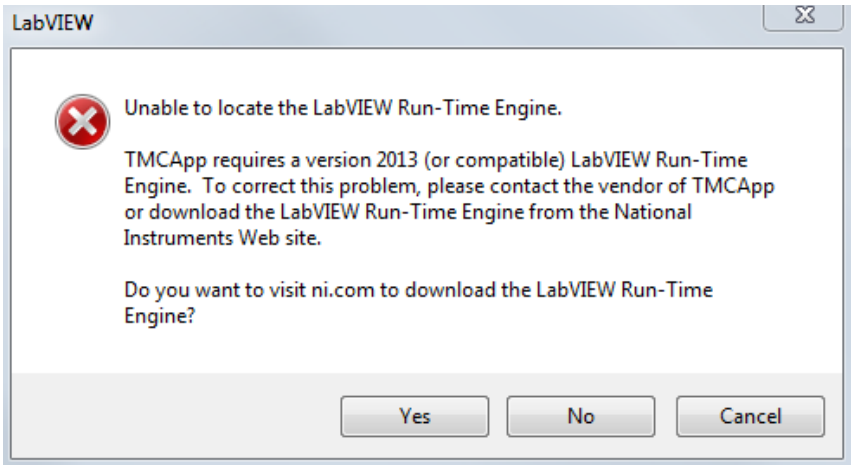

Figure 2.2-1 LabVIEW Run-Time Engine Alert Dialog Box

- NI Device drivers (2014.08 Part 1 and Part 2 at time of writing)

The application will repeatedly read the sensors from up to four connected FT51A EVM boards and display this on a chart. The application uses the LabView VISA drivers.

3 Test and Measurement Firmware

The firmware included in the example code demonstrates a test and measurement device.

The firmware is designed for the FT51A EVM module. It will use the force sensor, temperature sensor and heart rate sensor. The force and heart rate sensors use an analogue voltage input which is converted using the ADC features. The temperature sensor uses an ADT7310 which is connected to the FT51A *via* an SPI bus.

The DFU functionality is optional. It is normally enabled but can be disabled by setting the macro `DFU` to zero in the header file `tmc_app_usb.h` and removing the DFU library. The LCD output can also be enabled here by setting the `LCD` macro non-zero and adding both the LCD and I2C Master libraries to the project.

Three additional macros in the same file enable or disable the code to read the sensors. These can be changed individually to leave out the force, heart rate and temperature sensor reading code; the macros are `MEAS_FORCE`, `MEAS_HR` and `MEAS_TMP` respectively.

However as program space is limited not all functions can be enabled at the same time.

Removing the DFU code will reduce the size of the compiled code by approximately 1600 bytes. The LCD library code (I2C Master and LCD) total about 650 bytes.

In the project settings a symbol `FT51A_INTERRUPTS` is defined to remove support for unused interrupts in the FT51A_interrupts.c file. Removing unused interrupt handers will reduce the size of the compiled code.

When compiling, the SDCC command line will have the following macro added.

```
-D"FT51A_INTERRUPTS=(FT51A_INT_TIMER0+FT51A_INT_ADC+FT51A_INT_I2CM+FT51A_INT_PERIPHERALS)"
```

In Eclipse the symbol is defined in the Project Properties in the "Paths and Symbols" area of the "C/C++ General" section. Compiler symbols are defined in the "Symbols" Tab when "SDCC" language is selected.

The `FT51A_INT_I2CM` macro is for the I2C Master and is included in case the LCD function is added. `FT51A_INT_PERIPHERALS` includes the USB function.

In the USB library, there is a macro in the file `FT51A_usb_internal.h` called `USB_MAX_ENDPOINT_COUNT` which can enable or disable code to support multiple endpoints on a device. The default value is up-to 2 endpoints. This project has 2 non-control endpoints so the default value is left. If another endpoint is added then the macro can be added to the "Paths and Symbols".

3.1 USB Descriptors

The TMC firmware stores two sets of device descriptors and configuration descriptors. It stores a single table of string descriptors as the strings for run time and DFU modes can be selected by the descriptors as needed from the same table.

The control endpoint max packet size is defined as 16 bytes. The bulk IN and OUT endpoints for sending the TMC messages are set to a maximum of 64 bytes. Internal to the firmware the size of the buffer used for TMC messages is 64 bytes.

```
// USB Endpoint Zero packet size (both must match)
#define USB_CONTROL_EP_MAX_PACKET_SIZE 16
#define USB_CONTROL_EP_SIZE USB_EP_SIZE_16
// USB Bulk Endpoint packet size (both must match)
#define USB_BULK_EP_MAX_PACKET_SIZE 64
#define USB_BULK_EP_SIZE USB_EP_SIZE_64
// FTDI predefined TMC Demo Product ID.
#define USB_PID_DEMO 0x0feb
```

The Product IDs (PIDs) for run time (0x0FEB) and DFU mode (0x0FEE – the same as the AN_344 DFU Sample interface) are also defined in the source code.

These are example PID values and **must not** be used in a final product. VID and PID combinations must be unique to an application.

The USB class, subclass and protocols along with other general USB definitions are found in the file `FT51A_usb.h` library include file.

3.1.1 TMC Descriptors

The first set of descriptors is the run time set for the TMC function. The device descriptor contains the VID and PID for the TMC function.

```
__code USB_device_descriptor device_descriptor_demo =
{
    .bLength = 0x12,
    .bDescriptorType = USB_DESCRIPTOR_TYPE_DEVICE,
    .bcdUSB = USB_BCD_VERSION_2_0,          // V2.0
    .bDeviceClass = USB_CLASS_DEVICE,       // Defined in interface
    .bDeviceSubClass = USB_SUBCLASS_DEVICE, // Defined in interface
    .bDeviceProtocol = USB_PROTOCOL_DEVICE, // Defined in interface
    .bMaxPacketSize0 = USB_CONTROL_EP_MAX_PACKET_SIZE,
    .idVendor = USB_VID_FTDI,   // idVendor: 0x0403 (FTDI)
    .idProduct = USB_PID_DEMO, // idProduct: 0x0feb
    .bcdDevice = 0x0101,        // 1.1
    .iManufacturer = 0x01,      // Manufacturer
    .iProduct = 0x02,           // Product
    .iSerialNumber = 0x03,      // Serial Number
    .bNumConfigurations = 0x01
};
```

The configuration descriptor contains an interface descriptor, and 2 endpoint descriptors for the run time function.

If the `DFU` macro is non-zero, it also has an interface descriptor for a DFU interface. This includes a DFU functional descriptor. It must load the libusb-win32 driver for the DFU function to work.

```
// Structure containing layout of configuration descriptor
__code struct config_descriptor_demo
{
    USB_configuration_descriptor configuration;
    USB_interface_descriptor interface;
    USB_endpoint_descriptor endpoint_in;
    USB_endpoint_descriptor endpoint_out;
#if DFU
    USB_interface_descriptor dfu_interface;
    USB_dfu_functional_descriptor dfu_functional;
#endif // DFU
};
struct config_descriptor_demo config_descriptor_demo =
{
    .configuration.bLength = 0x09,
    .configuration.bDescriptorType = USB_DESCRIPTOR_TYPE_CONFIGURATION,
    .configuration.wTotalLength = sizeof(struct config_descriptor_demo),
#if DFU
    .configuration.bNumInterfaces = 0x02,
#else // DFU
    .configuration.bNumInterfaces = 0x01,
#endif // DFU
    .configuration.bConfigurationValue = 0x01,
    .configuration.iConfiguration = 0x00,
    .configuration.bmAttributes = USB_CONFIG_BMATTRIBUTES_VALUE,
    .configuration.bMaxPower = 0xFA,          // 500mA
```

```
    // ---- INTERFACE DESCRIPTOR for Demo ----
    .interface.bLength = 0x09,
    .interface.bDescriptorType = USB_DESCRIPTOR_TYPE_INTERFACE,
    .interface.bInterfaceNumber = 0,
    .interface.bAlternateSetting = 0x00,
    .interface.bNumEndpoints = 0x02,
    .interface.bInterfaceClass = USB_CLASS_APPLICATION, // Application class
    .interface.bInterfaceSubClass = 0x03,   // Test and Measurement
    .interface.bInterfaceProtocol = 0x00,   // 0x00 = USBTMC
                                            // 0x01 = USBTMC USB488 interface.
    .interface.iInterface = 0x06,           // "FT51A Demo"
    // ---- ENDPOINT DESCRIPTOR for Demo ----
    .endpoint_in.bLength = 0x07,
    .endpoint_in.bDescriptorType = USB_DESCRIPTOR_TYPE_ENDPOINT,
    .endpoint_in.bEndpointAddress = 0x01,
    .endpoint_in.bmAttributes = USB_ENDPOINT_DESCRIPTOR_ATTR_BULK,
    .endpoint_in.wMaxPacketSize = USB_BULK_EP_MAX_PACKET_SIZE,
    .endpoint_in.bInterval = 0x0,
    // ---- ENDPOINT DESCRIPTOR for Demo ----
    .endpoint_out.bLength = 0x07,
    .endpoint_out.bDescriptorType = USB_DESCRIPTOR_TYPE_ENDPOINT,
    .endpoint_out.bEndpointAddress = 0x81,
    .endpoint_out.bmAttributes = USB_ENDPOINT_DESCRIPTOR_ATTR_BULK,
    .endpoint_out.wMaxPacketSize = USB_BULK_EP_MAX_PACKET_SIZE,
    .endpoint_out.bInterval = 0x0,

#if DFU
    // ---- INTERFACE DESCRIPTOR for DFU Interface ----
    .dfu_interface.bLength = 0x09,
    .dfu_interface.bDescriptorType = USB_DESCRIPTOR_TYPE_INTERFACE,
    .dfu_interface.bInterfaceNumber = DFU_USB_INTERFACE_RUNTIME,
    .dfu_interface.bAlternateSetting = 0x00,
    .dfu_interface.bNumEndpoints = 0x00,
    .dfu_interface.bInterfaceClass = USB_CLASS_APPLICATION, // Application Specific Class
    .dfu_interface.bInterfaceSubClass = USB_SUBCLASS_DFU, // Device Firmware Update
    .dfu_interface.bInterfaceProtocol = USB_PROTOCOL_DFU_RUNTIME, // Runtime Protocol
    .dfu_interface.iInterface = 0x05,       // "DFU Interface"
    // ---- FUNCTIONAL DESCRIPTOR for DFU Interface ----
    .dfu_functional.bLength = 0x09,
    .dfu_functional.bDescriptorType = USB_DESCRIPTOR_TYPE_DFU_FUNCTIONAL,
    .dfu_functional.bmAttributes = 0x01,    // bitCanDnload
    .dfu_functional.wDetatchTimeOut = DFU_TIMEOUT, // suggest 8192ms
    .dfu_functional.wTransferSize = DFU_BLOCK_SIZE, // make it 16 bytes
    .dfu_functional.bcdDfuVersion = USB_BCD_VERSION_DFU_1_1, // DFU Version 1.1
#endif // DFU
};
```

3.1.2 DFU Descriptors

If enabled with the `DFU` macro set to non-zero, the firmware has a device descriptor for the DFU function. It contains the VID and PID for the DFU function. This may or may not be the same as the run time VID and PID. In line with the run time configuration this must load the libusb-win32 driver for DFU interface.

```
USB_device_descriptor device_descriptor_dfumode =
{
        .bLength = 0x12,
        .bDescriptorType = USB_DESCRIPTOR_TYPE_DEVICE,
        .bcdUSB = USB_BCD_VERSION_2_0,
        .bDeviceClass = USB_CLASS_DEVICE,
        .bDeviceSubClass = USB_SUBCLASS_DEVICE,
```

```
        .bDeviceProtocol = USB_PROTOCOL_DEVICE,
        .bMaxPacketSize0 = USB_CONTROL_EP_MAX_PACKET_SIZE,
        .idVendor = USB_VID_FTDI,   // 0x0403 (FTDI)
        .idProduct = DFU_USB_PID_DFUMODE, // 0x0fee
        .bcdDevice = 0x0101,
        .iManufacturer = 0x01,
        .iProduct = 0x04,
        .iSerialNumber = 0x03,
        .bNumConfigurations = 0x01
};
```

The configuration descriptor for DFU will contain only an interface descriptor and a functional descriptor for the DFU interface.

The USB class, subclass and protocol indicate that this device is now in DFU mode.

```
// Structure containing layout of configuration descriptor
struct config_descriptor_dfumode
{
    USB_configuration_descriptor configuration;
    USB_interface_descriptor interface;
    USB_dfu_functional_descriptor functional;
};

struct config_descriptor_dfumode config_descriptor_dfumode =
{
        .configuration.bLength = 0x09,
        .configuration.bDescriptorType = USB_DESCRIPTOR_TYPE_CONFIGURATION,
        .configuration.wTotalLength = sizeof(struct config_descriptor_dfumode),
        .configuration.bNumInterfaces = 0x01,
        .configuration.bConfigurationValue = 0x01,
        .configuration.iConfiguration = 0x00,
        .configuration.bmAttributes = USB_CONFIG_BMATTRIBUTES_VALUE,
        .configuration.bMaxPower = 0xFA, // 500 mA

              // ---- INTERFACE DESCRIPTOR for DFU Interface ----
        .interface.bLength = 0x09,
        .interface.bDescriptorType = USB_DESCRIPTOR_TYPE_INTERFACE,
        .interface.bInterfaceNumber = DFU_USB_INTERFACE_DFUMODE,
        .interface.bAlternateSetting = 0x00,
        .interface.bNumEndpoints = 0x00,
        .interface.bInterfaceClass = USB_CLASS_APPLICATION, // Application Specific Class
        .interface.bInterfaceSubClass = USB_SUBCLASS_DFU,   // Device Firmware Update
        .interface.bInterfaceProtocol = USB_PROTOCOL_DFU_DFUMODE, // Runtime Protocol
        .interface.iInterface = 0x05, // String 5

        // ---- FUNCTIONAL DESCRIPTOR for DFU Interface ----
        .functional.bLength = 0x09,
        .functional.bDescriptorType = USB_DESCRIPTOR_TYPE_DFU_FUNCTIONAL,
        .functional.bmAttributes = 0x01, // bitCanDnload
        .functional.wDetatchTimeOut = DFU_TIMEOUT, // suggest 8192ms
        .functional.wTransferSize = DFU_BLOCK_SIZE,// typically 64 bytes
        .functional.bcdDfuVersion = USB_BCD_VERSION_DFU_1_1,
};
```

The same `bmAttributes` mask must appear for the DFU functional descriptor in both run time and DFU modes.

3.1.3 Descriptor Selection

The standard request handler for GET_DESCRIPTOR requests needs to select the run time or DFU mode descriptors for device and configuration descriptors. Other descriptors, including the report descriptors and string descriptors, are not affected.

Determining if the firmware is in run time or DFU mode is achieved by calling the `dfu_is_runtime()` function from the DFU library.

A non-zero response will select the run time mode descriptors and a zero response, the DFU mode descriptors.

```
FT51A_STATUS standard_req_get_descriptor(USB_device_request *req)
{
    uint8_t   *src = NULL;
    uint16_t   length = req->wLength;
    uint8_t    hValue = req->wValue >> 8;
    uint8_t    lValue = req->wValue & 0x00ff;
    uint8_t    i, slen;

    switch (hValue)
    {
    case USB_DESCRIPTOR_TYPE_DEVICE:

#ifd DFU
        if (dfu_is_runtime())
#endif // DFU
        {
            src = (char *) &device_descriptor_demo;
        }
#if DFU
        else
        {
            src = (char *) &device_descriptor_dfumode;
        }
#endif // DFU
        if (length > sizeof(USB_device_descriptor)) // too many bytes requested
            length = sizeof(USB_device_descriptor); // Entire structure.
        break;

    case USB_DESCRIPTOR_TYPE_CONFIGURATION:

#if DFU
        if (dfu_is_runtime())
#endif // DFU
        {
            src = (char *) &config_descriptor_demo;
            if (length > sizeof(config_descriptor_demo)) // too many bytes requested
                length = sizeof(config_descriptor_demo); // Entire structure.
        }
#if DFU
        else
        {
            src = (char *) &config_descriptor_dfumode;
            if (length > sizeof(config_descriptor_dfumode)) // too many bytes requested
                length = sizeof(config_descriptor_dfumode); // Entire structure.
        }
#endif // DFU
        break;
```

The FT51A USB library will return the structure pointed to by the `standard_req_get_descriptor()` function.

Note that string descriptor selection is not shown in this code sample.

3.2 USB Class Requests

The firmware is responsible for handling USB class requests. It must determine if the firmware is in run time or DFU mode and whether a request has been directed to the DFU interface. This must not interfere with other class requests that may be decoded in the firmware.

The first check is that the class request is aimed at an interface:

```
FT51A_STATUS class_req_cb(USB_device_request *req)
{
    FT51A_STATUS         status = FT51A_FAILED;
    uint8_t             interface = LSB(req->wIndex) & 0x0F;

    // For DFU requests ensure the recipient is an interface...
    if ((req->bmRequestType & USB_BMREQUESTTYPE_RECIPIENT_MASK) ==
            USB_BMREQUESTTYPE_RECIPIENT_INTERFACE)
    {
```

If this is correct then the firmware must check if it is in run time or DFU mode before checking the interface number. The interface number for the DFU mode may differ from that of the run time mode.

Requests to the DFU interface are passed to the DFU library but others are handled as HID requests.

```
#if DFU
        // ...and that the interface is the correct Runtime interface
        if (dfu_is_runtime())
#endif // DFU
        {

#if DFU
            if ((interface == DFU_USB_INTERFACE_RUNTIME))
            {
                // Handle DFU requests DFU_DETATCH, DFU_GETSTATE and DFU_GETSTATUS
                // when in run time mode.
                switch (req->bRequest)
                {
                case USB_CLASS_REQUEST_DETACH:
                    dfu_class_req_detach(req->wValue);
                    status = FT51A_OK;
                    break;
                case USB_CLASS_REQUEST_GETSTATUS:
                    dfu_class_req_getstatus();
                    status = FT51A_OK;
                    break;
                case USB_CLASS_REQUEST_GETSTATE:
                    dfu_class_req_getstate();
                    status = FT51A_OK;
                    break;
                }
            }
            else
#endif // DFU
            {
                status = FT51A_OK;
                // Handle only TMC Class interface requests when not
                // in DFU mode.
                switch ( req->bRequest)
```

```
                {
                default:
                    status = FT51A_FAILED;
                    break;
                case USBTMC_GET_CAPABILITIES:
                    tmc_class_req_capabilities(0, 0);
                    break;
                case USBTMC_INITIATE_CLEAR:
                    tmc_class_init_clear();
                    break;
                case USBTMC_CHECK_CLEAR_STATUS:
                  tmc_class_check_clear();
                  break;
                }
            }
        }
    }
```

When the device is in DFU mode the DFU request handlers are called as described in the AN 344 DFU Sample.

The TMC class must handle GET_CAPABILITIES, INITIATE_CLEAR and CHECK_CLEAR_STATUS requests sent to the interface.

Requests sent to a TMC endpoint must also be handled.

```
    // Check for Endpoint recipient
    else if((req->bmRequestType & USB_BMREQUESTTYPE_RECIPIENT_MASK) ==
    USB_BMREQUESTTYPE_RECIPIENT_ENDPOINT)
    {
        // TMC class endpoint requests
        status = FT51A_OK;
        switch ( req->bRequest)
        {
        default:
            status = FT51A_FAILED;
            break;
        case USBTMC_INITIATE_ABORT_BULK_OUT:
            tmc_class_init_abort_bulk_out((req->wValue) & 0xFF, USB_EP_1, USB_DIR_OUT);
            break;
        case USBTMC_CHECK_ABORT_BULK_OUT_STATUS:
            tmc_class_check_abort_bulk_out();
            break;
        case USBTMC_INITIATE_ABORT_BULK_IN:
            tmc_class_init_abort_bulk_in((req->wValue) & 0xFF, USB_EP_1, USB_DIR_IN);
            break;
        case USBTMC_CHECK_ABORT_BULK_IN_STATUS:
            tmc_class_check_abort_bulk_in();
            break;
        }
    }
```

Each endpoint can receive INITIATE_ABORT_BULK_OUT/IN and CHECK_ABORT_BULK_OUT/IN requests. These are handled by the TMC library.

3.3 USB Reset Handler

The reset function handler is used to make the transition from run time mode to DFU mode.

3.4 Timer

A timer is used to provide delays and time measurements for implementing polling intervals.

The first timer `ms_timer` is used to create general purpose delays, for instance when resetting the temperature sensor.

Each of the three sensors has a timer `forceTimer`, `tempTimer` and `pulseTimer`. These timers operate independently.

The DFU also needs a millisecond timer to accurately return to the appIDLE state from the appDETACH state. The `dfu_timer()` function in the DFU library, if enabled, should be called every millisecond to enable this.

```
void ms_timer_interrupt(const uint8_t flags)
{
    (void) flags; // Flags not currently used
    if (ms_timer)
    {
        ms_timer--;
    }
#if MEAS_HR
    if (pulseTimer)
        pulseTimer--;
#endif // MEAS_HR
#if MEAS_TEMP
    if (tempTimer)
        tempTimer--;
#endif // MEAS_TEMP
#if MEAS_FORCE
    if (forceTimer)
        forceTimer--;
#endif // MEAS_FORCE
#if DFU
    // The DFU detach timer must be called once per millisecond
    dfu_timer();
#endif
    // Reload the timer
    TH0 = MSB(MICROSECONDS_TO_TIMER_TICKS(1000));
    TL0 = LSB(MICROSECONDS_TO_TIMER_TICKS(1000));
}

void ms_timer_initialise(void)
{
    // Register our own handler for interrupts from Timer 0
    interrupts_register(ms_timer_interrupt, interrupts_timer0);

    // Timer0 is controlled by TMOD bits 0 to 3, and TCON bits 4 to 5.
    TMOD &= 0xF0; // Clear Timer0 bits
    TMOD |= 0x01; // Put Timer0 in mode 1 (16 bit)
    // Set the count-up value so that it rolls over to 0 after 1 millisecond.
    TH0 = MSB(MICROSECONDS_TO_TIMER_TICKS(1000));
    TL0 = LSB(MICROSECONDS_TO_TIMER_TICKS(1000));
    TCON &= 0xCF; // Clear Timer0's Overflow and Run flags
    TCON |= 0x10; // Start Timer0 (set its Run flag)
}
```

3.5 TMC Messages

The TMC application will receive TMC messages from the TMC library.

In the `main` loop of the application there is a call to the `TMC_process()` function. This will check for messages received from the LabView application (TMCApp.exe) running on the host PC and process them. Messages are decoded by a callback into the application from the library. The callbacks are setup when initialising the TMC library:

```
    tmc_ctx.message_cmd_cb = tmc_command_cb;
    tmc_ctx.message_rsp_cb = tmc_response_cb;
    // Initialise the TMC interface
    tmc_initialise(&tmc_ctx);
```

The callback function `tmc_command_cb()` will decode the incoming command from a TMC Device Dependent Message Out. In this case the message is an SCPI command from the `SCPI_cmds` table. There is a matching table called `scpi_cmd_list` which contains pointers to handler functions to perform the required operation.

The `SCPI_cmds` and `scpi_cmd_list` tables are shown below. Additional commands and handlers can be added to support more SCPI commands.

```
__code char *SCPI_cmds[] = {
    "*IDN?\n",
#if MEAS_HR
    "*MEAS:hrm?\n",
#endif
#if MEAS_FORCE
    "*MEAS:for?\n",
#endif
#if MEAS_TEMP
    "*MEAS:tmp?\n",
#endif
    NULL,
};

__code SCPI_CMD_t scpi_cmd_list[] =
{
    scpi_cmd_identify_query, // *IDN?
#if MEAS_HR
    scpi_cmd_meas_hr, // *MEAS:hrm?
#endif
#if MEAS_FORCE
    scpi_cmd_meas_force, // *MEAS:for?
#endif
#if MEAS_TEMP
    scpi_cmd_meas_temp, // *MEAS:tmp?
#endif
    NULL,
};
```

In our implementation the SCPI command handler functions generate a response string from the latest measurements from the sensors. This is kept until the LabView application (TMCApp.exe) running on the host PC reads the response back with a TMC Device Dependent Message In.

The `tmc_response_cb()` callback function will copy the response string into the message sent back to the TMC application on the host.

This simple method of parsing TMC messages provides adequate coverage to allow a LabView application to use the VISA interface to access TMC devices, without adding too much code to the TMC firmware.

Note that there is neither, verification of bTag values in the TMC messages nor any method of satisfying multiple concurrent SCPI commands. A TMC application or LabView application can be written to avoid such overlapping messages.

3.6 Sensor Reading Acquisition

Each sensor is read from the firmware's `main` loop.

3.6.1 Force Sensor

When the `forceTimer` reaches zero then a new ADC is triggered and the timer restarted. The ADC conversion will take place in the background and an interrupt generated when it is complete.

```
        if (forceTimer == 0)
        {
            IO_REG_INTERRUPTS_WRITE(IO_CELL_SAMPLE_0_7_1, MASK_IO_CELL_0_SAMPLE);

            forceTimer = FORCE_TIMER;
        }
```

The ADC interrupt handler (section below) will update the `forceSample` and `forceSampleReady` variables when the reading is complete.

```
    IO_REG_INTERRUPTS_READ(IO_CELL_INT_0_1, interrupt);
    // FSR Output
    if (interrupt & MASK_SD_CELL_0_INT)
    {
        IO_REG_INTERRUPTS_READ(IO_CELL_0_ADC_DATA_L_1, sample_l);
        IO_REG_INTERRUPTS_READ(IO_CELL_0_ADC_DATA_U_1, sample_h);

        forceSample = ((sample_h << 8) | sample_l);
        forceSampleReady = TRUE;

        // Clear ADC interrupt register bit
        IO_REG_INTERRUPTS_WRITE(IO_CELL_INT_0_1, MASK_SD_CELL_0_INT);
    }
```

The main loop will arrive back at the force sensor code and update the force reading when the `forceSampleReady` flag is set by the interrupt handler.

```
        if (forceSampleReady)
        {
            forceSampleReady = FALSE;
            if (forceSample > 900)
            {
                forceSample = 900;
            }
            force = forceSample;
        }
```

The `forceTimer` will continue to run until the next sample is due.

3.6.1 Heart Rate Sensor

When the `pulseTimer` reaches zero then a new ADC is triggered and the timer restarted. The ADC conversion will take place in the background and an interrupt generated when it is complete.

```
        if (pulseTimer == 0)
        {
            IO_REG_INTERRUPTS_WRITE(IO_CELL_SAMPLE_8_15_1, MASK_IO_CELL_10_SAMPLE);
            pulseTimer = PULSE_TIMER;
        }
```

The ADC interrupt handler will update the `pulseSample` and `pulseSampleReady` variables when the reading is complete.

```
    IO_REG_INTERRUPTS_READ(IO_CELL_INT_1_1, interrupt);
    // Pulse Rate
    if (interrupt & MASK_SD_CELL_10_INT)
    {
        IO_REG_INTERRUPTS_READ(IO_CELL_10_ADC_DATA_L_1, sample_l);
        IO_REG_INTERRUPTS_READ(IO_CELL_10_ADC_DATA_U_1, sample_h);

        pulseSample = ((sample_h << 8) | sample_l);
        pulseSampleReady = TRUE;

        // Clear ADC interrupt register bit
        IO_REG_INTERRUPTS_WRITE(IO_CELL_INT_1_1, MASK_SD_CELL_10_INT);
    }
```

The main loop will arrive back at the heart rate sensor code and update the heart beat reading when the `pulseSampleReady` flag is set by the interrupt handler.

```
        if (pulseTimer == 0)
        {
            IO_REG_INTERRUPTS_WRITE(IO_CELL_SAMPLE_8_15_1, MASK_IO_CELL_10_SAMPLE);
            pulseTimer = PULSE_TIMER;
        }
        if (pulseSampleReady)
        {
            pulseSampleReady = FALSE;
            if (pulseSample > 0xff)
                pulseSample = 0xff;
            pulseSamples[pulseIndexer] = pulseSample;

            // Count the total number of beats (for the entire sample period) ...
            heartRate = 0;
            pulseCounterHystRising = pulseCounterHystFalling = 0;
            for (j = 0; j < PULSE_SAMPLES; j++)
            {
                if (pulseSamples[j] > PULSE_THRESHOLD)
                {
                    // Look for PULSE_IS_BEAT samples in a row to be above the
                    // threshold.
                    pulseCounterHystRising++;
                    if (pulseCounterHystRising == PULSE_IS_BEAT)
                    {
                        pulseCounterHystFalling = PULSE_IS_BEAT;
                        heartRate++;
                    }
                }
                else
```

```
                {
                    if (pulseCounterHystFalling)
                    {
                        pulseCounterHystFalling--;
                        pulseCounterHystRising = 0;
                    }
                }
            }
            // Correct for sample array size. i.e. 600 samples at 25ms intervals
            // will be 15 seconds of data. So multiply by 60/15 = 4 times to get
            // pulse per minute.
            heartRate *= (60000 / (PULSE_SAMPLES * PULSE_TIMER));

            pulseIndexer = (++pulseIndexer) % PULSE_SAMPLES;
        }
```

The loop fills a buffer with PULSE_SAMPLES number of samples of the converted analogue voltage from the heart rate circuit. It then parses the buffer to identify a certain number of consecutive samples above or below a threshold value of PULSE_THRESHOLD. Each of these transitions is regarded as a detected heartbeat. The final operation is to count these heartbeats and convert the result into beats per minute.

3.6.2 Temperature Sensor

An SPI bus is used to measure the temperature from an ADT7310 sensor connected to the SPI Master interface. When the tempTimer reaches zero the timer is reset and an SPI Master read is performed to the ADT3710.

```
        if (tempTimer == 0)
        {
            // Read the temperature from the SPI Master
            temperature = temperature_read();
            tempTimer = TEMP_TIMER;
        }
```

The temperature is returned in units of 0.01 ˚C.

4 LabView Application

The LabView application uses the National Instruments VISA driver to communicate with the TMC Firmware.

Provided is a pre-compiled application which will display the readings obtained from the FT51A EVM modules in a chart and as dials. The source code form is provided as well.

4.1 Requirements

Serial numbers for each FT51A EVM module running the firmware must be unique. The FT51 Software Development Kit has a command line utility called `FT51Astr.exe` that can alter string descriptors. String number 3 in the string descriptor table in the Test and Measurement firmware contains the serial number.

For each FT51A EVM module, modify a copy of the original firmware IHX file with the `FT51Astr.exe` utility to change string 3 to a unique value and program that onto the module.

```
C:\workspace\tmc_app\Debug>ft51str -v -3 "NI-VISA-42422-b" tmc_app.ihx
Reading input file tmc_app.ihx ... done
Finding strings ....... done
String table size 0x9E bytes with 7 entries
Modifying string 3 to "NI-VISA-42422-b"
Finding string 3 ... done
Writing replacement string ............... done
Writing tmc_app.ihx.mod ... done
```

5 Possible Improvements

It would be possible to implement a different method of handling the messages, potentially starting to acquire samples in a handler function when called through `tmc_command_cb()` then formatting the result in a new handler called from `tmc_response_cb()`. This would be useful in situations where a result cannot be continually updated or the time taken to formulate a response is relatively long, either due to sampling periods or the length of time that some hardware takes to respond.

6 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales) sales1@ftdichip.com
E-mail (Support) support1@ftdichip.com
E-mail (General Enquiries) admin1@ftdichip.com

Branch Office – Tigard, Oregon, USA

Future Technology Devices International Limited
(USA)
7130 SW Fir Loop
Tigard, OR 97223-8160
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

E-Mail (Sales) us.sales@ftdichip.com
E-Mail (Support) us.support@ftdichip.com
E-Mail (General Enquiries) us.admin@ftdichip.com

Branch Office – Taipei, Taiwan

Future Technology Devices International Limited
(Taiwan)
2F, No. 516, Sec. 1, NeiHu Road
Taipei 114
Taiwan , R.O.C.
Tel: +886 (0) 2 8791 3570
Fax: +886 (0) 2 8791 3576

E-mail (Sales) tw.sales1@ftdichip.com
E-mail (Support) tw.support1@ftdichip.com
E-mail (General Enquiries) tw.admin1@ftdichip.com

Branch Office – Shanghai, China

Future Technology Devices International Limited
(China)
Room 1103, No. 666 West Huaihai Road,
Shanghai, 200052
China
Tel: +86 21 62351596
Fax: +86 21 62351595

E-mail (Sales) cn.sales@ftdichip.com
E-mail (Support) cn.support@ftdichip.com
E-mail (General Enquiries) cn.admin@ftdichip.com

Web Site

http://ftdichip.com

Distributor and Sales Representatives

Please visit the Sales Network page of the FTDI Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A – References

Document References

FTDI MCU web page: http://www.ftdichip.com/MCU.html

USB Test and Measurement Class specification:
http://www.usb.org/developers/docs/devclass_docs/USBTMC_1_006a.zip

IVI Foundation: http://www.ivifoundation.org/

SCPI specification: http://www.ivifoundation.org/docs/scpi-99.pdf

USB Device Firmware Update Class specification:
http://www.usb.org/developers/docs/devclass_docs/DFU_1.1.pdf

Acronyms and Abbreviations

Terms	Description
HID	Human Interface Device
MTP	Multiple Time Program – non-volatile memory used to store program code on the FT51A.
SCPI	Standard Commands for Programmable Instruments
TMC	Test and Measurement Class
USB	Universal Serial Bus
USB-IF	USB Implementers Forum
VISA	Virtual Instruments Software Architecture

Appendix B – List of Tables & Figures

List of Figures

Appendix C – Revision History

Document Title: AN_347 FT51A Test and Measurement Sample
Document Reference No.: FT_001124
Clearance No.: FTDI# 432
Product Page: http://www.ftdichip.com/FTProducts.htm
Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial Release	2014-12-12
1.1	Update FT51 references to FT51A	2015-11-26