# Firmware Design Document

Module: Wireless bluetooth EKG glove monitor

Author:Matt Buonadonna

# Revision History

| Date | Author | Issue | Comment |
|---|---|---|---|
| 8/11/2014 | Matt Buonadonna | 1.0 | First version |
| 9/15/2014 | Matt Buonadonna | 2.0 | New module diagram, new state diagram with new features |
| 9/26/2014 | Matt Buonadonna | 2.1 | Drivers, Application and 3rd party added |
| 10/8/2014 | Matt Buonadonna | 2.2 | UI and Radio module description added |
| 10/10/2014 | Matt Buonadonna | 2.3 | Protocol analyzer module description added |
| 10/13/2014 | Ben Schanz | 2.4 | UI module description updated |
| 10/20/2014 | Matt Buonadonna | 2.5 | USB interface & mass storage |
| | | | |
| | | | |

# Reference


| | |
|---|---|
| 1 | TivaWare-ROM USER'S GUIDE, document ID spmu367 |
| 2 | TivaWare™ Boot Loader, document ID spmu301 |
| 3 | Tiva tm4c1233h6pm Microcontroller, document ID SPMS351E |
| 4 | iWRAP6_bluegiga-User_Guide, version 2.4 |
| 5 | |
| 6 | |
| 7 | |

# 1. Introduction

This document describes the design of the IneedMD module device (herein referred to as the device) that will be attached to the EKG glove. The overall function of this device is to receive analog data from a medical EKG type device or leads. The device will then store the received data locally. The device will also transmit wirelessly the data to an associated monitoring device such as a remote computer.

The modules purpose is to replace the wired connection between a patient and a monitoring computer.

# 2. Description

## 2.1 System description

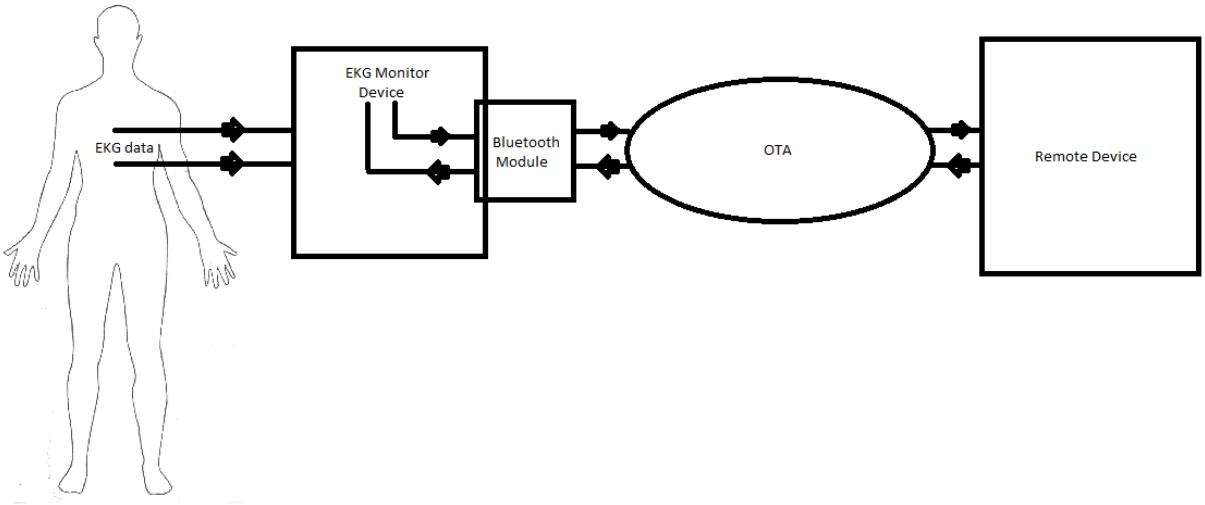



Image 2.1 - Overall system architecture

The overall system is to monitor a patient's EKG readings and display the output on a remote device. The remote device can be any device that can receive the wireless signal from the EKG monitoring Device. The remote device will display the EKG data as defined by the EKG standard. The OTA communications for this version of the product will be the bluetooth protocol utilizing a bluetooth module. The bluetooth module handles all the bluetooth protocol and handshaking process as defined by the EKG Monitoring Device's specifications. Patient data will "seamlessly" pass through the EKG module to the remote device for display and analysis.

## 2.2 Module description

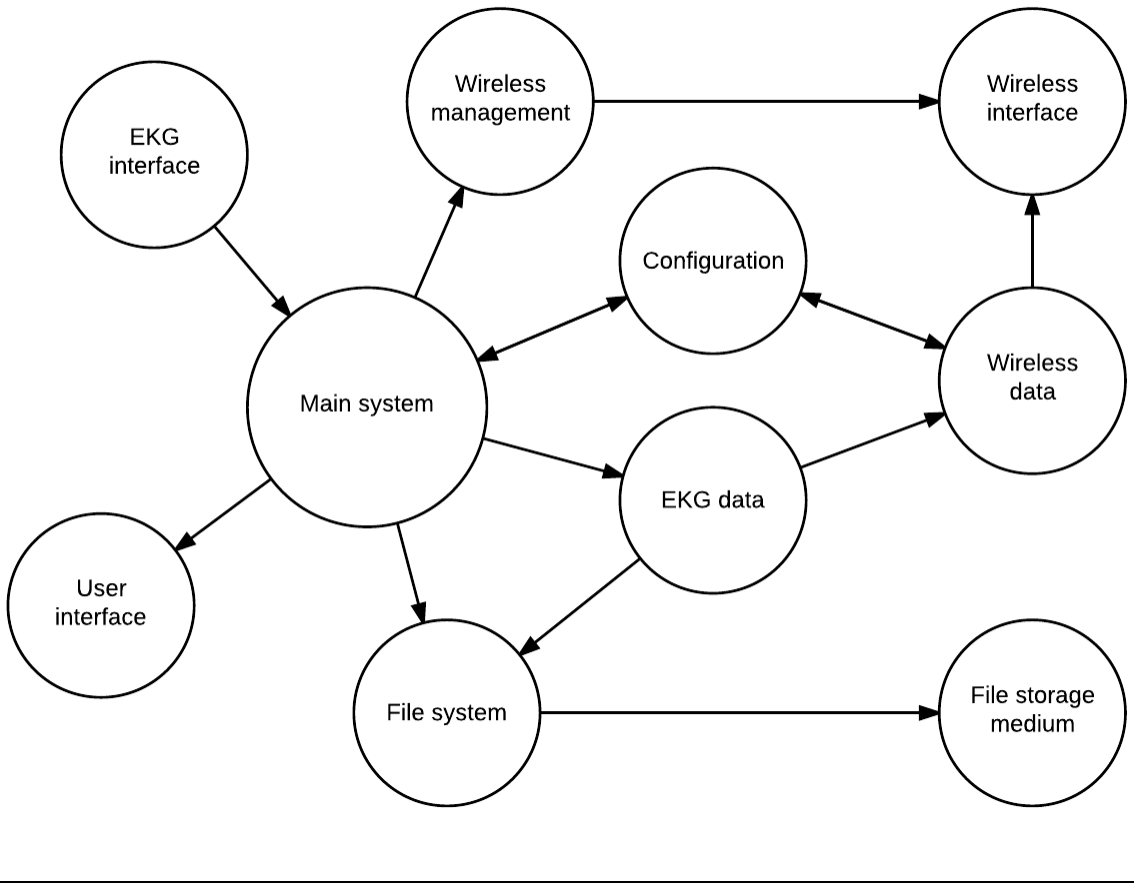



Image 2.2 - Module architecture

The EKG Monitoring device firmware overall architecture is laid out in Image 2.2. The module as a whole acts as an intermediary between the Patients EKG data and the outside world. The basic blocks of the module are as follows:

| | |
|---|---|
| Main system | This is the “glue” of the device that keeps all the other blocks in line and in sync. It is responsible for initializing the system and managing data flow to and from the communicating blocks. It will be in the form of a Real Time Operating System. |
| EKG interface | This block manages the data reception from the EKG device itself via driver interface. The primary responsibility of this block is to receive the raw data from the EKG device, |

| | |
|---|---|
| | convert it to a digital format and pass it on to the main system for further processing and or transmission. |
| User Interface | This module block controls the user interface of the device. The block receives the current status from the main system and displays or sounds the appropriate UI to the user. |
| File system | The file system is a software module that controls, inputs and outputs file data from the module management system to the data storage medium. It can receive data either from the |
| File storage medium | The driver interface to the storage medium that will hold EKG data |
| Wireless management | This module is responsible for setting up the wireless radio as required by the application. The application will specify how the wireless radio identifies itself, connection security, and connection type. |
| Wireless Interface | This is the driver module that takes I/O from the application and radio and sends it to its respective blocks. |
| Configuration | This is the data protocol block of the device. It takes data frames from the wireless module, identifies them as configuration data, and notifies the main system. These protocol frames configure and control the overall operation of the management system. These command frames are bi directional. |
| EKG data | This is the EKG handling block that sends data to the remote device. The primary responsibility of the EKG data block is to format the data,, and send the data frames to the radio or file system. |
| Wireless data | This module is responsible for receiving or sending data to the radio from either the protocol configuration data or EKG data modules. |

## 2.3 System States

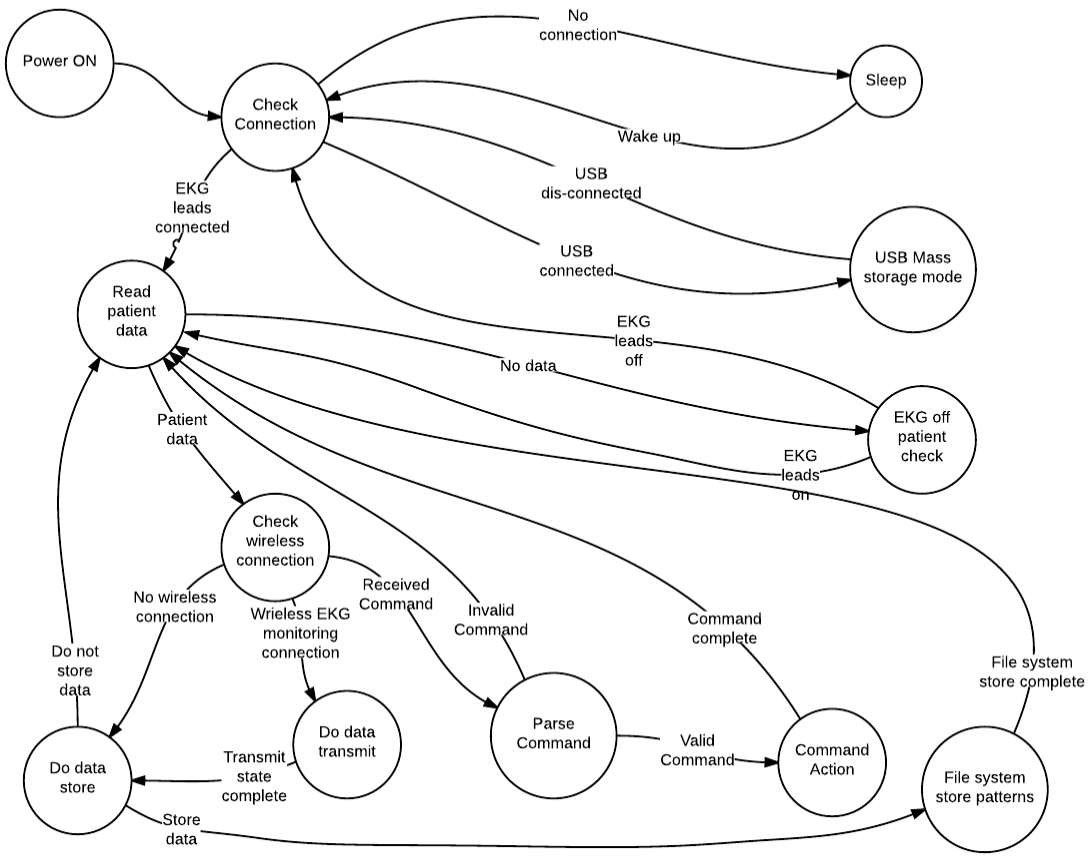



Image 2.3 - System State Diagram

The system state is mostly controlled by the current connection status with an external device. Since the system is strictly a communication interface between patient and monitor this is the primary means of state control. The other system controller is the current battery power.

### 2.3.1 Power ON

Device is powered on and goes through initialization.

### 2.3.2 Check Connection

System checks if a valid connection is made to get the system out of its sleep check state. If no connection is detected the system goes to a sleep state. There are two interfaces that are checked to get the system out of sleep state: EKG and USB. Once a valid connection is

detected the system will transition from an overall sleep hibernation state to an active state. Battery energy consumption will increase while in this state.

### 2.3.3 Sleep

The Sleep state puts the device in a low power mode. The system turns off peripherals and sets the sys clock to a low energy consumption rate. The Sleep mode will periodically transition to the Check Connection state to determine if the system needs to wake up and transition to the active state.

### 2.3.4 Read patient data

When the EKG leads are attached to the EKG port the system will detect the connection and transition to an active state. The data from the EKG port is read via A to D and verified. If there is no data to read or the data read is invalid the system will transition to an EKG lead check. If the data is valid the system will transition to the wireless connection state. Note that a shorting bar is used to keep the system in an active state. The data read from the A to D with the shorting bar is considered valid.

### 2.3.5 Check wireless connection

If patient data is received (including EKG shorting bar values) the system checks the wireless connection. During this check it is determined if any command frames were received and if an active EKG monitoring connection is made. If a command frame is received the system hands the command frame off to the protocol analyzer for processing. If an active EKG monitoring connection is present the system transmits the patient data to the remote monitor. If no active EKG monitoring connection is present the system will transition to a data store state.

### 2.3.6 Parse command

The parse command state takes the received command frame from the external device and checks it. If the command is valid the device transitions to the command action state. If the command is invalid the system transitions back to the patient data state.

### 2.3.7 Command Action

In the command action state the system takes the valid received command and performs the requested action. When the command action is completed the systems goes back to the patient data state.

### 2.3.8 Do data transmit

The data transmit state will either transmit the EKG data to the remote EKG monitor or not depending on the system settings. In either case of do or do not transmit data the system will transition out of the transmit data state.

### 2.3.9 Do data store

In the data store state the system settings for data storage will be checked. If the system is to store the data the system will transition to a file system storage state. If the data is not to be stored the system will transition to a patient data state.

#### 2.3.10 File system store patterns

The system store patterns state manages the file system for patient data storage. In this state data is stored according to the system settings (ie patient name, date, time, etc). Once data is stored the system transitions to a patient data state.

# 3. Drivers

The drivers are broken up into two major types: Low level and Device. Low level drivers are the basic drivers that almost every embedded microprocessor has. These types of drivers usually consist of GPIO, UART, SPI, etc, and are what talk to external device modules. Device drivers are the software that makes use of the level drivers to talk to external modules. The device drivers setup the external devices and get them ready for the application.

## 3.1 Low level drivers

Low level drivers are the basic drivers that almost every embedded microprocessor has. These types of drivers usually consist of GPIO, UART, SPI, etc, and are what talk to external device modules. They are well documented and usually easy to set up. Ti makes use of a simplistic build time HAL and built in ROM functions. The ROM functions initialize and manipulate the low level drivers. There is almost no need to call the low level driver registers directly since all the register modifications are handled by the ROM functions. The build time HAL uses project level defines to determine the hardware to link for. This HAL implementation routes the ROM functions to the proper addresses for the hardware being used. This provides another abstraction level away from direct register writes.

#### 3.1.1 GPIO

GPIO for the device is used to turn on and off peripheral devices and for debugging purposes.

1. **DEBUG_GPIO_PIN_1** -
    a. Pin type output
    b. This pin is part of the debugging hardware. Intended to be attached to whatever external device is to be used to monitor it (IE: Oscilloscope).
2. **INEEDMD_RADIO_ENABLE_PIN** -
    a. Pin type output
    b. Active high
    c. Power on or off the radio.
3. **INEEDMD_XTAL_ENABLE_PIN** -
    a. Pin type output
    b. Active high
    c. Turns on or off the external oscillator.
4. **ATOD_ENABLE** -
5. **RADIO_LOW_BATT** -
6. **INEEDMD_RADIO_CMND_PIN** -
    a. Pin type output
    b. Active high

    c. This pin is tied to a Pin Input Output (PIO) on the wireless radio. Its function is to put the radio into command mode when needed to by the device system process. In command mode the radio will receive configuration commands from the system process and no longer receive or transmit data.
7. **RADIO_CD** - (carrier detect)
    a. Pin type input
    b. Active high
8. **RADIO_READY** -
    a. Pin type input
    b. Active high
9. **INEEDMD_RADIO_RESET_PIN** -
    a. Pin type output
    b. Active high
    c. This pin is tied to the software reset pin on the wireless radio.When the signal is set the radio goes into reset and will not resume operation until the signal goes low.
10. **INEEDMD_ADC_RESET_PIN** -
    a. Pin type output
    b. Active low
    c. This pin is an output and tied to the reset pin of the A to D converter. The pin signal is active low. It takes 18 CLK cycles to complete the reset.
11. **INEEDMD_ADC_PWR_PIN** -
    a. Pin type output
    b. Active low
    c. This pin is an output and tied to the power down pin on the external Analog to Digital converter. It is active low in that when the pin signal is low the A to D is powered off. When the pin signal is high the A to D is powered on.
12. **INEEDMD_ADC_INTERUPT_PIN** -
    a. Pin type input
    b. Active high
    c. This pin is an input and is configured to cause an interrupt when the A to D signals there is data ready to be read.
13. **INEEDMD_ADC_START_PIN** -
    a. Pin type output
    b. Active high
    c. This pin is an output and signals  to the A to D to begin conversions.
14. **INEEDMD_PORTA_ADC_nCS_PIN** -
    a. Pin type output
    b. Active high
    c. This is an output pin and is the chip select for the SPI interface to the A to D converter.

### 3.1.2 UART

*3.1.2.1 RADIO_UART*

The radio UART is the two way serial interface to the external wireless radio. The UART configuration uses hardware flow control and transmits data at high speeds. The wireless radio uses the same UART for both configuration and data transmission. Therefore it is necessary to have the device driver be aware of the received data frame and route it to the proper task.

*3.1.2.2 DEBUG_UART*

The debug uart is a one way serial interface that can communicate with an external serial terminal. This interface output debugging messages that can be used by developers to debug the system during development.

### 3.1.3 SPI

*3.1.3.1 ATOD*

The Analog to Digital interface uses the SPI connection to transmit configuration commands and receive data from the external device.

*3.1.3.2 SD_IO*

### 3.1.4 I2C

*3.1.4.1 LED_I2C*

### 3.1.5 Watchdog

*3.1.5.1 INMD_WATCHDOG*

### 3.1.6 USB

*3.1.6.1 USB0*

The processor only has one USB port and that is identified as USB 0.

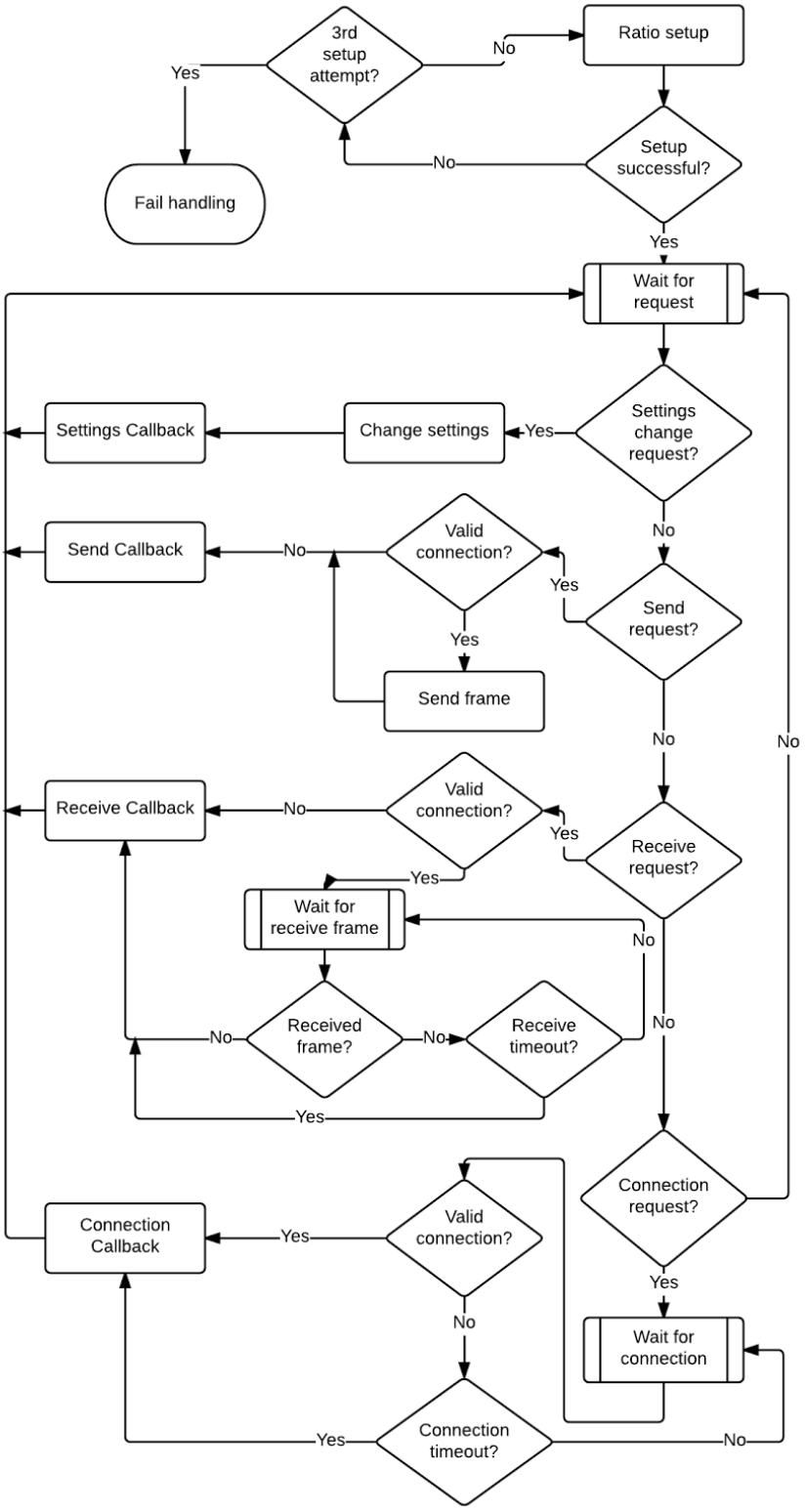
*3.2.6.2 USB port and pins*

1. INEEDMD_USB_GPIO_PORT
    a. INEEDMD_USBDP_PIN
    b. INEEDMD_USBDM_PIN

## 3.2 Device drivers

### 3.2.1 Bluetooth radio

The wireless bluetooth radio is the primary means of communication to the outside world for the device. It is controlled via a combination of GPIO and a single UART. The GPIO is used as a binary means of supplying power, software reset, connectivity notification and mode change. The UART is used to both configure the radio and transmit data.




Wireless radio flow diagram

At device startup the system assumes that the radio is not configured for the application. The system resets the radio to factory defaults to clear out any potentially erroneous changes that may have occurred to the radio. The system then proceeds to set the radio up for the application requirements.
During runtime the radio is both polled and monitored for data coming from OTA transmissions and/or GPIO interrupt input. Two types of data frames can be received from the radio: iWrap notification and Ineed MD command frames.

#### *3.2.1.1 radio settings*

#### *3.2.1.2 API*

##### 3.2.1.2.1 tRadio_request structure

1. eRequest; //radio request type
    a. *RADIO_REQUEST_POWER_ON* - Turns the radio on
    b. *RADIO_REQUEST_POWER_OFF* - Turns the radio off
    c. *RADIO_REQUEST_SEND_FRAME* - Sends a frame out the radio interface. uiBuff_size must be the length in bytes of the frame that was copied into uiBuff. If the vBuff_sent_callback parameter is not NULL the callback function will be called.
    d. *RADIO_REQUEST_RECEIVE_FRAME* - Sets the driver to receive a frame from the radio. The radio will automatically notify the appropriate task if a frame is received. uiBuff_size and uiBuff do not need to be set.
    e. *RADIO_REQUEST_WAIT_FOR_CONNECTION* - Puts the driver into a wait for radio/protocol connection state. uiTimeout is used to determine how long to wait for the connection.
    f. *RADIO_REQUEST_HALT_WAIT_FOR_CONNECTION* - Stops the driver from waiting for a radio/protocol connection. If the vConnection_status_callback parameter is not NULL the callback function will be called.
    g. *RADIO_REQUEST_BREAK_CONNECTION* - Forces the driver to break a radio/protocol connection
2. eSetting; //radio setting change
    a. *RADIO_SETTINGS_RFD* -
    b. *RADIO_SETTINGS_BAUD_115K* -
    c. *RADIO_SETTINGS_BAUD_1382K* -
    d. *RADIO_SETTINGS_APPLICATION_DEFAULT* -
3. eTX_Priority; //transmit priority
    a. *TX_PRIORITY_IMMEDIATE* - //will post frame to front of task queue
    b. *TX_PRIORITY_QUEUE* - //will post frame to back of queue
4. uiBuff_size - size of the data frame held in uiBuff. max is 256.
5. uiBuff - Data frame buff. Data to send needs to be copied into the buffer before doing the request.

6. uiTimeout - the number of milliseconds to wait before timing the request out. A value of 0 is passed the request will wait forever.
7. **void** (* vBuff_sent_callback)(uint32_t uiBytes_sent) - pointer to a function assigned by the requester to be called when a requested buffer is sent. The parameter is the number of bytes sent.
8. **void** (* vConnection_status_callback)(eRadio_connection_state eRadio_conn) - pointer to a function assigned by the requester to be called when a connection is made/broken. The parameter is the connection state of the radio.
    a. *RADIO_CONN_NONE* - No connection was made during the requested timeout period
    b. *RADIO_CONN_WAITING_FOR_CONNECTION* - radio is currently waiting for a connection.
    c. *RADIO_CONN_VERIFYING_CONNECTION* - radio is currently verifying the connection.
    d. *RADIO_CONN_CONNECTED* - radio has established a connection
    e. *RADIO_CONN_HALT* - radio has halted waiting for a connection
    f. *RADIO_CONN_ERROR* - radio has encountered an error and stopped

#### 3.2.1.2.2 eIneedmd_radio_request_params_init(tRadio_request * tParams)

Takes the address of a radio request structure as the parameter and initializes the elements to default values.

1. eRequest = *RADIO_REQUEST_NONE*
2. eSetting = *RADIO_SETTINGS_NONE*
3. eTX_Priority = *TX_PRIORITY_QUEUE*
4. uiBuff_size = 0
5. uiBuff = (buffer contents are set to 0)
6. uiTimeout = 0
7. vBuff_sent_callback = NULL
8. vConnection_status_callback = NULL

Returns the ok error code if the parameters were properly initialized.

#### 3.2.1.2.3 eGet_radio_connection_state()

Returns the current radio connection state. It can be called at any time.

1. *RADIO_CONN_UNKNOWN* - Radio has not been setup yet. Connection state is unknown.
2. *RADIO_CONN_NONE* - No connection was made during the requested timeout period
3. *RADIO_CONN_WAITING_FOR_CONNECTION* - radio is currently waiting for a connection.
4. *RADIO_CONN_VERIFYING_CONNECTION* - radio is currently verifying the connection.
5. *RADIO_CONN_CONNECTED* - radio has established a connection
6. *RADIO_CONN_HALT* - radio has halted waiting for a connection
7. *RADIO_CONN_ERROR* - radio has encountered an error and stopped

#### 3.2.1.2.4 eGet_radio_setup_state()

Returns the current radio setup state. It can be called at any time.

1. *RDIO_SETUP_NOT_STARTED* - Setup has not started
2. *RDIO_SETUP_ENABLE_INTERFACE* - Setup is enabling the low level interface to the radio.
3. *RDIO_SETUP_INTERFACE_ENABLED* - Setup has completed enabling the low level interface to the radio.
4. *RDIO_SETUP_START_POWER* - Setup is powering up the radio.
5. *RDIO_SETUP_FINISH_POWER* - The radio has finished powering up.
6. *RDIO_SETUP_START_SOFT_RESET* - Setup is soft resetting the ratio.
7. *RDIO_SETUP_FINISH_SOFT_RESET* - Soft reset has completed.
8. *RDIO_SETUP_START_RFD* - Setup is performing a reset to factory defaults on the radio.
9. *RDIO_SETUP_FINISH_RFD* - Radio RFD complete.
10. *RDIO_SETUP_READY* - Radio has completed setup and is ready to connect.
11. *RDIO_SETUP_RADIO_READY_POWERED_OFF* - Radio has completed setup but has been powered off to save power.
12. *RDIO_SETUP_ERROR* - Setup encountered an error setting up the radio
13. *RDIO_SETUP_UNKNOWN* - The radio is in an unknown setup state.

#### 3.2.1.2.5 eIneedmd_radio_request(tRadio_request * tParams)

Takes the address of a radio request structure as the parameter and posts it to the radio task. Returns the ok error code if the request was accepted.
If the request is to transmit a buffer the requesters buffer can be deallocated, overwritten or destroyed. The buffer value passed in the request is copied into the radios own transmit buffer. The function will return an error code depending on the request made and the current radio status

1. ER_OK: frame was queued or transmitted.
2. ER_FAIL: frame to send was not queued or transmitted due to message post to task failure.
3. ER_PARAM: pointer to the radio request structure is invalid.
4. ER_PARAM1: the priority parameter is invalid.
5. ER_OFF: the radio is currently turned off and a request to turn it on needs to be posted first.
6. ER_REQUEST: the request parameter was invalid.
7. ER_BUFF: the buffer pointer was invalid
8. ER_BUFF_SIZE: the size of the buffer was invalid

### *3.2.1.1 iWrap notifications*

iWrap notifications are simple ascii based frames from the radio to the device processor. There are four notification frames that we can expect from the radio syntax error and low battery. These notification frames will be parsed by the bluetooth radio driver and handled only by the driver for system wide notification.

1. **SYNTAX ERROR** - This notification frame is present only when there is an error with the iWrap command and its parameters transmitted to the radio.
2. **BATTERY LOW {voltage}** - This notification frame is posted by the radio only when the radio detects that its power source is below the low battery setting voltage. It is meant to notify the system that the power supply is low and at or below the {voltage} level reported.
3. **BATTERY SHUTDOWN {voltage}** - This notification frame is posted by the radio only when the radio detects that its power source is at or below the battery shutdown setting voltage. It it meant to notify the system that the power supply is currently at {voltage} and is insufficient to power the radio. The radio will then power down after this message and wait until the device has gone through power recovery.
4. **BATTERY FULL {voltage}** -This notification frame is posted by the radio only when the radio detects that the power source is at or above the battery full {voltage} setting.

#### *3.2.1.2 IneedMD command protocol*

IneedMD command protocol is the proprietary interface used to configure the system as a whole. The radio driver will verify the received frame is in IneedMD protocol format and then route the frame to the IneedMD protocol parser.

### 3.2.2 LED Driver

Settings

None

#### *3.2.2.1 API*

3.2.2.1.1 eIneedmd_LED_driver_setup()

3.2.2.1.2 ERROR_CODE eIneedmd_LED_pattern()

### 3.2.3 A to D

#### *3.2.3.1 Settings*

ADC Power Up/Power Down

Reference Internal/External

Referece On/Off

Sample Rate

Lead Detect On/Off

Signal Connect Disconnect

#### *3.2.3.2 API*

3.2.3.2.1 eineedmd_adc_Power_status()

Returns  the current power state  state of the ADC. It can be called at any time.

Returns type ERROR_CODE

1. *ER_ON* - Power up sequence has been complete.
2. *ER_OFF* - Power sequence had not been complete.

3.2.3.2.1 eineedmd_adc_Power_On()

Sets the  the current power state  state of the ADC to ON. It can be called at any time.

Returns type ERROR_CODE

1. *ER_ON* - Power up sequence has been complete.
2. *ER_FAIL* - Power sequence did not sucessfully complete.

Note that a RESET is performed by this function during the call of this function.

3.2.3.2.1 eineedmd_adc_Power_Off()

Powers down the ADC

Returns type ERROR_CODE

1. *ER_OFF* - Power down sequence has been complete.
2. *ER_FAIL* - Power down sequence had not been complete.

3.2.3.2.1 eineedmd_adc_Power_status()

Returns the current power state state of the ADC. It can be called at any time.

Returns type ERROR_CODE

1. *ER_ON* - Power up sequence has been complete.
2. *ER_OFF* - Power sequence had not been complete.

3.2.3.2.1 ineedmd_adc_12_Lead_Config()

Returns type ERROR_CODE on completion. Sets up the input for a 12 lead ADC input

Returns ERROR_CODE for the current power state state of the ADC. It can be called at any time.

Returns Type ERROR_CODE

1. *ER_FAIL* - The configuration was unsuccessful.
2. *ER_OK* - The configuration was successful.

The configuration routeen checks power state, reference configuration, measured the DC offset for all channels, then configures them for normal input. If measurements are in process they will be paused while the function runs.

3.2.3.2.1 needmd_adc_conversion_mode_get()

Returns the present conversion mode for the front end.

Returns Type CONVERSION_MODE

1. *CONVERSION_MODE_FAIL* - Indicated an unsuccessful attempt to read the conversion mode.
2. *CONVERSION_MODE_NON* - Indicates that no conversion mode is set.
3. *CONVERSION_MODE_SINGLE* - Indicated that the front end is set up to deliver a single measurement.
4. *CONVERSION_MODE_CONTINIOUS* - Indicated that the front end is setup to deliver a continuous stream of data..

3.2.3.2.1 ineedmd_adc_conversion_mode_set(CONVERSION_MODE)

Sets the present conversion mode for the front end.

Returns Type ERROR_CODE

- *ER_FAIL* - The configuration was unsuccessful.
- *ER_OK* - The configuration was successful.

Arguement Type CONVERSION_MODE

1. *CONVERSION_MODE_FAIL* - Causes a return or ER_FAIL
2. *CONVERSION_MODE_NON* - shuts down the front end conversion.
3. *CONVERSION_MODE_SINGLE* - Sets the front end to deliver a single measurement.
4. *CONVERSION_MODE_CONTINIOUS* - Sets the front end to deliver a single measurement.

3.2.3.2.1 eineedmd_adc_Rate_get()
Returns the present setting for the conversion rate.
Returns Type CONVERSION_RATE

1. *CONVERSION_RATE_FAIL* - Indicated an unsuccessful attempt to read the conversion rate.
2. *CONVERSION_RATE_8kSPS* - Indicates an 8k sps rate.
3. *CONVERSION_RATE_4kSPS* - Indicates an 8k sps rate.
4. *CONVERSION_RATE_2kSPS* - Indicates an 8k sps rate.
5. *CONVERSION_RATE_1kSPS* - Indicates an 8k sps rate.
6. *CONVERSION_RATE_500SPS* - Indicates an 8k sps rate.
7. *CONVERSION_RATE_250SPS* - Indicates an 8k sps rate.
8. *CONVERSION_RATE_125SPS* - Indicates an 8k sps rate.

3.2.3.2.1 eineedmd_adc_Rate_set(CONVERSION_RATE)
Sets the conversion rate for the ADC
Returns Type ERROR_CODE

1. *ER_FAIL* - The configuration was unsuccessful.
2. *ER_OK* - The configuration was successful.

Returns Type CONVERSION_RATE

1. *CONVERSION_RATE_FAIL* - Indicated an unsuccessful attempt to read the conversion rate.
2. *CONVERSION_RATE_8kSPS* - Indicates an 8k sps rate.
3. *CONVERSION_RATE_4kSPS* - Indicates an 8k sps rate.
4. *CONVERSION_RATE_2kSPS* - Indicates an 8k sps rate.
5. *CONVERSION_RATE_1kSPS* - Indicates an 8k sps rate.
6. *CONVERSION_RATE_500SPS* - Indicates an 8k sps rate.
7. *CONVERSION_RATE_250SPS* - Indicates an 8k sps rate.
8. *CONVERSION_RATE_125SPS* - Indicates an 8k sps rate.

3.2.3.2.1 ineedmd_adc_reference_gett()
Sets the present conversion mode for the front end.
Returns Type REFERENCE_MODE

5. *REFERENCE_MODE_FAIL* - Referemce Mode Set failed
6. *REFERENCE_OFF* - The references are shut down.
7. *REFERENCE INTERNAL* - The Internal reference is being used.
8. *REFERENCE EXTERNAL* - The external reference is being used.

3.2.3.2.1 ineedmd_adc_conversion_mode_set(CONVERSION_MODE)
Sets the present conversion mode for the front end.
Returns Type ERROR_CODE

- *ER_FAIL* - The configuration was unsuccessful.
- *ER_OK* - The configuration was successful.

Arguement Type CONVERSION_MODE

9. *REFERENCE_MODE_FAIL* - Returns ER FAIL if called
10. *REFERENCE_OFF* - Switches off the reference and powers down the buffer.
11. *REFERENCE INTERNAL* - Sets the internal reference ON and to 2.5V, shutting down the external reference.
12. *REFERENCE EXTERNAL* - Sets External reference to ON and shuts down the internal reference.measurement.

3.2.3.2.1 ineedmd_adc_leadoff_gett()

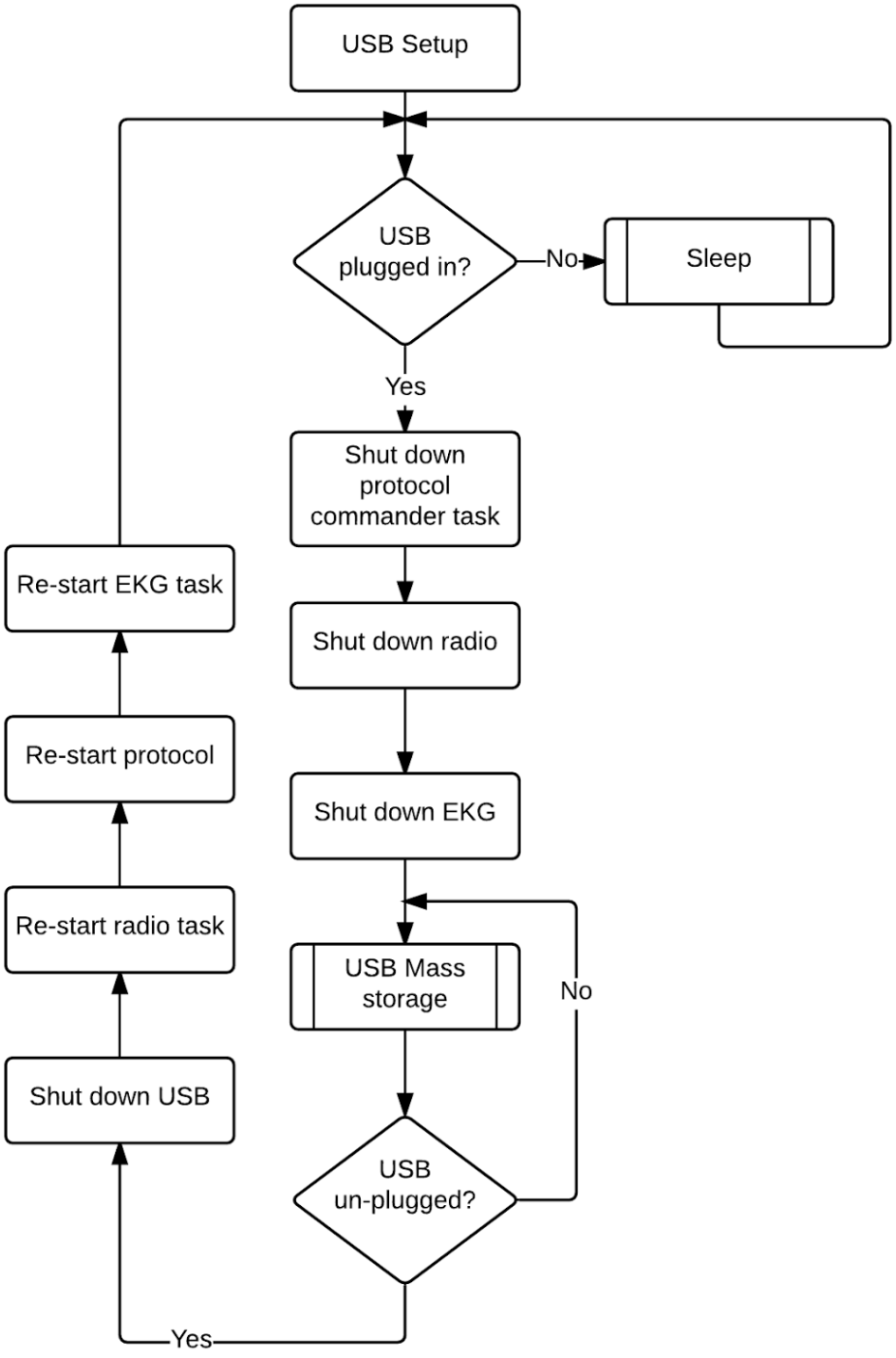
Returns a 16 bit representation of the connected leads.

### 3.2.4 SD card

### 3.2.5 Speaker driver

### 3.2.6 USB Mass storage

*3.2.6.1 USB task*




USB task flow diagram

*3.2.6.2 API*

3.2.6.2.1 tUSB_req

1. eRequest
    a. *USB_REQUEST_REGISTER_CONN_CALLBACK* - Register the connection callback function with the USB driver. vUSB_connection_callback must not be null.
    b. *USB_REQUEST_UNREGISTER_CONN_CALLBACK* - Unregister the connection callback function with the USB driver. vUSB_connection_callback must not be null.
    c. *USB_REQUEST_FORCE_DISCONNECT* - Forces the USB to shut down and disconnect.
    d. *USB_REQUEST_RECONNECT* - Requests the USB to re-establish a connection. Only works if there was a previous USB connection made.
2. **void** (* vUSB_connection_callback) ( **bool** bUSB_Physical_connection,
   **bool** bUSB_Data_connection);
    a. Callback function called when the USB connection status changes.
    b. Up to 3 callbacks can be registered with the USB driver.
    c. The parameters are two separate booleans
        i. bUSB_Physical_connection - set to true if a physical connection is made
        ii. bUSB_Data_connection - is set to true if a data connection is made

3.2.6.2.2 eUSB_request_params_init(tUSB_req * tParams)

1. eRequest = *USB_REQUEST_NONE*
2. vUSB_connection_callback = NULL

Returns an error code when the param init function is complete.

1. ER_OK: Params were initialized.
2. ER_FAIL: Params were not initialized.

3.2.6.2.3 eUSB_request(tUSB_req * tRequest)

Takes the address of a radio request structure as the parameter and posts it to the radio task.
Returns an error code when the request is done.

1. ER_OK: USB request was accepted.
2. ER_FAIL: request was not queued or due to message post to task failure.
3. ER_PARAM: pointer to the request structure is invalid.
4. ER_REQUEST: the request parameter was invalid.
5. ER_PARAM1: the callback parameter is invalid.
6. ER_FULL: the number of callbacks registered is at its limit.
7. ER_NOT_SET: the callback requested to unregister was not present in the callback list.

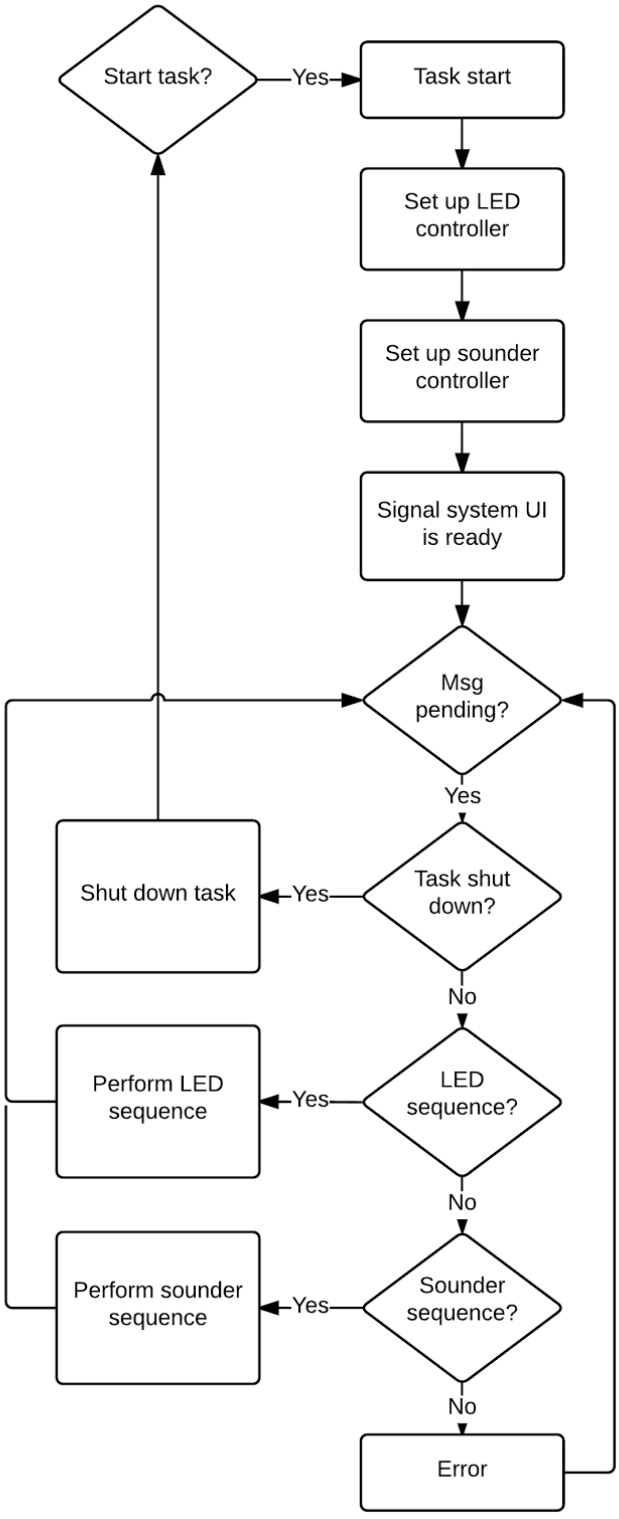
# 4. Application

## 4.1 User interface

The user interface will be a low priority task in the overall system process. Its responsibility is to receive system change or notification commands that will affect the UI and display or sound the proper UI sequence.

### 4.1.1 User interface task

The task has direct control over the UI elements. It will be responsible for initiating the setup of the high level device driver for LED control and the sounder. Once the setup is complete the UI task will signal it is ready to receive messages.




UI task flow chart

### 4.1.2 User Interface API

#### *4.1.2.1 tUI_request structure*

When requesting a UI change the parameters for the change will be passed in using the tUI_request structure. The structure contains which UI element to sequence, and the sequence ID. Multiple UI elements can be OR'ed together in a single request but only one sequence per element can be performed.

```
tStruct.uiUI_element = (INMD_UI_ELEMENT_HEART_LED | INMD_UI_ELEMENT_COMMS_LED);
tStruct.eHeart_led_sequence = HEART_LED_LEAD_ON;
tStruct.eComms_led_sequence = COMMS_LED_USB_CONNECTION_SUCCESSFUL;
```

**Correct coding**

```
tStruct.uiUI_element = INMD_UI_ELEMENT_HEART_LED;
tStruct.eHeart_led_sequence = (HEART_LED_LEAD_ON | HEART_LED_DIGITAL_FLATLINE);
```

**Incorrect coding**

1. uiUI_element - an unsigned int that can be OR'ed with any 4 of the current UI elements to change.

a. **INMD_UI_ELEMENT_HEART_LED** - visual signal relevant patient data
b. **INMD_UI_ELEMENT_COMMS_LED** - visual signal current comms status
c. **INMD_UI_ELEMENT_POWER_LED** - visual signal current power status
d. **INMD_UI_ELEMENT_SPEAKER** - audio signal alert status

2. eAlert_sound - enumerated type eALERT_SOUND_UI for what sound to play. It can only be a single value from the enum and not a combination.

a. ALERT_SOUND_NO_UI - speaker will perform no change action
b. ALERT_SOUND_UI_OFF - turns the speaker off

3. eHeart_led_sequence - enumerated type eHEART_LED_UI for what heart led sequence to display. It can only be a single value from the enum and not a combination.

a. HEART_LED_NO_UI - heart LED will perform no change action
b. HEART_LED_UI_OFF - turn the heart LED off
c. HEART_LED_LEAD_OFF_NO_DATA - display the leads off sequence
d. HEART_LED_LEAD_ON - display the leads on sequence
e. HEART_LED_DIGITAL_FLATLINE - display flatline sequence

4. eComms_led_sequence - enumerated type eCOMMS_LED_UI for what communication status sequence to display. It can be a single value from the enum and not a combination.

a. COMMS_LED_NO_UI – comm LED will perform no change action
b. COMMS_LED_UI_OFF – turn the comms LED off
c. COMMS_LED_BLOOTHOOTH_CONNECTION_SUCESSFUL – comm led will display Bluetooth connection successful sequence
d. COMMS_LED_BLUETOOTH_PAIRING – comm LED will display pairing sequence
e. COMMS_LED_BLUETOOTH_PAIRING_FAILIED – comm LED will display pairing failed sequence

f. COMMS_LED_USB_CONNECTION_SUCCESSFUL – comm LED will display USB connection successful sequence
g. COMMS_LED_USB_CONNECTION_FAILED – comm LED will display usb connection failed sequence
h. COMMS_LED_DATA_TRANSFERING_FROM_DONGLE – display the data transferring from dongle LED sequence
i. COMMS_LED_DATA_TRANSFER_SUCESSFUL – display the data transfer successful LED sequence
j. COMMS_LED_DONGLE_STORAGE_WARNING – display the dongle storage warning LED sequence
k. COMMS_LED_ERASING_STORED_DATA – display the erasing stored data LED sequence
l. COMMS_LED_ERASE_COMPLETE – display the erase complete LED sequence
5. ePower_led_sequence - enumerated type ePOWER_LED_UI for what power status sequence to display. It can be a single value from the enum and not a combination.
a. POWER_LED_NO_UI – Power LED will perform no change action
b. POWER_LED_UI_OFF – turn the Power LED off
c. POWER_LED_POWER_ON_90to100 – display the power on, battery level 90 to 100 percent sequence
d. POWER_LED_POWER_ON_50to90 - display the power on, battery level 50 to 90 percent sequence
e. POWER_LED_POWER_ON_25to50 - display the power on, battery level 25 to 50 percent sequence
f. POWER_LED_POWER_ON_0to25 - display the power on, battery level 0 to 25 percent sequence
g. POWER_LED_POWER_ON_CHARGE_90to100 – display power on, charging, battery level 90 to 100 percent sequence
h. POWER_LED_POWER_ON_ CHARGE _50to90 - display power on, charging, battery level 50 to 90 percent sequence
i. POWER_LED_POWER_ON_ CHARGE _25to50 - display power on, charging, battery level 25 to 50 percent sequence
j. POWER_LED_POWER_ON_ CHARGE _0to25 - display power on, charging, battery level 0 to 25 percent sequence
k. POWER_LED_POWER_ON_BLIP_90to100 - display power on, battery status blip, battery level 90 to 100 percent sequence
l. POWER_LED_POWER_ON_BLIP_50to90 - display power on, battery status blip, battery level 50 to 90 percent sequence
m. POWER_LED_POWER_ON_BLIP_25to50 - display power on, battery status blip, battery level 25 to 50 percent sequence
n. POWER_LED_POWER_ON_BLIP_0to25 - display power on, battery status blip, battery level 0 to 25 percent sequence

4.1.2.2 eIneedmd_UI_params_init(tUI_request *)

Used to initialize a tUI_request to the default values below.

1. uiUI_element = ALERT_SOUND_NO_UI
2. eAlert_sound = COMMS_LED_NO_UI
3. eHeart_led_sequence = HEART_LED_NO_UI
4. eComms_led_sequence = COMMS_LED_NO_UI
5. ePower_led_sequence = POWER_LED_NO_UI

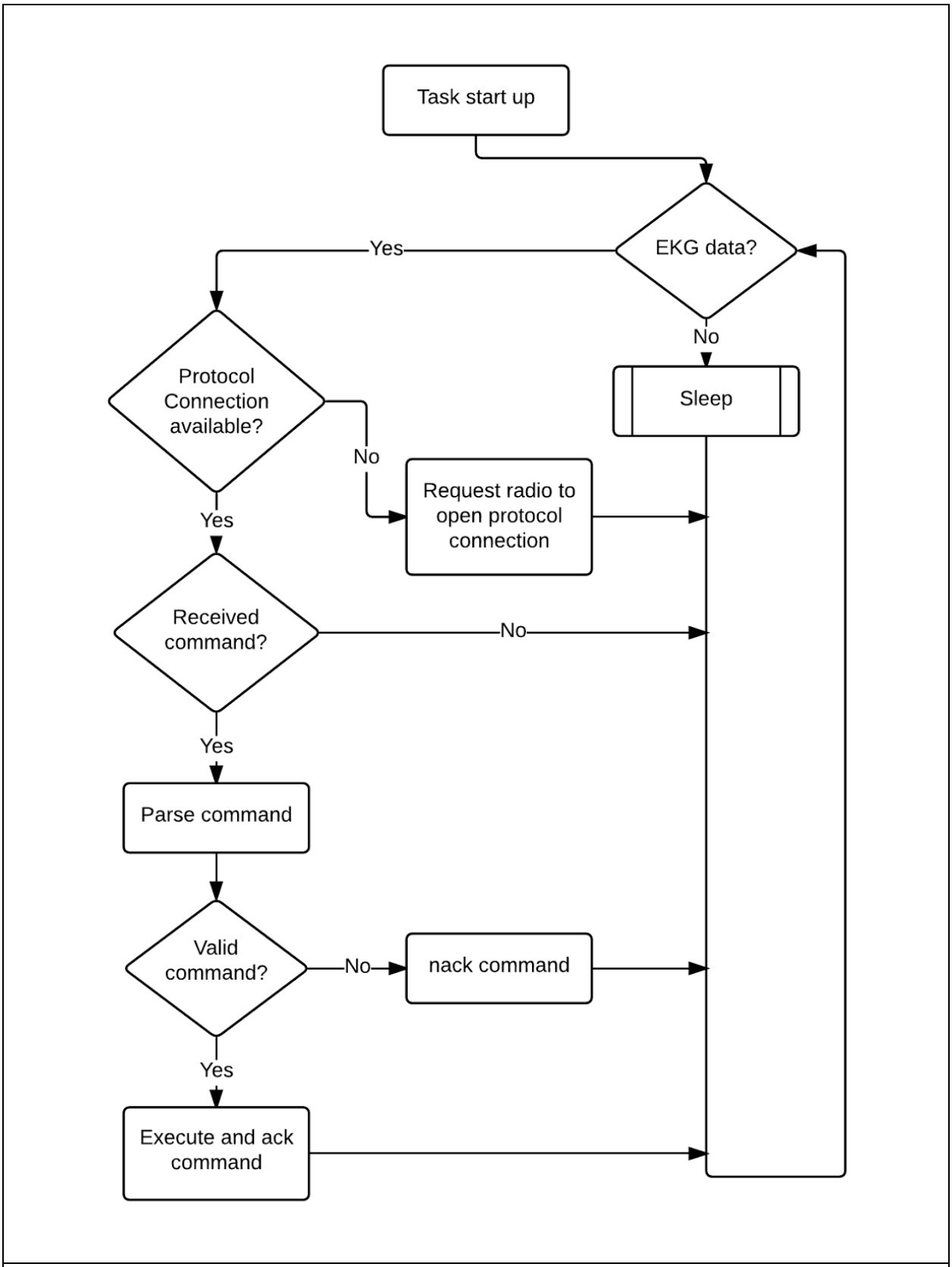
#### *4.1.2.3 eIneedmd_UI_request(tUI_request *)*

Used to post a UI change to the UI task. Once a tUI_request structure is setup, call this function to post the UI change.

## 4.2 Command protocol

The command protocol task is responsible for receiving IneedMD command protocol frames and performing the requested action.

### 4.2.1 Command protocol task




Command protocol task flow chart

### 4.2.2 Command protocol task API

#### *4.2.2.1 tINMD_protocol_req_notify structure*

The parameter passed to the command protocol parser is a structure containing the following elements.

1. eReq_Notify
    a. *CMND_REQUEST_WAKEUP* - Single command, the other parameters will be ignored. Performs a system wide wakeup procedure to prepare for data reading and transmission.
    b. *CMND_REQUEST_SLEEP* - Single command, the other parameters will be ignored. Performs a system wide sleep procedure. Will power down and or hibernate peripherals.
    c. *CMND_REQUEST_RCV_PROTOCOL_FRAME* - Single command, frame pointer and frame length much be present or will be returned as an error. Receives the protocol frame, parses the frame and performs the protocol command if valid.
    d. *CMND_NOTIFY_INTERFACE_ESTABLISHED* - Single command, the other parameters will be ignored. Notifies the protocol analyzer that a protocol capable interface is available.
    e. *CMND_NOTIFY_PROTOCOL_INTERFACE_ESTABLISHED* - Single command, the other parameters will be ignored. Notifies the protocol analyzer that a protocol interface connection was established.
    f. *CMND_NOTIFY_PROTOCOL_INTERFACE_CLOSED* - Single command, the other parameters will be ignored. Notifies the protocol analyzer that the protocol interface connection was closed.
2. * uiCmnd_Frame - pointer to the buffer containing the protocol command frame.
3. uiFrame_Len - length of the command frame in number of bytes including header and footer.

#### *4.2.2.2 eIneedmd_cmnd_Proto_ReqNote_params_init() funciton*

Takes the address of a command protocol request structure as the parameter and initializes the elements to default values.

1. eReq_Notify = *CMND_REQUEST_NONE*
2. * uiCmnd_Frame = NULL
3. uiFrame_Len = 0

#### *4.2.2.3 eIneedmd_cmnd_Proto_Request_Notify function*

This function is the main API communication function for the protocol analyzer task. Other than callbacks this should be the only function called from outside of this module. The parameter is an address to a tINMD_protocol_req_notify structure. The function will return an error code notifying the caller of the result.

1. *ER_OK* - request/notification accepted
2. *ER_FAIL* - request/notification failed to post to mailbox queue.
3. *ER_PARAM* - pointer to parameter structure passed in is invalid
4. *ER_LEN* - command frame length is wrong.

5. *ER_SIZE* - command frame length is too large.
6. *ER_COMMAND* - command is malformed.
7. *ER_INVALID* - command pointer is invalid

## 4.3 EKG monitoring

The EKG task will monitor the EKG A to D interface. Its main purpose is to read the A to D data, wrap the data in a data transmit frame. The data is then both/either/or sent to the radio for transmission and file storage for saving.

### 4.3.1 EKG task

EKG monitor task flow diagram

#### 4.3.2 EKG task API

*tINMD_EKG_req structure*

1. eEKG_Request_ID
    a. EKG_REQUEST_SHORTING_BAR,
    b. EKG_REQUEST_DFU,
    c. EKG_REQUEST_SHIPPING_HOLD,
    d. EKG_REQUEST_TEST_PATTERN,
    e. EKG_REQUEST_EKG_MONITOR,
2. void (* vEKG_read_callback) (uint32_t uiEKG_data, bool bEKG_read)

*ERROR_CODE eIneedmd_EKG_request_param_init(tINMD_EKG_req * ptRequest);*
*ERROR_CODE eIneedmd_EKG_request(tINMD_EKG_req * ptRequest);*

### 4.4 Battery monitoring

Battery monitoring task will periodically check the power supply voltage. This monitoring task will determine if the battery has sufficient power to operate the device or if a different power mode needs to be selected.

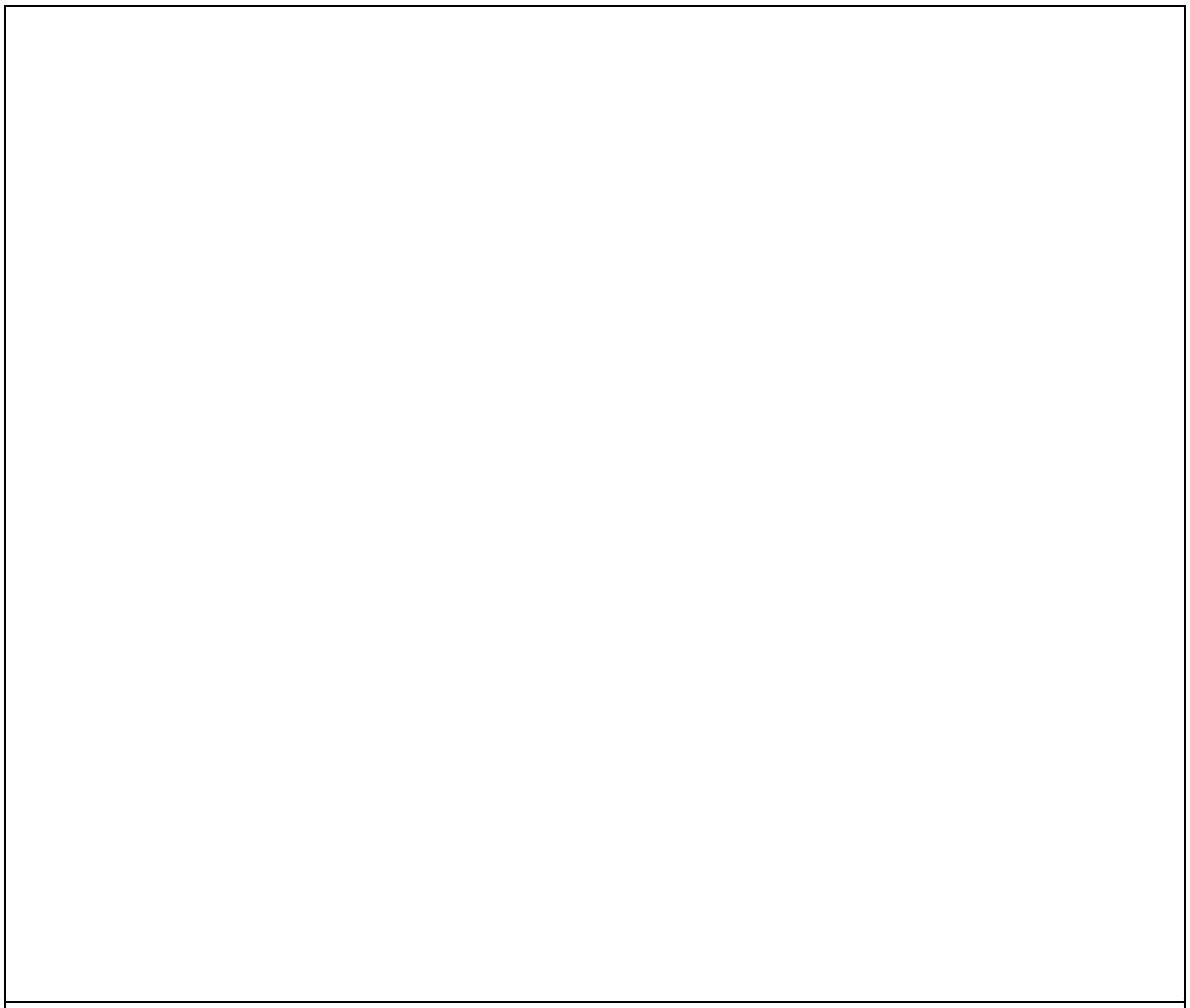
## 5. 3rd party software

### 5.1 File system FatFS

TI-RTOS comes with a ready built implementation of FatFS. It is a widely used file system for embedded applications. It has its own fully abstracted HAL and can easily be integrated into a file system wrapper for further abstraction. It's primary means of file storage will be on an SD card via the SPI interface driver.

### 5.2 Real time operating system TI-RTOS

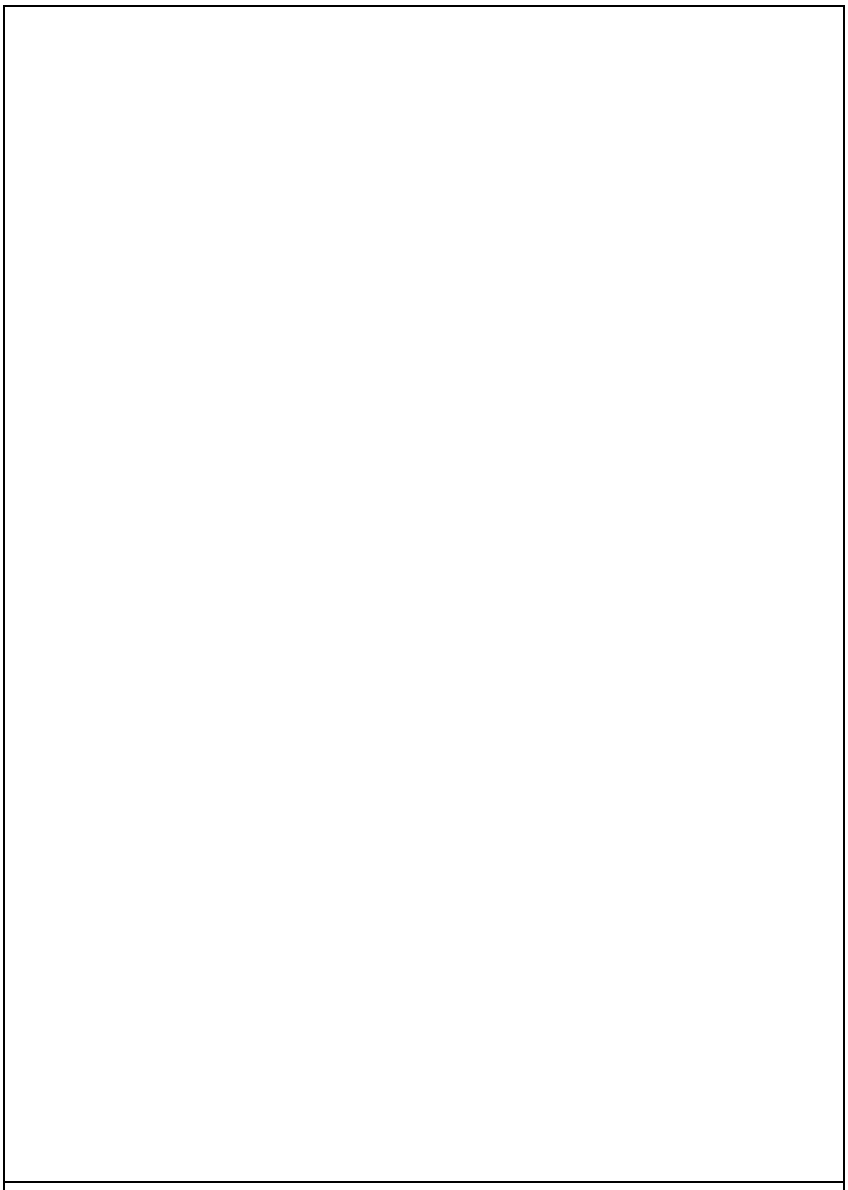
TI-RTOS is a real time operating system created by Texas Instruments. It was compared to other RTOS's that were available and found to be the one of choice for the medical device. TI-RTOS uses a GUI to design the overall RTOS functionality, bring in device drivers and include 3rd party software.



TI-RTOS system overview GUI for including device drivers

### 5.2.1 TI-RTOS Scheduling

Scheduling is where task, interrupts, clocks and idle tasks are created and added to the RTOS scheduler. Scheduling is added to the project via the TI-RTOS gui in the Available Products window under TI-RTOS>Products>SYSBIOS>Scheduling.



Adding tasks to TI-RTOS

For the device firmware all of the available scheduling implementations will be used.

*5.2.1.1 Task*

The Task scheduler is where the base process will be created. It will include the command parser, UI, some drivers, and data handling.

*5.2.1.2 Swi*

Software interrupts in the scheduler will be used to notify the system of data read or write completion. This will be primarily used in data handling and file storage.

*5.2.1.3 Hwi*

Hardware interrupts will be used to notify the system of transmission requests, data processing and interface connectivity.

*5.2.1.4 Idle*

The idle task will be where the system attempts to conserve power and handle the watch dog.

#### *5.2.1.5 Clock*

Real time clocking is needed for time stamping the data that will be processed.

#### *5.2.1.6 Seconds*

Time alive will be needed for battery monitoring processing.

#### *5.2.1.7 Timer*

The timer will be needed for live data connection and battery conservation.

### 5.2.2 RTOS Task Priorities

Ordered for high numbers for high priority to low numbers for low priority.

| | Priority | Task |
|---|---|---|
| | 16 | |
| | 15 | |
| | 14 | USB |
| | 13 | Radio Uart |
| | 12 | EKG waveform |
| | 11 | Pwr Management |
| | 10 | |
| | 9 | |
| | 8 | |
| | 7 | |
| | 6 | |
| | 5 | |
| | 4 | Protocol Commander |
| | 3 | |
| | 2 | UI |
| | 1 | |