MrT

Jason Berger

Aug 10, 2023

USING MRT:

CHAPTER

ONE

GETTING STARTED

This section of the document gives a basic overview of installing and using the modules

1.1 Installation

The code modules themselves are imported as submodules, so there are no libraries that need to be installed. But there is a toolset `mrtutils` which makes it easier to manage the modules.

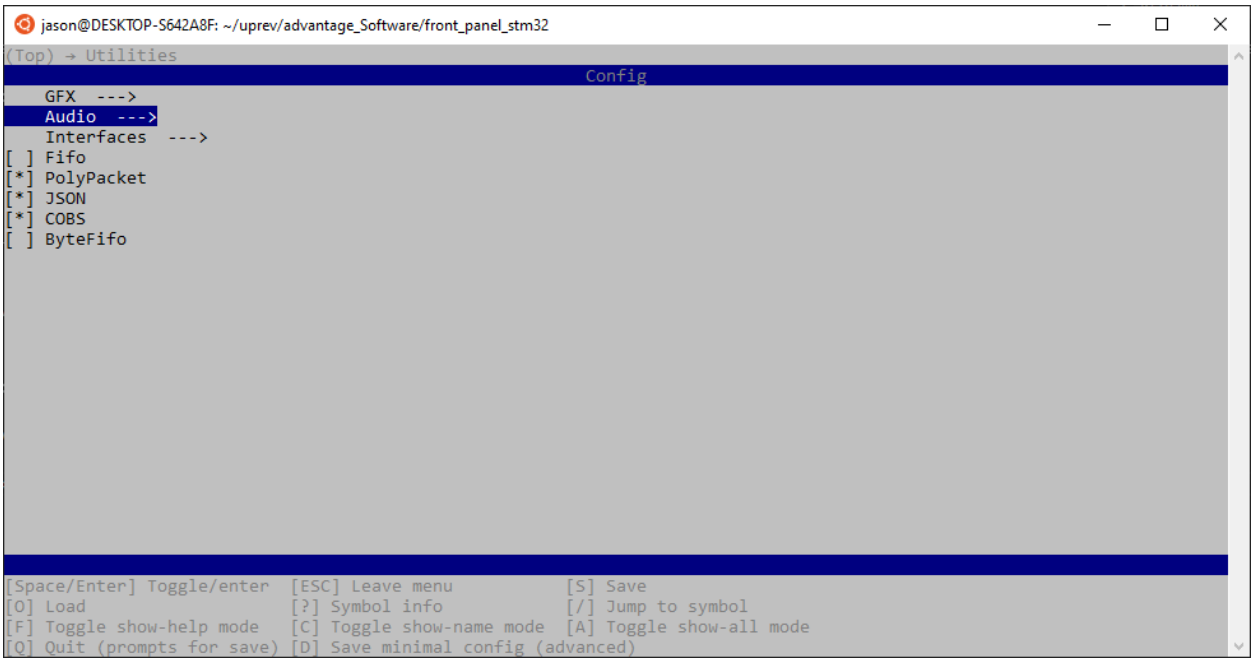
```
pip install mrtutils
```

1.2 Integrating MrT into your project

```
cd <path/to/project>

mrt-config <relative/path/for/MrT/root>
```

Note: If no path is provided, it will default to ./MrT and create the directory if it does not exist

This will open the `mrt-config` tool which allows you to select which modules you would like to integrate into your project. The UI is based on *menuconfig* to be as flexible as possible in terms of where you can run it, ie in containers or remote development environments over ssh.

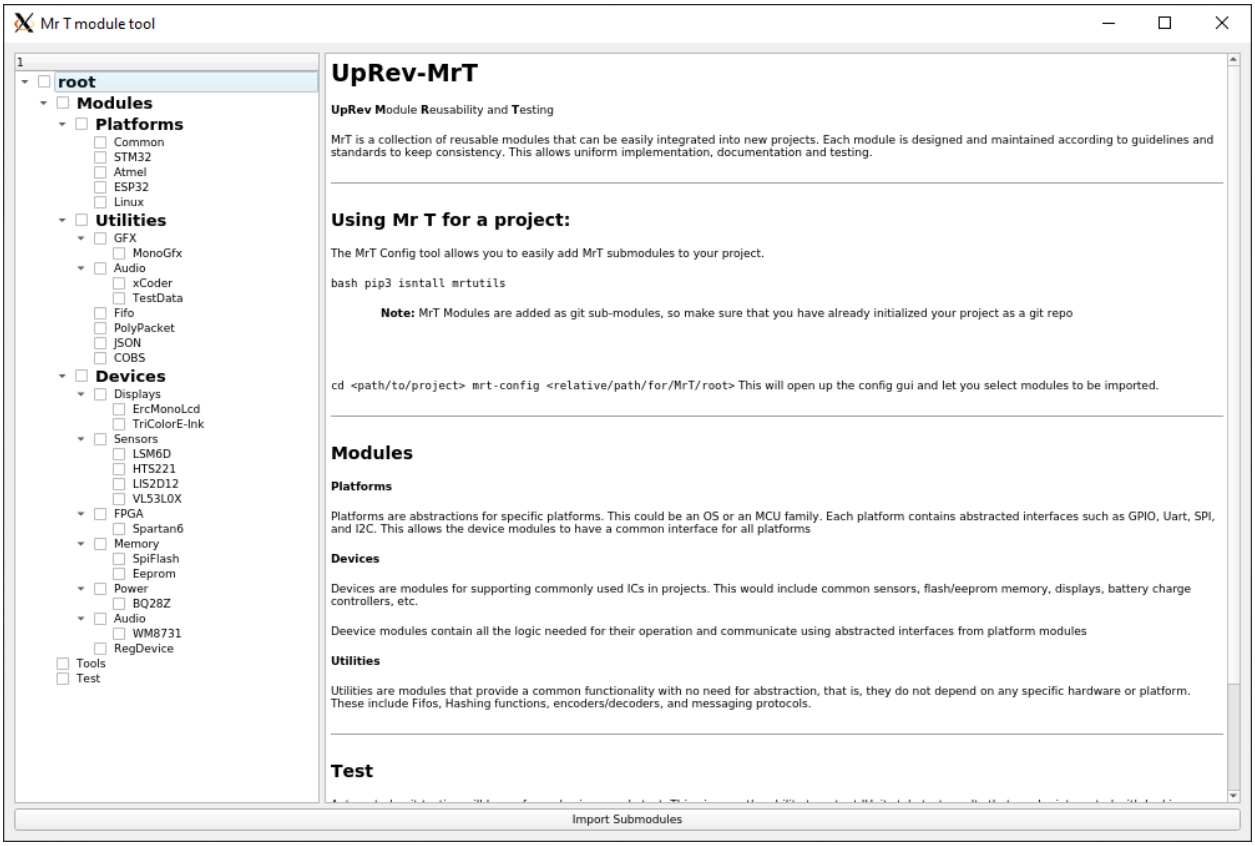


Note: MrT Modules are added as git sub-modules, if you are in a directory that does not contain a git repo, it will initialize one.

mrt-config-gui

If you prefer to use a gui interface, you can use the pyQt5 based `mrt-config-gui`:

```
mrt-config-gui <relative/path/for/MrT/root>
```


Mr T module tool
root
Modules
Platforms
Common
STM32
Atmel
ESP32
Linux
Utilities
GFX
MonoGfx
Audio
xCoder
TestData
Fifo
PolyPacket
JSON
COBS
Devices
Displays
ErcMonoLcd
TriColorE-Ink
Sensors
LSM6D
HTS221
LIS2D12
VL53L0X
FPGA
Spartan6
Memory
SpiFlash
Eeprom
Power
BQ28Z
Audio
WM8731
RegDevice
Tools
Test
UpRev-MrT
UpRev Module Reusability and Testing
MrT is a collection of reusable modules that can be easily integrated into new projects. Each module is designed and maintained according to guidelines and standards to keep consistency. This allows uniform implementation, documentation and testing.
Using Mr T for a project:
The MrT Config tool allows you to easily add MrT submodules to your project.
bash pip3 isntall mrtutils
Note: MrT Modules are added as git sub-modules, so make sure that you have already initialized your project as a git repo
cd <path/to/project> mrt-config <relative/path/for/MrT/root> This will open up the config gui and let you select modules to be imported.
Modules
Platforms
Platforms are abstractions for specific platforms. This could be an OS or an MCU family. Each platform contains abstracted interfaces such as GPIO, Uart, SPI, and I2C. This allows the device modules to have a common interface for all platforms
Devices
Devices are modules for supporting commonly used ICs in projects. This would include common sensors, flash/eeprom memory, displays, battery charge controllers, etc.
Deevice modules contain all the logic needed for their operation and communicate using abstracted interfaces from platform modules
Utilities
Utilities are modules that provide a common functionality with no need for abstraction, that is, they do not depend on any specific hardware or platform. These include Fifos, Hashing functions, encoders/decoders, and messaging protocols.
Test
Import Submodules

CHAPTER

TWO

TUTORIAL

This is a guide for incorporating MrT modules into a project. The guides walks through the full implementation of a project using MrT, a generated device driver, and a custom messaging protocol. This guide will be broken up into stages.

The project files are all in the mrt-tutorial repo and there is a '*reference*' branch with Tags showing the end of each stage

At head of the master branch is the project start. An STM32 project has already been created to target the STM32L4 (Their IOT node dev board)

- Uart1: 115200 baud
- I2C2
- GPIO PB14 as output, labeled as LED_GRN

Note: The setup for this project is not in the scope of this tutorial, but using STM32CUBE is pretty well documented online

2.1 Step 1: Installing tools

MrT modules are just individual git repositories that get included in your project as submodules . You could simply add them as submodules manually, but this would require looking up the urls, and making sure the path to each module is correct, because some modules reference others.

To make this easier, you can use the mrt-config tool from the mrttutils package.

mrttutils is a python package managed with pip

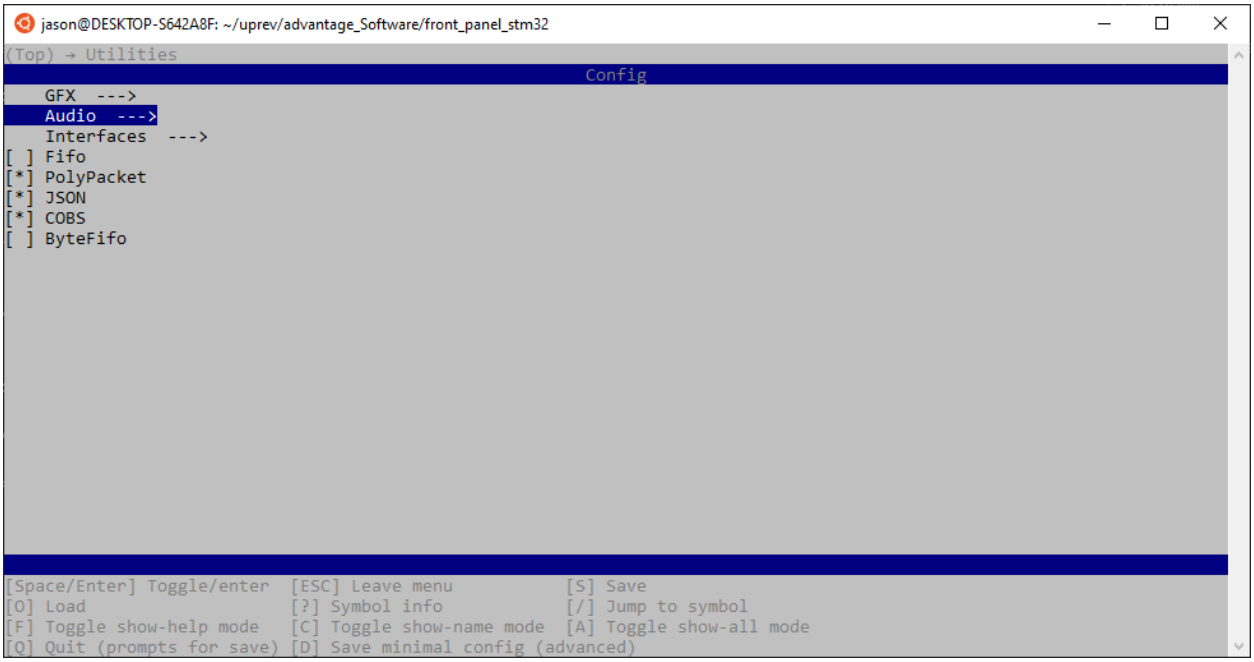
```
pip3 install mrtutils
```

2.2 Step 2: Add MrT Modules

Once you have installed mrtutils, adding modules is very simple. just run mrt-config and tell it where you want to put the modules *(It will create the directory)*

```
cd /path/to/mrt-tutorial
mrt-config MrT
```

This will open the mrt-config gui:



This tool will open a `menuconfig` style UI that lets you browse the available modules and select the ones you want to include

For Now select the following modules:

Platforms/STM32
: this is the absctraction layer for STM32 MCUs. it provides definitions/Macros to map hardware interaction with the STM32 HAL

Platforms/Common
: this module is required when using any platform abstraction layer

Devices/RegDevice
: This is the base module for generic register based devices. It is needed later in [Creating Device Driver using mrt-device tool](#mrt-device)

Once you have selected the required modules, press `q` to quite, then `y` when prompted to save changes

You should now have a folder called 'MrT' in your projects directory with the submodules inside of it.

Now you need to configure the project to use these submodules. Each platform module should have instructions in its README.

Here are the instructions from STM32/README.md :

Note: after importing modules, right click the project and hit refresh so it sees the new directories

To use the STM32 platform, cofigure the following settings:

Project->Properties->C/C++ General->Path and Symbols : * Under the Symbols tab add a symbol named **MRT_PLATFORM** with the value **MRT_STM32_HAL** * Under the Source Location tab click add and select the **Modules** directory under Mr T* * Under the Includes tab, click add and add the path to the **Modules** directory under Mr T

Build the project

2.3 Step 3: Toggle LED

Now we can use the MrT abstraction layer for stm32. We are going to blink the LED on the board just as a basic example. Add the following code snippets:

main.c:26 (in the USER CODE INCLUDES section)

```
#include "Platforms/Common/mrt_platform.h" /* This will include the stm32 layer based on␣
↪the MRT_PLATFORM symbol we set*/
```

main.c:108 (in the USER CODE WHILE section)

```
/** STM32 HAL does not have a type for pins, all of its functions use (port,pin). MRT_
↪GPIO() is a macro that wraps them
  * This is so that device drivers have a single struct for pins
  */
MRT_GPIO_WRITE(MRT_GPIO(LED_GRN),HIGH);     //set the pin high
MRT_DELAY_MS(1000);                         //wait 1000 ms
MRT_GPIO_WRITE(MRT_GPIO(LED_GRN),LOW);      //set the pin low
MRT_DELAY_MS(1000);                         //wait 1000 ms
```

Now build and run the project, the green LED on the board should blink!

2.4 Step 4: Create a device driver

Obviously an abstraction layer to toggle a gpio is a bit overkill. But the point of this is to write device drivers that can run on any platform. So now we are going to create a device driver for the HTS221 temperature and humidity sensor on the board.

For this we will use the mrt-device tool

This is part of mrtutils, so it is already installed.

Normally you would create a device driver as a submodule, so that it can be re-used as a MrT module, but for the purpose of this tutorial we will just create it in a subdirectory. mrt-device can generate a template to get you started:

```
mkdir MrT/Modules/Devices/hts221
cd MrT/Modules/Devices/hts221
mrt-device -t my_device
```

now you should have a new file 'my_device.yml' to fill out. in the 'doc' folder there are 2 files to looks at:

- **device.yml** - this is the yaml file for the device driver that I already created
- **hts221.pdf** - this is the section of the datasheet that describes the registers.

Comparing the two files and referencing the mrt-device wiki should help you get an idea of how to structure the file. *(A lot of the information at the top is not really needed, but good for documentation)*

Once you feel comfortable with the structure, generate the driver:

```
mrt-device -i my_device.yml -o .
```

This will create 3 new files:

- **hts221.h** - header for driver
- **hts221.c** - source for driver
- **hts221_regs.h** - various symbols and macros for device registers

  adding the -d flag will generate documentation:

  ```
  mrt-device -i my_device.yml -o . -d .
  ```

Now we have a basic driver with access to all of the register/fields in the device. If the temperature and humidity values could be read directly, wed be done.. But they cant. So we just need to add the logic.

> This particular device has a pretty convoluted calibration table that has to be read to get conversion constants. You can ignore the logic involved, the take away is that there are code blocks in the driver that will **not** be overwritten if you regenerate the driver. It also shows use of the devices macros for reading fields/registers

First we are going to add some properties to the device struct:

hts221.h:85 *between the user-block-struct tags* :

```
  int mPrevTemp;
  int mPrevHum;

  struct{
    int16_t T0_out;
    int16_t T1_out;
    int16_t T0_degC;
    int16_t T1_degC;
    uint8_t H0_rH;
    uint8_t H1_rH;
    int16_t H0_T0_OUT;
    int16_t H1_T0_OUT;
  } mCalData;
```

Next add functions for reading temperature and humidity :

hts221.h:100 *between the user-block-bottom tags* :

```
/**
 * @brief reads humidity from device
 * @param dev ptr to hts221 device
 * @return relative humidity in 1/100th of a percent. i.e. 4520 = %45.2
```

(continues on next page)

(continued from previous page)

```
 */
int hts_read_humidity(hts221_t* dev);


/**
 * @brief reads temperature from device
 * @param dev ptr to hts221 device
 * @return temperature in 1/100th of a degress C. i.e. 2312 = 23.12 C
 */
int hts_read_temp(hts221_t* dev);
```

Add the code to get calibration constants from calibration table :

hts221.c:40 *in the user-block-init section* :

```
 dev->mPrevHum =0;
    dev->mPrevTemp =0;

    /* device requires a bit ORd with register address to auto increment reg addr */
    dev->mRegDev.mAutoIncrement = true;
    dev->mRegDev.mAiMask = 0x80;

    /* Load calibration data */
     uint8_t H0_rh_x2, H1_rh_x2, T0_degC_x8, T1_degC_x8, T1T0_msb;

    /* These registers can be read directly into the cal values */
    dev->mCalData.H0_T0_OUT = hts_read_reg(dev, &dev->mH0T0Out);
    dev->mCalData.H1_T0_OUT = hts_read_reg(dev, &dev->mH1T0Out);
    dev->mCalData.T0_out = hts_read_reg(dev, &dev->mT0Out);
    dev->mCalData.T1_out = hts_read_reg(dev, &dev->mT1Out);

    /* These registers need to be processed to get the values we need */
    H0_rh_x2 = hts_read_reg(dev, &dev->mH0RhX2);
    H1_rh_x2 = hts_read_reg(dev, &dev->mH1RhX2);
    T0_degC_x8 = hts_read_reg(dev, &dev->mT0DegcX8);
    T1_degC_x8 = hts_read_reg(dev, &dev->mT1DegcX8);
    T1T0_msb = hts_read_reg(dev, &dev->mT1t0Msb);

    /* These values just need to be divided down (for some reason they are stored with a␣
↪multiplier of 2..) */
    dev->mCalData.H0_rH = H0_rh_x2 >> 1;
    dev->mCalData.H1_rH = H1_rh_x2 >> 1;

    /* T0 and T1 are 10 bits, the MSBs are stored together in the T1T0_MSB Register.␣
↪They have to be put together, and then divided by 8.. (see link to application note) */
    dev->mCalData.T0_degC = ((uint16_t) T0_degC_x8 | (((uint16_t)(T1T0_msb & 0x03)) <<␣
↪8)) >> 3;
    dev->mCalData.T1_degC = ((uint16_t) T1_degC_x8 | (((uint16_t)(T1T0_msb & 0x0C)) <<␣
↪6)) >> 3;
```

The driver is generated with a 'test' function. we will add the logic to test the devices connection:

hts221.c:97 *in the user-block-test section* :

```
    if( hts_read_reg(dev, &dev->mWhoAmI) == HTS_WHO_AM_I_DEFAULT)
    {
        return MRT_STATUS_OK;
    }
```

Finally add the code for the temperature and humidity functions:

hts221.c:108 *in the user-block-bottom section* :

```
int hts_read_humidity(hts221_t* dev)
{
    int16_t raw_adc;
    float tmp_f;

    //check to make sure data is ready, if not just use previous value
    if(! hts_check_flag(dev,&dev->mStatus, HTS_STATUS_HUM_READY))
    {
        return dev->mPrevHum;
    }

    //get raw adc value
    raw_adc = hts_read_reg(dev, &dev->mHumidityOut);

    //Use calibration coefs to interpolate data to RH%
    tmp_f = ((float)(raw_adc - dev->mCalData.H0_T0_OUT) * (float)(dev->mCalData.H1_rH -␣
↪dev->mCalData.H0_rH) / (float)(dev->mCalData.H1_T0_OUT - dev->mCalData.H0_T0_OUT))  + ␣
↪dev->mCalData.H0_rH;

    dev->mPrevHum = tmp_f * 100;
    return dev->mPrevHum;
}

int hts_read_temp(hts221_t* dev)
{
    int16_t raw_adc;
    float tmp_f;

    //check to make sure data is ready, if not just use previous value
    if(! hts_check_flag(dev,&dev->mStatus, HTS_STATUS_TEMP_READY))
    {
        return dev->mPrevTemp;
    }


    //get raw adc value
    raw_adc = hts_read_reg(dev, &dev->mTempOut);

    //Use calibration coefs to interpolate data to deg C
    tmp_f = ((float)(raw_adc - dev->mCalData.T0_out) * (float)(dev->mCalData.T1_degC -␣
↪dev->mCalData.T0_degC) / (float)(dev->mCalData.T1_out - dev->mCalData.T0_out))  +  dev-
↪>mCalData.T0_degC;
```

(continues on next page)

(continued from previous page)

```
    dev->mPrevTemp = tmp_f * 100;
    return dev->mPrevTemp;
}
```

Now lets try out our driver:

main.c:27 *USER CODE includes section*

```
#include "Devices/hts221/hts221.h"
```

main.c:95 *USER CODE 2 section*

```
  uint32_t ticks =0;
  int temperature;
  int humidity;
  hts221_t hts;                                 /* create instance of hts221 device*/
  hts_init_i2c(&hts, &hi2c2);                   /* Initialize it on I2C2 bus*/

  if(hts_test(&hts) == MRT_STATUS_OK)           /* Turn on LED if device passes test */
  {
      MRT_GPIO_WRITE(MRT_GPIO(LED_GRN),HIGH );
  }

  /* Set flags/fields for start up and 1hz data*/

  hts_set_flag(&hts, &hts.mCtrl1, HTS_CTRL1_PD);  /* set PD flag of CTRL 1 register, to␣
↪turn on device*/
  hts_set_ctrl1_odr(&hts, HTS_CTRL1_ODR_1HZ);     /* Set ODR field in CTRL1 Register to␣
↪1Hz*/

  /* OR we could use the configuration we created with: HTS_LOAD_CONFIG_AUTO_1HZ(&hts) */
```

main.c:122 *Replace entire while loop*:

```
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {


    /* Every 500 ms see if new data is ready, and read it */
    MRT_EVERY( 50, ticks)   /* convenience macro for systick timing*/
    {
      if(hts_check_flag(&hts, &hts.mStatus, ( HTS_STATUS_TEMP_READY | HTS_STATUS_HUM_
↪READY )  )) /*wait until both flags are set */
      {
        temperature = hts_read_temp(&hts);
        humidity = hts_read_humidity(&hts);
      }
    }
    ticks++;
    MRT_DELAY_MS(10);
```

(continues on next page)

(continued from previous page)

```
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
```

build the project and run it. The led should turn on to show it passed the device test. If you step through code you will see valid temperature/humidity readings.

2.5 Step 5: Create a PolyPacket Service

Now That we have a working device driver, lets create a messaging protocol so we can ask the device for data over the com port.

first we will need to add in some more MrT modules to support polypacket. back out to your projects root directory and open mrt-config again:

```
cd /path/to/project/root
mrt-config MrT
```

Select the following modules to import :

- Utilities/PolyPacket
- Utilities/JSON
- Utilities/COBS

Once they are imported, create your protocol template:

```
poly-make -t my_protocol
```

There should now be a file named `my_protocol.yml` in the root of your project. You can keep this wherever you want, but I find it handy to have it in the root of the project when debugging.

now modify the file to match the my_protocol.yml in the doc folder. For a detailed eplanation of the document reference PolyPacket.wiki/Defining-a-protocol

Once the descriptor is filled out, create a directory for your service, and then generate your service with an application layer:

```
mkdir MrT/Modules/my_service
poly-make -i my_protocol.yml -a -o MrT/Modules/my_service/
```

Add "-d doc " to create an ICD in the doc folder

This will generate 4 files:

- **my_protocolService.h** - header for service, you should never need to edit this
- **my_protocolService.c** - source for service, you should never need to edit this
- **app_my_protocol.h** - header for application layer
- **app_my_protocol.c** - source for application layer, this is where you will fill out packet handlers

First include the service:

main.c:28:

```
#include "my_service/app_my_protocol.h"
```

Next intialize the app layer

main.c:118:

```
  /* OR we could use the configuration we created with: HTS_LOAD_CONFIG_AUTO_1HZ(&hts) */

  app_my_protocol_init(&huart1); /* initialize the app layer and give it a uart␣
↪interface */

  /* For the UART RX, we are going to use some low level tricks for the stm32, because␣
↪their HAL layer is not great at receiving
   * unkown lengths of data. This will set us up with an interrupt everytime a new byte␣
↪comes in. its just cleaner and less hassle
   */
  UART_MASK_COMPUTATION(&huart1);
↪                                                                    /* Sets Uart1's␣
↪internal data mask based on STMCUBE configuration*/
  SET_BIT(huart1.Instance->CR1, USART_CR1_PEIE | USART_CR1_RXNEIE); /* Enable the␣
↪interrupts for STM32 UART receive */

  /* USER CODE END 2 */
```

Then add a call to process our service in the main loop

main.c:118:

```
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {


    /* Every 500 ms see if new data is ready, and read it */
    MRT_EVERY( 100, ticks)   /* convenience macro for systick timing*/
    {
      if(hts_check_flag(&hts, &hts.mStatus, ( HTS_STATUS_TEMP_READY | HTS_STATUS_HUM_
↪READY )  )) /*wait until both flags are set */
      {
        temperature = hts_read_temp(&hts);
        humidity = hts_read_humidity(&hts);
      }
    }
    app_my_protocol_process(); /* process our service*/
    ticks++;
    MRT_DELAY_MS(10);


    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
```

Now lets use the uart interrupt to feed our service

stm32l4xx_it.c:26

```
#include "my_service/app_my_protocol.h"
```

stm32l4xx_it.c:201

```
/**
  * @brief This function handles USART1 global interrupt.
  */
void USART1_IRQHandler(void)
{
  /* USER CODE BEGIN USART1_IRQn 0 */

    /* If RX-Not-Empty flag is set, then we have a byte of data */
    if(huart1.Instance->ISR & UART_FLAG_RXNE)
    {
      uint8_t data = (uint8_t)(huart1.Instance->RDR & (uint8_t)huart1.Mask); /* Mask Off␣
↪Data */
      mp_service_feed(0, &data ,1); /* feed the byte to our service */
    }

  /* USER CODE END USART1_IRQn 0 */
  HAL_UART_IRQHandler(&huart1);
  /* USER CODE BEGIN USART1_IRQn 1 */

  /* USER CODE END USART1_IRQn 1 */
}
```

Since we are using the interrupt we can disable the uart_read in our application layer:

app_my_protocol.c:66 - *comment out iface0_read()*

```
void app_my_protocol_process()
{
  /* read in new data from iface 0*/
  // iface0_read();

  /* process the actual service */
  mp_service_process();

}
```

Before we add anything else, lets test our service. Find the com port that the device is on in device manager. in my case it is COM3

> for WSL, COM ports are mapped to /dev/ttyS<Port Number>

open the poly-packet interpretter:

```
poly-packet -i my_protocol.yml
```

inside of poly-packet *connect over serial with a baud of 115200*

```
connect serial:/dev/ttyS3:115200
Ping
```

> note: every protocol is built with a ping and ack packet

You should see your 'Ping packet go out, and an ack returned'

```
______      _       ______           _         _
| ___ \    | |      | ___ \         | |       | |
| |_/ /__  | |_   _| |_/ /_ _  ___| | _____| |_
|  __/ _ \| | | | |  __/ _` |/ __| |/ / _ \ __|
| | | (_) | | |_| | | | (_| | (__|   <  __/ |_
\_|  \___/|_|\__, \_|  \__,_|\___|_|\_\___|\__|    [my_protocol protocol]
              __/ |
             |___/

 Port Opened : /dev/ttyS3

 --> { "packetType" : "Ping", "icd" : 3174876862}
 <-- { "packetType" : "Ack"}
```

tip: inside poly-packet, you can press tab to see available packets to send

2.6 Step 6: Customize the service

Now that we have our service working, lets make it do something useful. When we created the protocol description, we set up a poly_struct named 'Device'. We are going to use this to store and serve information about the device.

First we will make changes to our application layer, we will add our struct, and clean up some un-used sections while we are there

app_my_protocol.c:8

```
/************************************************************
        Application Layer
************************************************************/

#include "app_my_protocol.h"
mrt_uart_handle_t ifac0;

mp_struct_t myDevice; /* create device struct for storing/serving our data */

static inline HandlerStatus_e iface0_write(uint8_t* data, int len)
{
  /* Place code for writing bytes on interface 0 here */
  MRT_UART_TX(ifac0, data, len, 10);

  return PACKET_SENT;
}


/*****************************************************************************
  App Init/end
*****************************************************************************/
void app_my_protocol_init(mrt_uart_handle_t uart_handle)
{
  /* Set ifac0 to uart handle, this can use any peripheral, but uart is the most common␣
↪case */
```

(continues on next page)

(continued from previous page)

```
  ifac0 = uart_handle; //set interface to uart handle

  //initialize service
  mp_service_init(1,16);

  mp_struct_build(&myDevice, MP_STRUCT_DEVICE); /* builds the generic poly_struct into a␣
↪Device struct */
  mp_setDeviceName(&myDevice, "Jerry");         /* set the 'Name' field of the device␣
↪struct */

  mp_service_register_bytes_tx(0, iface0_write);

}
```

Next we can fill out our packet handlers.

The only packets we need to handle are: getdata, whoAreYou, and setName. the rest of the handlers can be deleted. (They are defined weakly in the service layer)

app_my_protocol.c:63

```
/*****************************************************************************
  Packet handlers
*****************************************************************************/
/**
  *@brief Handler for receiving getData packets
  *@param getData incoming getData packet
  *@param sensorData sensorData packet to respond with
  *@return handling mp_status
  */
HandlerStatus_e mp_GetData_handler(mp_packet_t* mp_getData, mp_packet_t* mp_sensorData)
{

  mp_packet_copy(mp_sensorData, &myDevice); /* copy fields from 'myDevice' into the␣
↪response packet*/

  return PACKET_HANDLED;  /* Make sure to change this to PACKET_HANDLED*/
}

/**
  *@brief Handler for receiving whoAreYou packets
  *@param whoAreYou incoming whoAreYou packet
  *@param myNameIs myNameIs packet to respond with
  *@return handling mp_status
  */
HandlerStatus_e mp_WhoAreYou_handler(mp_packet_t* mp_whoAreYou, mp_packet_t* mp_myNameIs)
{

  mp_packet_copy(mp_myNameIs, &myDevice);  /* copy fields from 'myDevice' into the␣
↪response packet*/

  return PACKET_HANDLED;   /* Make sure to change this to PACKET_HANDLED*/
}
```

(continues on next page)

(continued from previous page)

```
/**
  *@brief Handler for receiving setName packets
  *@param setName incoming setName packet
  *@return handling mp_status
  */
HandlerStatus_e mp_SetName_handler(mp_packet_t* mp_setName)
{

  mp_packet_copy(&myDevice, mp_setName); /* Copy fields from incoming packet to 'myDevice
↪' */

  return PACKET_HANDLED; /* Make sure to change this to PACKET_HANDLED*/
}
```

** Now we will use the sensor data from our device driver to set the fields of 'myDevice'

make it available to our main.c

app_my_protocol.h:11

```
extern mp_struct_t myDevice;
```

Then add code to set the values in our main loop:

main.c:129

```
 /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {


    /* Every 500 ms see if new data is ready, and read it */
    MRT_EVERY( 100, ticks)   /* convenience macro for systick timing*/
    {
      if(hts_check_flag(&hts, &hts.mStatus, ( HTS_STATUS_TEMP_READY | HTS_STATUS_HUM_
↪READY )  )) /*wait until both flags are set */
      {
        temperature = hts_read_temp(&hts);
        humidity = hts_read_humidity(&hts);

        mp_setTemp(&myDevice, temperature);
        mp_setHumidity(&myDevice, humidity);

      }
    }
    app_my_protocol_process(); /* process our service*/
    ticks++;
    MRT_DELAY_MS(10);
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
```

build!

2.7 Step 7: Interact with poly-packet

Now that our service is complete, we can interact with it using the poly-packet cli

```
poly-packet -i my_protocol.yml -c connect serial:/dev/ttyS3:115200
```

-c lets you pass a command on start-up, I use it as a convenient way to connect

Once you are in the CLI, you can send some packets

```
whoAreYou
getData
setName deviceName: Jason Berger
whoAreYou
```

produced the following output:

```
______      _       ______           _         _
| ___ \    | |      | ___ \         | |       | |
| |_/ /__  | |_   _| |_/ /_ _  ___| | _____| |_
|  __/ _ \| | | | |  __/ _` |/ __| |/ / _ \ __|
| | | (_) | | |_| | | | (_| | (__|   <  __/ |_
\_|  \___/|_|\__, \_|  \__,_|\___|_|\_\___|\__|    [my_protocol protocol]
              __/ |
             |___/

 Port Opened : /dev/ttyS3

 --> { "packetType" : "whoAreYou"}
 <-- { "packetType" : "myNameIs", "deviceName" : "Jerry"}

 --> { "packetType" : "getData"}
 <-- { "packetType" : "sensorData", "temp" : 2865, "humidity" : 4939}

 --> { "packetType" : "setName", "deviceName" : "Jason Berger"}
 <-- { "packetType" : "Ack"}

 --> { "packetType" : "whoAreYou"}
 <-- { "packetType" : "myNameIs", "deviceName" : "Jason Berger"}
```

CHAPTER

THREE

MRTUTILS

`mrtutils` is a collection of tools for working with the MrT framework. It includes tools for managing modules, creating device drivers, and implementing custom BLE profiles on supported platforms

```
pip install mrtutils
```

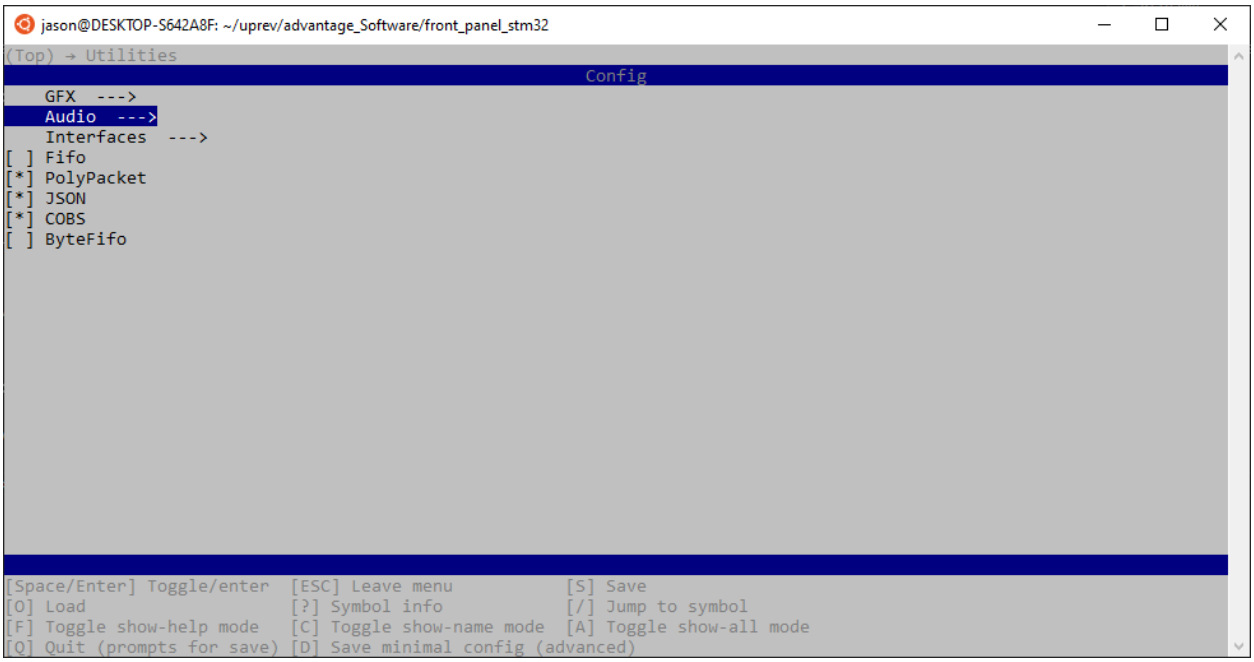
3.1 mrt-config

mrt-config is the tool used to manage the MrT Modules in your project.

```
cd <path/to/project>
mrt-config <relative/path/for/MrT/root>
```

Note: If no path is provided, it will default to ./MrT and create the directory if it does not exist

This will open the `mrt-config` tool which allows you to select which modules you would like to integrate into your project. The UI is based on *menuconfig* to be as flexible as possible in terms of where you can run it, ie in containers or remote development environments over ssh.



Note: MrT Modules are added as git sub-modules, if you are in a directory that does not contain a git repo, it will initialize one.

3.2 mrt-device

The mrt-device tool allows user to create driver code from device description files. This provides very consistent drivers and also creates an easily parseable device file as a byproduct. This can be used for better documentation as well as a basis for automated testing of hardware.

Note: The code generated from this tool requires the Mrt RegDev module

3.2.1 Step 1: Define device:

Devices are defined with a YAML file.

To generate a blank template to start from:

```
mrt-device -t /path/to/file.yml
```

example from hts221 driver

```
---
name: HTS221
description: Humidity and Temperature Sensor
category: Device
requires: [RegDevice,Platform]
datasheet: https://www.st.com/content/ccc/resource/technical/document/datasheet/4d/9a/9c/
↪ad/25/07/42/34/DM00116291.pdf/files/DM00116291.pdf/jcr:content/translations/en.
↪DM00116291.pdf
mfr: STMicroelectronics
mfr_pn: HTS221TR
digikey_pn: 497-15382-1-ND

prefix: HTS
bus: I2C
i2c_addr: 0xBE


##############################################################################################
↪##################
#                                    Registers                                               ␣
↪                  #
##############################################################################################
↪##################

registers:
- WHO_AM_I:      { addr: 0x0F , type: uint8_t, perm: R, desc:  Id Register, default: 0xBC}
- AV_CONF:       { addr: 0x10 , type: uint8_t, perm: RW, desc: Humidity and temperature␣
```

(continues on next page)

(continued from previous page)

```
↪resolution mode}
- CTRL1:        { addr: 0x20 , type: uint8_t, perm: RW, desc: Control register 1}
- CTRL2:        { addr: 0x21 , type: uint8_t, perm: RW, desc: Control register 2}
- CTRL3:        { addr: 0x22 , type: uint8_t, perm: RW, desc: Control register 3}
- STATUS:       { addr: 0x27 , type: uint8_t, perm: R, desc: Status register}
- HUMIDITY_OUT: { addr: 0x28 , type: int16_t, perm: R, desc: Relative humidity data }
- TEMP_OUT:     { addr: 0x2A , type: int16_t, perm: R, desc: Temperature data}

- H0_rH_x2:     { addr: 0x30 , type: uint8_t, perm: R, desc: Calibration data}
- H1_rH_x2:     { addr: 0x31 , type: uint8_t, perm: R, desc: Calibration data}
- T0_DEGC_x8:   { addr: 0x32 , type: uint8_t, perm: R, desc: Calibration data}
- T1_DEGC_x8:   { addr: 0x33 , type: uint8_t, perm: R, desc: Calibration data}
- T1T0_MSB:     { addr: 0x35 , type: uint8_t, perm: R, desc: Calibration data}
- H0_T0_OUT:    { addr: 0x36 , type: int16_t, perm: R, desc: Calibration data}
- H1_T0_OUT:    { addr: 0x3A , type: int16_t, perm: R, desc: Calibration data}
- T0_OUT:       { addr: 0x3C , type: int16_t, perm: R, desc: Calibration data}
- T1_OUT:       { addr: 0x3E , type: int16_t, perm: R, desc: Calibration data}

################################################################################################
↪##################
#                                       FIELDS                                                 ␣
↪                    #
################################################################################################
↪##################
fields:
- STATUS:
    - TEMP_READY: { mask: 0x01, desc: indicates that a temperature reading is ready }
    - HUM_READY: { mask: 0x02, desc: indicates that a humidity reading is ready }

- CTRL1:
    - PD: {mask: 0x80, desc: power down mode}
    - BDU: {mask: 0x04, desc: Block Data update. Prevents update until LSB of data is␣
↪read}
    - ODR:
        mask: 0x03
        desc: Selects the Output rate for the sensor data
        vals:
        - ONESHOT: { val: 0, desc: readings must be requested}
        - 1HZ: { val: 1, desc: 1 hz sampling}
        - 7HZ: { val: 2, desc: 7 hz sampling}
        - 12_5HZ: { val: 3, desc: 12.5 hz sampling}

- CTRL2:
    - BOOT: {mask: 0x80, desc: Reboot memory content}
    - HEATER: {mask: 0x02, desc: Enable intenal heating element}
    - ONESHOT: {mask: 0x01, desc: Start conversion for new data}

- TEMP_OUT:
    - TEMP_OUT: {mask: 0xFFFF, desc: Current ADC reading for temperature sensor}

- HUMIDITY_OUT:
    - HUM_OUT: {mask: 0xFFFF, desc: Current ADC reading for humidity sensor}
```

(continues on next page)

(continued from previous page)

```
##########################################################################################
↪##################
#                                    Preset Configs                                      ␣
↪                  #
##########################################################################################
↪##################
configs:
- auto_1hz:
    desc: Sets device to update every second
    registers:
        - CTRL2: {BOOT: 1, delay: 20} #20 ms delay after register write
        - CTRL1: { ODR: 1HZ, BDU: 1}
```

The descriptor file contains device information such as part numbers, links to datashees, and other relevant information. It also contains definitions of registers and data structures on the device. The main sections are *Header Properties* , *Registers* , and *Fields*

Header Properties

The header of the descriptor file contains several Properties. `name` and `description` are required, but others should also be included if they apply

name
: Name of device

description
: Description of device

datasheet
: url to public datasheet

mfr
: Name of manufacturer

mfr_pn
: Manufacturer part number

digikey_pn
: Digikey part number

prefix
: prefix to append to struct and function names to prevent conflicts in projects

bus
: bus type for driver, can be `I2C`, `SPI`, `UART`, or any combination of those (comma separated)

i2c_addr
: I2C address for device. For devices with configurable address, set this to the base address. It can be changed in the driver

Registers

`registers` are individualy addressable memory registers on the device. each register can have the folowing attributes:

- **addr**: register address on device
- **type**: register type, (default is uin8_t)
- **perm**: premissions on register R for read, W for write
- **desc**: description of register. used for code documentation
- **default**: default value of the register

Fields

`fields` are data fields contained in registers. They are grouped by register and they contain the following attributes:

- **mask** : this specifies the mask for the field. This is used to mask and shift data to match the field.
- **vals** : this is a list of possible values and their descriptions for the field.

Note: If a field is defined with a single bit mask, and no values, it is interpretted as a 'flag'. Flag fields have macros generated for setting, clearing, and checking them.

Configs

`Configs` allow the user to define preset configs for common use cases. This will create a macro for setting up the registers

```
/**
* @brief Sets device to update every second
* @param dev ptr to HTS221 device
*/
#define HTS_LOAD_CONFIG_AUTO_1HZ(dev) \
hts_write_reg( (dev), &(dev)->mCtrl2, 0x80);     /* BOOT: 1 */                    \
MRT_DELAY_MS(20);                                /* Delay for CTRL2 */ \
hts_write_reg( (dev), &(dev)->mCtrl1, 0x05);     /* ODR: 1HZ , BDU: 1 */          \
```

3.2.2 Step 2: generate the code

To generate the code, use mrt-device and specify an input and an output path:

```
mrt-device -i device.yaml -o .
```

The tool will generate 3 files (using hts221 as an example):

- **hts221.h :** header file for driver
- **hts221.c :** Source file for driver
- **hts221_dev.h :** Macros generated from device file. this contains macros for addresses, values, masks, and functions for accessing fields/flags in registers.

3.2.3 Step 3: customize

This will provide a good base with access to all of the register. To add more functionality you can add to the code. If you want to ability to modify the device file further, keep your code inside of the 'user code' blocks provided:

```
/*user-block-init-start*/
/*user-block-init-end*/
```

If the device does not follow the normal register access schemes, you can specify your own, and redirect the mrt_regdev_t fRead and fWrite function pointers to them.

```
/**
  *@brief writes buffer to address of device
  *@param dev ptr to generic register device
  *@param addr address in memory to write
  *@param data ptr to data to be written
  *param len length of data to write
  *@return status (type defined by platform)
  */
mrt_status_t my_write_function(mrt_regdev_t* dev, uint32_t addr, uint8_t* data,int len )
{
    //Do Something
}

static mrt_status_t hts_init(hts221_t* dev)
{
    /*user-block-init-start*/
    dev->mRegDev.fWrite = my_write_function;
    /*user-block-init-end*/
    return MRT_STATUS_OK;
}
```

3.3 mrt-ble

`mrt-ble` is a tool for creating gatt profile to use on BLE projects. It uses a yaml descriptor file to create C code and documentation for the Gatt profile. The generated Documentation includes a `Live ICD` which is a single page web app that can connect to the `ble` device and interact with the GATT Server.

`mrt-ble` is a tool in `mrtutils`, so if that is not already installed, install it first:

```
pip install mrtutils
```

To get started, you can create a template:

```
mrt-ble -t my_profile
```

this will create a decriptor file my_profile.yml with an example profile filled out

3.3.1 Step 1: Define the profile

The Generated example descriptor file has comments to explaind the various fields. The overall structure is that each descriptor file creates a `Profile`. A `Profile` is a group of `Services`, and a `Service` is a group of related `Characteristics`

Header Properties

The beggining of the document contains properties for the profile .

name
: Name of Profile

description
: Description of Profile

prefix
: short prefix to append to profile structs and functions to avoid conflicts in code

Services

Services can be custom, or imported from Bluetooth SIG standards using a URI. When importing from a SIG standard, all `Mandatory` Characteristics are automatically added, but optional ones must be specified. See the `Device Service` and `Battery Service` in the *Example file* for an example of this.

Every service must have a prefix. And all custom services must have a `UUID`.

Optional properties:

icon
: named icon from FontAwesome

Characteristics

Characteristics are individual fields in a `Service`. In a SIG standard `Service` you can use the SIG standard `Characteristics` by specifying a URI.

Custom Characteristics must have a `type`. this can be any of the following

Type	Description
uint8	**Basic Unsigned Integer** Types
uint16	
uint32	
uint64	
uint	
char	
int8	**Basic Signed Integer** Types
int16	
int32	
int64	
int	
float	decimal types
double	
string	array of chars
Enum	uint8 with named values. Each value gets a symbol in code
flags	Bitmask with a defined symbol in code for each bit. (maximum of 32 bits in a Characteristic)
mask	
Array	specified with <type>*<size> ex: uint16*32 is an array of 32 uint16 values

Other properties in a `Characteristic` include:

Example File

```
---
name: sample
author: Jason Berger
created: 02/20/2020
desc: GATT profile example
prefix: tp


services: #list multiple services in file to create full profile

############################################################
#                     Device Service                       #
#                                                          #
#  Shows example of using Bluetoot SIG define Services     #
############################################################
- Device:
    uri: org.bluetooth.service.device_information #User URI of bluetooth sig standard␣
↪service. For a list of all standard services visit https://www.bluetooth.com/
↪specifications/gatt/services
```

(continues on next page)

(continued from previous page)

```
    prefix: dvc
    chars:  #list out uris of 'optional' desired chars for bluetooth SIG services
        - {uri: org.bluetooth.characteristic.manufacturer_name_string , default: Up-Rev}
↪#Set a default value
        - {uri: org.bluetooth.characteristic.serial_number_string}
        - {uri: org.bluetooth.characteristic.hardware_revision_string}
        - {uri: org.bluetooth.characteristic.firmware_revision_string, desc: Firmware␣
↪revision} #You can override defaults from Bluetooth SIG (name,desc, perm, etc..)

################################################################
#                      Battery Service                         #
#                                                              #
#  Shows example of inline declaration for standard serivce    #
################################################################
- Battery: {uri: org.bluetooth.service.battery_service}
    #if a prefix isnt specified it will create one using the first 3 characters of the␣
↪name.
    #no need to list chars, because there is only one for the battery service and it is␣
↪mandatory per the SIG spec

################################################################
#                        Sprinkler Servive                     #
#                                                              #
# Show example of creating a custom service to control an      #
# Automated sprinler system                                    #
#                                                              #
#  - Controls 6 valves and pump for sprinklers                 #
#  - Temperature sensor                                        #
#  - 6 soil moisture sensors                                   #
################################################################
- Sprinkler:
    prefix: spr
    desc: Custom service for a sprinkler system
    uuid: 71a8-1b49-ce39-0088-6b62-c8ed-9e20-9a5b
    icon: fa-faucet # This adds an icon to the Live ICD for the service using Font-
↪Awesome. Visit their site to view options: https://fontawesome.com/icons?d=gallery&
↪m=free
    chars:

        - Thresh: { type: uint16, perm: RW , desc: Moisture Threshold to turn on the␣
↪sprinklers} #if char uuid is blank, it will increment from previous char, or service␣
↪uuid if it is the first in the service

        - Temperature: { type: uint16, perm: RN , desc: Temperature reading from sensor,␣
↪unit: °f, coef: 0.01} #unit and coef have no affect on data, just how ther are␣
↪displayed in the live ICD

        - Moisture: {type: uint16*6, desc: Moisture readings from all 6 zones, unit: "%"␣
↪} # Create an array of 6 uint16_t values.

        - Relays:
            type: flags #flags create an array of bits which are individualy controlled
```

(continues on next page)

(continued from previous page)

```
            perm: RWN   #Read Write and Notify permissions
            desc: Controls Relays for pump and valves
            vals:
            - pump: {desc: pump control}
            - valve01: valve 1 control #For convenience values can be written in this␣
↪shorthand. same as '- valve01: {desc: valve 1 control}'
            - valve02: valve 2 control
            - valve03: valve 3 control
            - valve04: valve 4 control
            - valve05: valve 5 control
            - valve06: valve 6 control

        - SoilType:
            type: enum  #enums are treated as an unsigned int, but they have symbols␣
↪defined and a switch case generated in the write handler
            perm: RW
            desc: Soil type for the yard
            vals:
            - Peat: Peat soil
            - Sand: Peat soil
            - Clay: Peat soil
            - TopSoil


################################################################
#                      Firmware OTA Service                    #
################################################################
- FOTA:
    desc: sercive for performing over the air updates
    uuid: 71a8-1b49-ce39-0088-6b62-c8ed-9A10-9a5b
    prefix: ota
    chars:
        - version:    { type: string,   perm: RW, desc: current Firmware version}  #␣
↪uuid: 0x9A11
        - newVerion:  {type: string,   perm: RW, desc: version of new firmware being␣
↪loaded}
        - data:       {type: uint8*64,  perm: RW, desc: current block of data}
        - seq:        {type: uint32,    perm: RW, desc: sequence number of current block␣
↪ }
        - crc:        {type: uint32,    perm: RW, desc: crc of new firmware  }
        - status:
            type: enum
            perm: RW
            desc: status of OTA process
            vals:
            - IDLE: { desc: no ota operation taking place}
            - DOWNLOAD: { desc: Currently downloading new firmware}
            - COMPLETE: { desc: Firmware download complete. ready to update}
```

3.3.2 Step 2: Generate Code

Once you have the profile defined, you can generate the code with

```
mrt-ble -i <yaml file> -o <output/path> -d <doc/path>
```

Note: regenerating the source code will **not** overwrite any code in the handler functions for the profile or services.

This will generate the following structure with source/header files:

outputDir
├── svc
│ ├── dev_svc.h
│ ├── dev_svc.c
│ ├── ss_svc.h
│ ├── ss_svc.c
│ ├── bat_svc.h
│ ├── bat_svc.c
│ ├── ota_svc.h
│ └── ota_svc.c
├── app_dev_svc.c
├── app_ss_svc.c
├── app_bat_svc.c
├── app_ota_svc.c
└── sample_profile.c/h

3.3.3 Step 3: Integrating Code

The files in the `svc` folder are the low level descriptors and weakly defined handler functions. In most cases, there is no need to modify these files.

The `app_xx_svc.c` files are for application level logic and contain the actual handler functions. This is where you will put in your logic for handling events for each characteristic.

Each service will have an event handler for each `Characteristic` and a `post_init` handler. The `post_init` handler is called after the GATT server is initialized. This is where default values will be set.

The `Characteristic` event handlers handle all events for a given `Characteristic`. The `mrt_gatt_evt_t` struct contains the type of event [READ, WRITE,NOTIFY], as well as the raw data, and data size for the event.

example handlers from app_dev_svc.c:

```
/* Post Init ------------------------------------------------------------------*/

/**
* @brief Called after GATT Server is intialized
*/
void dev_svc_post_init_handler(void)
{
    dvc_set_manufacturer_name("Up-Rev");
```

(continues on next page)

(continued from previous page)

```
    dvc_set_firmware_revision("0.1.9");
    dvc_set_serial_number("001");
}

/* Characteristic Event Handlers-------------------------------------------------*/

/**
* @brief Handles GATT event on Manufacturer_Name Characteristic
* @param event - ptr to mrt_gatt_evt_t event with data and event type
*/
mrt_status_t dev_manufacturer_name_handler(mrt_gatt_evt_t* event)
{
    if(event->mType == GATT_EVT_VALUE_WRITE)
    {
        char* val = ((char*) event->mData.data); /* Cast to correct data type*/
        MRT_PRINTF("Device name set to %s", val);
    }

    return MRT_STATUS_OK;
}
```

Note: For more information on the mrt_gatt_evt_t struct, read the docs for the gatt-server module

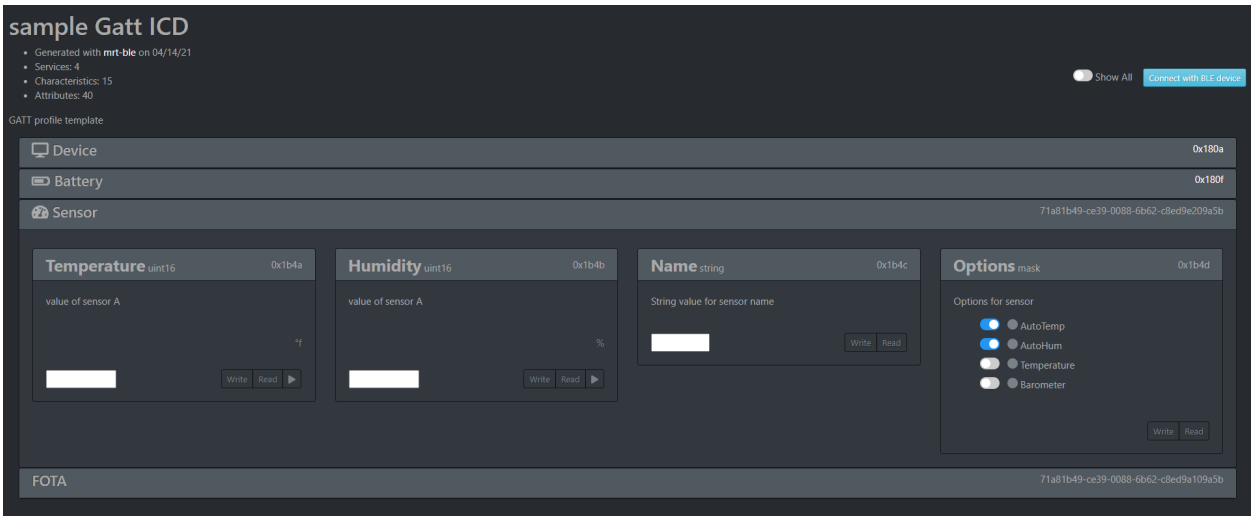
The source code and header for `sample_profile.c` contain the initialization funtion which will initialize all of the services. This function is called by the platform once the GATT server is up. This will vary from platform to platform so check the `Platform` documentation for how to implement this. But the most common method is to register the init function, before starting any bluetooth services.

```
MRT_GATT_REGISTER_PROFILE_INIT(sample_profile_init);
```

Once the function is registered, it is up to the Platform layer to call the function at the appropriate time.

3.3.4 Live ICD

Once your GATT profile is running on the target device, it is useful to be able to interact with it for testing and development. When the code is generated with documentation it produces 2 files. The first is a plain text ICD for documentation, and the second is a `Live ICD`. This is a single page web app which can connect to the device over BLE and provide a GUI for interacting with the device.



3.4 mrt-version

`mrt-version` is a tool for managing the version information in a project. It keeps the version information in a header file, and provides a convenient way to update it and use the version with continuous integration tools.

3.4.1 Creating the header file

```
mrt-version main/version.h
```

This will create the header file, with the initial version set to 0.0.0.0

Note: 'yml', 'env', 'json', and 'h' files are supported

```
/**
 * @file version.h
 * @author generated by mrt-version (https://mrt.readthedocs.io/en/latest/pages/mrtutils/
↪mrt-version.html)
 * @brief version header
 * @date 05/01/21
 */

#define VERSION_MAJOR      0
#define VERSION_MINOR      0
#define VERSION_PATCH      0
#define VERSION_BUILD      0
#define VERSION_BRANCH     "master"
#define VERSION_COMMIT     "c4526b4ec43b9a74c572bfbb6059b65bce4b0029"

#define VERSION_STRING "0.0.0.0"
```

Note: To include repo information, call the tool from the root of the projects repo. when the branch is not 'master' an '*' will be added to the end of VERSION_STRING. This makes it clear to the user/tester that they are not using an

official build

3.4.2 Supported File Types

`mrt-version` can be used with several file types for different types of projects. The file type is automatically detected from the extension of the filename.

```
mrt-version version.env  # environment variable file
mrt-version version.h    # C header file
mrt-version version.json # JSON file
mrt-version version.yml  # YAML file
```

3.4.3 Updating the Version

After the initial file is created, you can set specific parts with the command line arguments (–major,–minor,–patch, –build). These values can be set to a value or incremented by a value. `Minor` and `Patch` can also be set to `auto`. `auto` will count the number of commits since the parent portion was last updated. i.e. If `Patch` is set to `auto` it will count the number of commits on the master branch since `Minor` was last updated, and use that count as the new value for `Patch`

```
mrt-version main/version.h --patch +1 --build 44
```

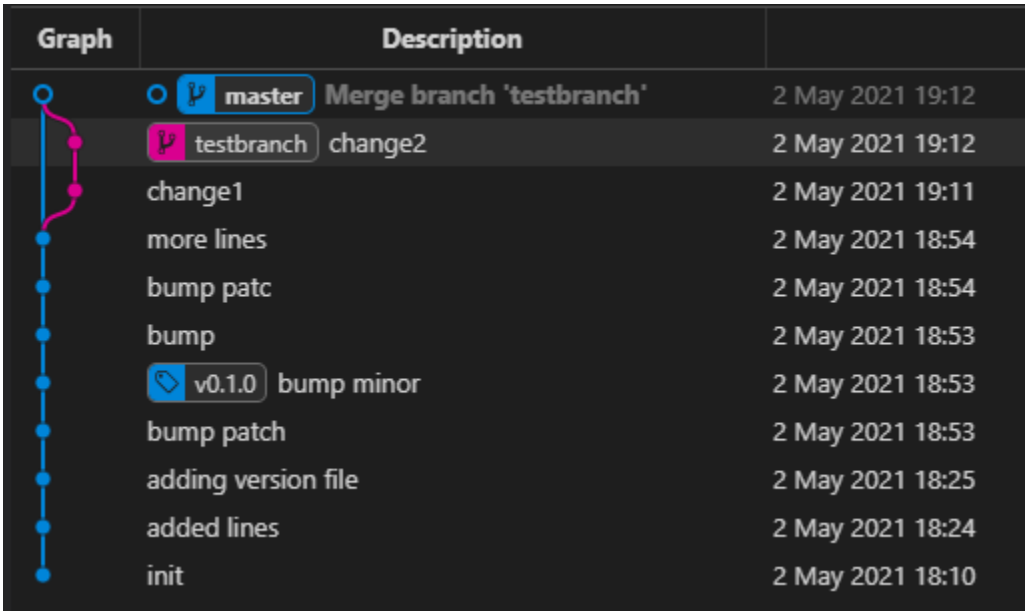
```
#define VERSION_MAJOR      0
#define VERSION_MINOR      0
#define VERSION_PATCH      1
#define VERSION_BUILD      44
#define VERSION_BRANCH     "master"
#define VERSION_COMMIT     "c4526b4ec43b9a74c572bfbb6059b65bce4b0029"

#define VERSION_STRING "0.0.1.44"
```

Note: Incrementing `Minor` will reset `Patch` to 0, and incrementing `Major` will reset `Minor` and `Patch` to 0.

Auto

`Minor` and `Patch` can also be set to `auto`. `auto` will count the number of commits since the parent portion was last updated. i.e. If `Patch` is set to `auto` it will count the number of commits on the master branch since `Minor` was last updated, and use that count as the new value for `Patch`

example:




```
mrt-version inc/version.h --patch auto
```

This would change the version to v0.1.4 since there have been 4 commits to the master branch since the `Minor` was incremented at the v0.1.0 tag

3.4.4 Build System/Webhook integration

The tool will always output the version string so it can be easily used for other things such as git tags and documentation.

In this example `patch` is incremented by 1, and then the commit is tagged in the repo with the output (i.e. 'v2.1.3')

```
VERSION_STR=$(mrt-version main/version.h --patch +1 )
git tag -a $VERSION_STR -m "Adding Version Tag"
```

By default the output format is *Majon.Minor.Patch*, but it can be customized with the *–format* flag. It uses simple string substition and the available variables are *$MAJOR*, *$MINOR*, *$PATCH*, *$BUILD* , *$BRANCH*, and *$HASH*.

3.4.5 Future Improvements

The next step will be to have this tool generate and update changelog as the version is updated.

3.5 mrt-doc

`mrt-doc` is a tool used for documentation in projects. It gathers all of the mrt.yml files from the modules and creates a master `mrt.yml` in the root MrT directory. It can also be used to combine all of the documentationusing the `-d` flag.

```
mrt-doc -d doc/moddocs
```

This will create the directory `doc/moddocs` and populate it with a folder structure that matches the structure of the modules, along with any README files. Each directory in the structure will contain an `index.rst` containing a `toctree` for that folder.

Note: Currently supported file types are `reStructuredText`_ and `Markdown`_

Example project contain some MrT modules:

```
doc
└── moddocs
    ├── Devices
    │   ├── RegDevice
    │   │   └── README.rst
    │   ├── Sensors
    │   │   ├── HTS221
    │   │   │   └── README.rst
    │   │   └── index.rst
    │   └── index.rst
    ├── Platforms
    │   ├── Atmel
    │   │   └── README.md
    │   ├── Common
    │   │   └── README.md
    │   └── index.rst
    ├── Utilities
    │   ├── COBS
    │   │   └── README.md
    │   ├── JSON
    │   │   └── README.md
    │   ├── PolyPacket
    │   │   ├── doc
    │   │   │   └── logo.png
    │   │   └── README.rst
    │   └── index.rst
    └── index.rst
```

Note: If you would like to include additional files (documents, pictures, etc) in a submodules documentation, add them to a `doc` folder in the submodule. This folder will also be copied into the structure.

3.6 mrt-gen

3.6.1 Code Templates

`mrt-gen` is a tool used to create common project components. By default it creates plain .h/.c files

```
mrt-gen src/test
```

This creates `src/test.h` and `src/test.c`. Adding '-t cpp' will create `src/test.h` and `src/test.cpp`

3.6.2 Creating Sphinx documentation

If a `--type` is not specified, and the path ends with `docs`, it will create a docs folder with a sphinx template.

```
mrt-gen ./docs
```

```
docs
├── Makefile
├── assets
│   └── diagrams
│       └── samplediagram.dio.png
├── conf.py
├── index.rst
└── pages
    └── samplepage.rst
```

This can be used to generate an html or pdf version of the documentation

```
cd docs
make latexpdf
make html
```

3.6.3 MrT Module Template

using the `-m` flag will create an MrT submodule with the required items.

```
mrt-gen -m mymodule
```

```
└── mymodule
    ├── README.rst
    ├── mrt.yml
    ├── mymodule.c
    ├── mymodule.h
    └── mymodule_UT.cpp
```

3.7 Tools

mrt-config
: Manages MrT modules in a project.

mrt-device
: Generates device drivers for register based devices

mrt-ble
: Creates custom Bluetooth Low Energy GATT profiles along with C code, documentation, and a single page web client for the GATT server using Web Bluetooth API.

mrt-version
: Used to manage version information of a project

mrt-doc
: Generates project documentation

mrt-gen
: Generates MrT module template

CHAPTER

FOUR

POLYPACKET

Poly Packet is a set of tools aimed at generating serial communication protocols from embedded projects. Protocols are described in an YAML document which can be easily shared with all components of a system.

A python script is used to parse the YAML file and generate C/C++ code as well as documentation. The code generation tool can create the back end service, application layer, and even an entire linux utility app

4.1 Installation

while `PolyPacket` is its own package separate from `mrtutils`, it is automatically installed when `mrtutils` is installed. But if you want to install it separately you can:

```
pip install polypacket
```

4.2 Step 1: Defining a Protocol

Protocols are defined with a YAML file. To get started you can generate a sample template:

```
poly-make -t my_protocol
```

This will generate `my_protocol.yml`

4.2.1 Descriptor File

Protocols are generated using YAML. The messaging structure is made up 4 entity types:

- Fields
- Packets
- Vals
- Structs

4.2.2 Fields

A field is a data object within a packet. These can be expressed either as nested yaml, or an inline dictionary

Example fields:

```
fields:
- sensorA: { type: int16 ,desc: Value of Sensor A}
- sensorB:
    type: int
    format: hex
    desc: Value of Sensor B

- sensorsC_Z:
    type: int*24
    desc: Values for remaining 24 sensors
```

type
: The data type for the field. *n indicates it is an array with a max size of n

format
: (optional) This sets the display format used for the toString and toJsonString methods [hex , dec , assci]

desc
: (optional) The description of the field. This is used to create the documentation

Supported types:

Type	Description
uint8	**Basic Unsigned Integer** Types
uint16	
uint32	
uint64	
uint	
char	
int8	**Basic Signed Integer** Types
int16	
int32	
int64	
int	
float	decimal types
double	
string	array of chars
Enum	uint8 with named values. Each value gets a symbol in code
flags	Bitmask with a defined symbol in code for each bit. (maximum of 32 bits in a Characteristic)
mask	
Array	specified with <type>*<size> ex: uint16*32 is an array of 64 uint16 values

Fields can be nested into 'Field Groups' for convenience

```
fields:
- header:
    - src: {type: uint16, desc: Address of node sending message }
    - dst: {type: uint16, desc: Address of node to receive message }
```

Note: these will be added to the packet as regular fields. The grouping is just for convenience

4.2.3 Packets

A Packet describes an entire message and is made up of fields

example Packet:

```
packets:
- Data:
    desc: contains data from a sensor
    fields:
        - header
        - sensorA
        - sensorB
        - sensorName
```

name
The name of the packet

desc
(optional) description of the packet for documentation

response
(optional) name of the packet type expected in response to this message (if any)

within the packet we reference Fields which have already been declared in the Fields section. these references contain 3 attributes:

name
The name of the field

req
(optional) makes the field a requirement for this packet type

desc
(optional) description of this field for this packet type, will override fields description in the documentation for this packet type only

4.2.4 Val

Val entities are used for defining options in `enum` and `flags` fields.

```
fields:
- cmd:
    type: enum
    format: hex
    desc: command byte for controlling node
    vals:
        - led_ON: { desc: turns on led}
        - led_OFF: { desc: turns off led}
        - reset: { desc: resets device }
```

In this example an enum is used to set up some predefined options for the **cmd** field. enums are created with sequential values starting at 0. a **flags** field is defined in the same way, but instead of sequential numbers, it shifts bits to the left, to create a group of individually set-able flags.

4.2.5 Struct

Structs are meant to store a model of an object locally. at the low level structs are essentially the same thing as packets in that they are a collection of fields. The only real difference is the name, and how they are documented.

>The purpose of structs is they make it easy to manage remote object(s). poly_packet_copy(dst,src) copies all mutual fields from src to dst, so using a single line in the handlers for the get/set packets gives us a remotely configurable node

```
structs:

    - Node:
        desc: struct for modeling node
        field:
            - sensorA
            - sensorB
            - sensorName
```

Example of Struct usage:

```
sp_struct_t thisNode; //must be initialized with sp_struct_build(&thisNode, SP_STRUCT_
↪NODE);

HandlerStatus_e sp_Data_handler(sp_packet_t* sp_data)
{

sp_packet_copy(&thisNode, sp_data); //update thisNode from incoming data packet

return PACKET_HANDLED;
}

HandlerStatus_e sp_GetData_handler(sp_packet_t* sp_getData, sp_packet_t* sp_data)
{

sp_packet_copy( sp_data, &thisNode);  //update data packet with fields from thisNode

return PACKET_HANDLED;
}
```

4.2.6 Example Protocol

Here is an example file. This is the starting point when you generate a template:

```
---
name: sample
prefix: sp  #this defines the prefix used for functions and types in the code. This␣
↪allows multiple protocols to be used in a project
desc: This is a sample protocol made up to demonstrate features of the PolyPacket
code generation tool. The idea is to have a tool that can automatically create parseable/
↪serializable
messaging for embedded systems

##############################################################################################
↪##################
#                                     FIELDS                                                 ␣
↪                  #
##############################################################################################
↪##################

fields:

#Fields can be nested into a 'Field Group' for convenience. They will be put in the␣
↪packet just like regular fields
- header:
    - src: {type: uint16, desc: Address of node sending message }
    - dst: {type: uint16, desc: Address of node to receive message }

- sensorA: { type: int16 ,desc: Value of Sensor A}  #Simple Fields can be defined as␣
↪inline dictionares to save space

- sensorB:
```

(continues on next page)

(continued from previous page)

```
    type: int
    desc: Value of Sensor B

- sensorName:
    type: string
    desc: Name of sensor

- cmd:
    type: enum
    format: hex
    desc: command byte for controlling node
    vals:
        - led_ON: { desc: turns on led}
        - led_OFF: { desc: turns off led}
        - reset: { desc: resets device }

##############################################################################################
↪##################
#                                    Packets                                                 ␣
↪                  #
##############################################################################################
↪##################
packets:
- SendCmd:
    desc: Message to send command to node
    fields:
        - header
        - cmd


- GetData:
    desc: Message tp get data from node
    response: Data          #A response packet can be specified
    fields:
        - header

- Data:
    desc: contains data from a sensor
    fields:
        - header
        - sensorA
        - sensorB
        - sensorName : {desc: Name of sensor sending data }   #Field descriptions can be␣
↪overriden for different packets
##############################################################################################
↪##################
#                                    Structs                                                 ␣
↪                  #
##############################################################################################
↪##################

structs:
```

(continues on next page)

(continued from previous page)

```
- Node:
    desc: struct for modeling node
    fields:
        - sensorA
        - sensorB
        - sensorName
```

4.2.7 Agents

Agents allow the CLI to be extended to simulate behavior and use custom commands. They do not affect the way code is generated, they are only used when running the CLI tool.

- Display custom/calculated information based on packet data
- route packets to other interfaces
- simulate values or responses for testing
- create full a test utility which verifies data in the packets

```
##############################################################################################
↪##################
#                                    Agents                                                  ␣
↪                  #
##############################################################################################
↪##################
agents:
    # This creates an agent named 'node' to load it, add '-s node' when running poly packet
    # naming an agent 'default' will cause it to load automatically when the CLI is␣
↪started
    - node:
        # init signature is init(service):
        # There is a global dicst named DataStore that can be used to store variables
        init: |
            DataStore['node'] = service.newStruct('Node')
            DataStore['node'].setField('sensorName', 'node01')
            DataStore['node'].setField('sensorA', 25)
            DataStore['node'].setField('sensorB', 65)
            node = DataStore['node']
            service.print('\nCreating Sensor node:\n   name: {0}\n   sensorA: {1}\n  ␣
↪sensorB: {2}\n'.format(node.getField('sensorName'),node.getField('sensorA'),node.
↪getField('sensorB') ))

            def myFunc():
                service.print('myFunc called')


        #handlers fill out a function with the signature <name>_handler(service, req,␣
↪resp):
        # you can print out to the console with service.print(text)
        handlers:
```

(continues on next page)

(continued from previous page)

```
            #Use packets/nodes can be copied to eachother. All shared fields that are␣
↪present in the source will get copied to the destination
            - SetData: |
                req.copyTo(DataStore['node'])

            - GetData: |
                DataStore['node'].copyTo(resp)

        #You can add custom commands to an agent that will be loaded in for autocomplete␣
↪and help menus in the CLI
        commands:
            - rename:
                desc: renames the node
                args:
                    - name: {desc: new name for node, default: new_name}
                handler: |
                DataStore['node'].setField('sensorName', name)
                service.print('\nRenaming Sensor node:\n   name: {0}\n'.format(name))
```

Note: Agents can be loaded by adding the '-a <agent_name>' flag when running the CLI, or using the loadAgent command in the CLI. If an agent named 'default' is present, it will be loaded automatically when the CLI is started.

Each *agent* has 3 sections:

init:

This is run when the agent is loaded. It is used to initialize the agent and set up any variables that will be used in the handlers. This block of code is executed in the global scope, so functions defined here will be available to the handlers. This section can also be used to import modules that will be used in the handlers.

handlers:

This is a list of packet handlers. The name of the handler must match the name of the packet it handles.

The signature of the handler is: <name>_handler(service, req, resp)

- service - The poly packet service. This is used to access the packet data and send packets
- req - The incoming request packet
- resp - the outgoing response packet

commands:

This is a list of custom commands that can be run from the CLI. The name of the command is the name of the command that will be run from the CLI.The handler is a python script that will be run when the command is called.

The signature of the command handler is: <name>_cmd_handler(service, args)

- service - The poly packet service. This is used to access the packet data and send packets
- args - A dictionary of the arguments passed to the command. The keys are the names of the arguments and the values are the values passed in. * If no value is passed in, the default value will be used. If no default value is specified, the argument will be None * args are defined in the handler, so you can use them by name without needing to use *args['name']*

4.2.8 Plugins:

Protocol files can include other protocol files. This allows you to create a library of common packets and structs that can be used across multiple protocols. To inlude a protocol file, use the *Plugins* directive.

```
plugins:
    - https://gitlab.com/uprev/public/mrt/Modules/Utilities/OTA/poly/ota-protocol.yml:
↪{prefix: ota}
    - /path/to/protocol2.yml
```

- Plugin paths can be local or a url.
- The prefix is used to prefix all packets and fields in the plugin. This can be used to avoid name collisions between plugins and the base protocol

4.3 Step 2: Generating the Code

`poly-make` is the tool that will turn the yaml description into c code for projects.

```
poly-make -i my_protocol.yml -o . - a
```

-i
: sets the input file

-o
: tells it where to create the C files for the service

-a
: tells the tool to create the application layer (this is not required, but is a helpful starting point)

4.4 Step 3a: Using The Code C/C++

The C code generated for the service in step 2 relies on the MrT module `/Utilities/PolyPacket`.

4.4.1 Initializing service

To initialize a service call the service_init function.

Note: all service functions are prepended with the service prefix to allow multiple services to co-exist

```
sp_service_init(1, 8); //initialize the service with 1 interface, and a spool size of 8
```

This example initalizes the service with 1 interface. An `interface` is an abstract port into and out of the service. If your device needs to use the protocol on multiple hardware ports (Uart, TCP/IP, SPI, etc..) each one of these would have its own interface.

The `Spool size` just determines how much memory the message spool (per interface) uses. With a size of 8, we can have 8 messages on the outgoing spool for each interface at a time. This really only comes into play when we are using auto-retries since packets stay on the spool until they are acknowledged or exceed the max-retry count.

4.4.2 Register Tx functions

For each interface we need to register a send function. This allows the service to handle the actual sending so we can automate things like acknowledgements and retries. There are two types of send callbacks that can be registered:

```
typedef HandlerStatus_e (*poly_tx_byte_callback)(uint8_t* data , int len);
typedef HandlerStatus_e (*poly_tx_packet_callback)(poly_packet_t* packet );
```

The `poly_tx_byte_callback` will pass the packet as an array of COBS encoded bytes which can be sent directly over a serial connection.

The `poly_tx_packet_callback` will pass a reference to the packet itself which can be converted to JSON, or manipulated before sending.

```
sp_service_register_tx_bytes(0, &uart_send ); // register sending function for raw bytes␣
↪on interface 0

sp_service_register_tx_packet(0, &json_send ); // register sending function for entire␣
↪packet on interface 0
```

once we have registered a callback for an interface, we can send messages to it using the quick send functions generated for the service.

```
sp_sendGetData(0); // Sends a 'GetData' packet over interface 0
```

4.4.3 Feed the service

The underlying service is responsible for packing and parsing the data. So wherever you read bytes off of the hardware interface, just feed them to the service.

```
void uart_rx_handler(uint8_t* data, int len)
{
    sp_service_feed(0, data, len); //feed the bytes to interface 0
}
```

From here the service will take care of parsing the data and dispatching messages to the proper message handler.

4.4.4 Sending messages

The service creates one-liner functions for easily sending simple messages

Using the example protocol we can send a message to get data from a remote device on interface 0 with:

```
sp_sendGetData(0); //send a 'GetData' packet over interface 0
```

for packet types with data fields, the datafields get turned into the arguments for the function

Note: Only 'required' fields can be used as arguments

```
sp_sendData(0, 97, 98, "My Sensor name"); //send a 'Data' packet over interface 0
```

Occasionally you may need to send a packet , but do not want to use the quick-send functions. an example of this would be sending a packet that includes optional fields. This can be done by using the <prefix>_packet_build function:

```
sp_packet_t msg;
sp_packet_build(&msg,SP_DATA_PACKET);
```

next we set fields in the message

```
sp_setSensorA(msg,97 );
sp_setSensorName(msg,"my sensor");
```

```
sp_send(0,&msg);
```

Important: If you build a package, but do not send it, be sure to clean it! The safest practice is to just always clean it. There is no harm in cleaning a packet that has been sent.

```
sp_clean(&msg);
```

4.4.5 Receive Handlers

The generated service creates a handler for each packet type, they are created with weak attributes, so they can be overridden by just declaring them again in our code. If you specify a response for a packet in the YAML, the service will initialize that packet and pass a reference to the handler.

The handler can return the following statuses:

PACKET_HANDLED
: service will respond with the response packet (or an ack if none is specified)

PACKET_UNHANDLED
: packet will drop through to the **Default_handler**

PACKET_IGNORED
: packet will be ignored and skip the default handler

The following is our handler for '*SetData*' type packets

```
/**
  *@brief Handler for receiving GetData packets
  *@param GetData incoming GetData packet
  *@param Data Data packet to respond with
  *@return handling status
  */
HandlerStatus_e sp_GetData_handler(sp_packet_t* sp_GetData, sp_packet_t* sp_Data)
{
    //set the fields of the responese packet
    sp_setSensorA(sp_Data, 97);
    sp_setSensorB(sp_Data, 98);
    sp_setSensorName(sp_Data, "My sensor");

    return PACKET_HANDLED;  //respond with response packet
}
```

4.4.6 Process

The service is meant to be run on many platforms, so it does not have built in threading/tasking. For it to continue handling messages, we have to call its process function either in a thread/task or in our super-loop

```
while(1)
{
sp_service_process();
}
```

4.5 Step 3b: Using The Code JSON

If you are working with json you can register a **poly_tx_packet_callback** and convert your packets to json strings for sending.

```
HandlerStatus_e json_send(poly_packet_t* packet)
{
    char buf[256];
    int len;

    len = sp_print_json(packet, buf); //print json string to buffer
    some_tcp_function(buf, len);      //send json string out

    return PACKET_SENT;
}
```

after you initialize the service, register the callback:

```
sp_service_register_tx_packet(0, &json_send ); // register sending function for entire␣
↪packet on interface 0
```

Now when messages are sent out on interface 0, they will be converted to json strings and sent out with `some_tcp_function`.

4.5.1 Handling JSON packets

For handling incoming json packets, there are two options. you can feed the json message to the service for normal handling or call the json handler to bypass the normal service queue. This option makes it easy to use the service in synchronous tasks such as responding to an http request

4.5.2 Async JSON

```
void app_json_async_handler(char* strJson, int len)
{
    sp_service_feed_json(0,strJson, len);
}
```

4.5.3 Sync JSON

```
void app_json_sync_handler(const char* strRequest, int len, char* strResp)
{
    HandlerStatus_e status;
    status = sp_handle_json(strRequest, len, strResp);
}
```

4.6 PolyPacket CLI Tool

Once you have a descriptor file, you can run a live interface of the protocol using poly-packet

Open two terminals and connect them over udp to test it out:

terminal 1:

```
poly-packet -i sample_protocol.yml -c udp:8020
```

terminal 2:

```
poly-packet -i sample_protocol.yml -c udp:8010:8020
```

Note: The tool can connect over `tcp`, `udp`, and `serial`

The terminal interface uses autocompletion, so hit tab to show available packet/ field types. To send a packet just type the packet name followed by comma seperated field names and values.

example: .. code-block:: bash

    Data sensorA: 45, sensorB: 78, sensorName: mySensor

The instance of the service running on port 8020 will respond to the packet with an 'ack'

CHAPTER

FIVE

MODULES

This section contains the documentation for the individual modules. They are all pulled from the modules during build/tests of the main MrT repo.

5.1 Platforms

5.1.1 Platform-NRF5

To use the NRF5 Abstraction layer, create a repo with an NRF5 project.

Use the mrt-config tool to add in submodules. Make sure to import the **Platforms/Common** and **Platforms/NRF5 modules**

Setting the Platform

To set the project to use the NRF5 Abstraction module you need to create an **MRT_PLATFORM** symbol with a value of **MRT_NRF5**. If using the nrf5 project template, this can be done by adding the following linke to `Makefile`:

5.1.2 Linux

Platform abstraction for linux

5.1.3 ESP32

Requires: Modules/Platforms/Common

Following the example projects to create a template, you should end up with a main directory containing a component.mk file.

add the following lines to this coponent.mk, filling in the modules used...

```
CFLAGS+= -DMRT_PLATFORM=MRT_ESP32

COMPONENT_ADD_INCLUDEDIRS := Path/To/MrT/Modules  Path/To/MrtModules/<module-1-path>␣
↪Path/To/MrtModules/<module-2-path>

COMPONENT_SRCDIRS := Path/To/MrT/Modules  Path/To/MrtModules/<module-1-path> Path/To/
↪MrtModules/<module-2-path>
```

This is plannned to be improved so you dont have to list each module path

5.1.4 STM32

To use the stm32 Abstraction layer, create a repo with an STM32 project. The recommended tool is the STM32CubeIDE

Use the mrt-config tool to add in submodules. Make sure to import the **Platforms/Common** and **Platforms/STM32 modules**

Note: after importing modules, right click the project and hit refresh so it sees the new directories

---

To use the STM32 platform, cofigure the following settings:

- Project->Properties->C/C++ General->Path and Symbols:
  - Under the Symbols tab add a symbol named **MRT_PLATFORM** with the value **MRT_STM32_HAL**
  - Under the Source Location tab click add and select the **Modules** directory under Mr T
  - Under the Includes tab, click add and add the path to the **Modules** directory under Mr T

Troubleshooting common problems

main.h no such file

- ***main.h* no such file or directory**
  This issue is normally accompanied by a wrench icon on the MrT directory which indicates local directory settings overriding the workspace settins. To correct this, right click the folder and click *Resource configurations -> Reset to Defaults*

Using ACI BLE

Important: deprecrated. Gatt Interface should now use the *stm32_gatt_adapter*

To use the STM32 ACI interface for BLE:

- Project->Properties->C/C++ General->Path and Symbols: * Under the Symbols tab add a symbol named `STM32_GATT_MODULE_ENABLED`

Generate the services/profile using mrt-ble

The output will be a header/source for each service, and a header/source for the profile. In main.c, before 'APPE_Init();' register the profile init function:

```
MRT_GATT_REGISTER_PROFILE_INIT(example_profile_init);
```

When the server is initialized by the system it will create and register all services and characteristics. To update a value use:

```
MRT_GATT_UPDATE_CHAR(&env_svc.mTemp, (uint8_t*)&temp, sizeof(temp)); /* replace env_svc.
↪mTemp with a char in one of your services*/
```

Enabling printf

The Stm32 programmers use the SWO pin to print messages back to the host. This can be useful to log out messages to the console for debug. To enable printf to work through the SWO pin follow these steps:

1. add '-lc -lrdimon' to linker flags
2. in the Debug configuration (little arrow by the bug icon) under 'Start Up' tab add "monitor arm semihosting enable" to initialization commands
3. add the following code snippets:

top of main.c:

```
#include "stdio.h"

int __io_putchar(int ch)
{
  ITM_SendChar(ch);
  return(ch);
}


int _write(int file, char *ptr, int len)
{
  int DataIdx;
  for (DataIdx = 0; DataIdx < len; DataIdx++)
  {
        __io_putchar(*ptr++);
  }
  return len;
}

extern void initialise_monitor_handles(void);
```

inside main()

```
initialise_monitor_handles();
```

5.1.5 Atmel

Requires: Modules/Platforms/Common

To use with an atmel asf project, include the Mr T repo as a submodule of the project

(for more detailed instruction visit the README from the MrT/Config Repo)

Integrating to Atmel Studio

Once you have created your project and imported the Mr T modules needed, open the project in Atmel Studio and follow these steps:

1. Click the 'Show All files' button in the solution explorer
2. right click mrt/Modules/mrt_platform, and select 'Include in project'
3. go to the poject properties and add the symbol for your framework

MRT_PLATFORM_ATMEL_ASF for atmel asf projects **MRT_PLATFORM_ATMEL_START** for atmel-start projects

1. go to the poject properties and add 'src/mrt/Modules' as an include path
2. for each module you would like to use, right click the module directory in the solution explorer, and click 'Include in project'

5.1.6 Common

This module defines definitions and functions common to all platforms. It must be included with any project that uses on the of the Platform abstractions.

FreeRTOS

To enable FreeRTOS add the symbol **MRT_FREERTOS** to the project or define it above the include statement for **mrt_platform.h**. This will override the malloc and free functions, and map the MRT_MUTEX macros accordingly:

```
/**
  *@file mrt_FreeRTOS.h
  *@brief Abstraction header FreeRTOS
  *@author Jason Berger
  *@date 8/27/2020
  */

#pragma once

#include "cmsis_os.h"
#include "semphr.h"


#define malloc(size) pvPortMalloc(size)
#define free(ptr) vPortFree(ptr)

#define MRT_MUTEX_TYPE SemaphoreHandle_t
#define MRT_MUTEX_CREATE(m) (m) = xSemaphoreCreateMutex()
#define MRT_MUTEX_LOCK(m) xSemaphoreTake((m), portMAX_DELAY)
#define MRT_MUTEX_UNLOCK(m) xSemaphoreGive((m))
#define MRT_MUTEX_DELETE(m) vSemaphoreDelete((m))
```

5.2 Utilities

5.2.1 GFX

ColorGfx

Module for graphics buffering. The module is intended to handle graphics using multiple color modes (mono,565,24bit). Once functional this will be able to replace the existing MonoGFX. One benefit to this ithe ability to convert assets between color modes for displaying on any display. example: a color image could be displayed on a mono chromatic display, or a tri-color which buffers as 3 separate monochramitic canvases.

the gfx_t struct can be initialized 'buffered' or 'unbuffered'. When it is buffered, it allocates its buffer in memory and works with the local copy. When it is unbuffered, all drawing functions are sent to the callback function for writing pixels. This allows the use of displays with areas too large to store in ram.

buffered example:

```
gfx_t gfx;

//initialize a 128x32 canvase
gfx_init_buffered(&gfx,128,32, GFX_COLOR_MODE_888);

//Set pen with stroke of 1 pixel and color red
gfx_set_pen(&gfx, 1, GFX_COLOR_RED);

//draw a 30x20 rectangle at x,y = 5,5 and fill it in
gfx_draw_rect(&gfx, 5,5,30,20, GFX_OPT_FILL);
```

5.2.2 Audio

utility-audio-test

Utility for various audio testing

utility-AudioXcoder

Transcoding utility for audio data

5.2.3 Interfaces

Gatt Interface

Backend C code for mrt-ble generated Gatt Profiles.

5.2.4 OTA

This module provides utility functions and structs for managing OTA (Over the Air) update images in memory.

Typical OTA Update processes can be broken down into several steps:

1. Staging the update image (Downloading the update image to a staging area in memory)
2. Applying the update (Moving the update from the staging area to program memory)
3. Rebooting the device to run the new firmware

OTA Image Manager

The *ota_img_mgr_t* struct provides a way to manage multiple OTA disks. This is useful if you have multiple storage devices, such as an SD card and a SPI flash chip.

- **ota_img_mgr_t** - Collection of ota_dsk_t structs.
- **ota_dsk_t** - A disk is a storage device. This can be a SPI flash chip, EEPROM, embedded flash, SD card, etc.
- **ota_partition_t** - A partition is a section of a disk. A disk can be split up into multiple partitions. Each partition contains a single image
- **ota_img_t** - An image is a file that contains the update data. This is usually a binary file.

Initializing Staging disk

```
#include "ota_img.h"
ota_img_mgr_t ota_mgr;

void spi_flash_write(uint32_t addr, uint8_t *data, uint32_t len)
{
    //TODO write to flash
}

void spi_flash_read(uint32_t addr, uint8_t *data, uint32_t len)
{
    //TODO read from flash
}

int main(void)
{
    ota_img_mgr_init(&ota_mgr);

    ota_dsk_t* stagingDsk = ota_img_mgr_add_dsk(&ota_mgr, "staging", spi_flash_write,␣
↪spi_flash_read);


    //If there are no partitions, add them this is first boot, create partitions
    if(stagingDsk->partitionCount == 0)
    {
        //If dsk has not been partitioned, add partitions
        ota_dsk_add_partition(stagingDsk, 0, 56000, "firmware");
        ota_dsk_add_partition(stagingDsk, 0, 56000, "fpga");
    }


}
```

Staging an OTA Update image

```
#include "ota_img.h"
ota_img_mgr_t ota_mgr;

ota_partition_t* firmware_partition = NULL;
ota_partition_t* fpga_partition = NULL;

..

void ota_firmware_block_callback(uint32_t offset, uint8_t *data, uint32_t len)
{
    ota_partition_write_image( firmware_partition, offset, data, len)
}

int main(void)
{
    ota_img_mgr_init(&ota_mgr);

    ota_dsk_t* stagingDsk = ota_img_mgr_add_dsk(&ota_mgr, "staging", spi_flash_write,␣
↪spi_flash_read);

    //If there are no partitions, add them this is first boot, create partitions
    if(stagingDsk->partitionCount == 0)
    {
        //If dsk has not been partitioned, add partitions
        ota_dsk_add_partition(stagingDsk, 0, 56000, "firmware");
        ota_dsk_add_partition(stagingDsk, 0, 56000, "fpga");
    }

    firmware_partition = ota_dsk_get_partition(&stagingDsk, "firmware");

    while( 1)
    {
        //TODO Request next block from server
    }


}
```

Applying the Update

This step would usually take place in the bootlaoder.

If the staging area is already in a location where it can be executed, then you can just jump to that location. If there are multiple images for a ping-pong style update, then you can use the *ota_dsk_get_active_partition* function to get the active partition. only one partition can be active at a time. Setting a partition as active will set the other partition(s) as inactive.

```
#include "ota.h"
ota_img_mgr_t ota_mgr;
```

(continues on next page)

(continued from previous page)

```
void stage_update() //This would be called during the staging process
{
    ota_ctx_init_staging(&spiFlash, spi_flash_write, spi_flash_read);
    ota_img_mgr_init(&ota_mgr);
    ota_dsk_t* stagingDsk = ota_img_mgr_add_dsk(&ota_mgr, "staging", spi_flash_write,␣
↪spi_flash_read);

    ota_partition_t* partA = ota_dsk_get_partition(stagingDsk, "firmwareA");
    ota_partition_t* partB = ota_dsk_get_partition(stagingDsk, "firmwareB");

    if(partA->flags && OTA_PARTITION_FLAG_ACTIVE)
    {

        //TODO Write new image to partB

        ota_partition_set_active(partB);
    }
    else
    {
        //TODO Write new image to partA
        ota_partition_set_active(partA);
    }

}


int launch_application(void)  //This would be called during the boot process
{

    ota_img_mgr_init(&ota_mgr);
    ota_dsk_t* stagingDsk = ota_img_mgr_add_dsk(&ota_mgr, "staging", spi_flash_write,␣
↪spi_flash_read);


    //Get the active partition
    ota_partition_t* active_part = ota_dsk_get_active_partition(stagingDsk); //This will␣
↪return partA if it is active, or partB if it is active.
    printf("Active partition is %s", active_part->label  );
    //Jump to active_part->image.addr
}
```

If the staging area is not an executable location, then you will need to copy the image to an executable location. This can be done manually, or by giving the ota_dsk struct a read and write callback for the destination dsk and using the *ota_partition_apply_update* function. This function will copy the image to the destination and then verify it with the provided CRC32.

```
#include "ota.h"
#include "crc32.h"
ota_img_mgr_t ota_mgr;

ota_dsk_t* stagingDsk = NULL;
```

(continues on next page)

(continued from previous page)

```
ota_dsk_t* nvsDsk = NULL;

#define BLOCK_SIZE 256
#define APPLICATION_ADDR 0x10000

mrt_status_t nvs_write(uint32_t addr, uint8_t *data, uint32_t len)
{
    //TODO write to nvs
}

mrt_status_t nvs_read(uint32_t addr, uint8_t *data, uint32_t len)
{
    //TODO read from nvs
}

int main(void)
{
    mrt_status_t status = MRT_STATUS_OK;
    ota_ctx_init(&ota, spi_flash_write,  spi_flash_read, nvs_write, nvs_read);

    ota_img_mgr_init(&ota_mgr);
    stagingDsk = ota_img_mgr_add_dsk(&ota_mgr, "staging", spi_flash_write, spi_flash_
↪read);
    nvsDsk = ota_img_mgr_add_dsk(&ota_mgr, "nvs", nvs_write, nvs_read);

    ota_partition_t* firmware_staging_partition = ota_dsk_get_partition(stagingDsk,
↪"firmware");
    ota_partition_t* firmware_exec_partition = ota_dsk_get_partition(nvsDsk, "firmware");


    //IF the firmware partition is not null and the new flag is set, then apply the␣
↪update
    if((firmware_staging_partition == NULL) && (firmware_partition->flags && OTA_
↪PARTITION_FLAG_NEW) && (firmware_exec_partition != NULL))
    {

        status = ota_partition_copy(firmware_staging_partition, firmware_exec_partition);
↪//This will copy the image from the staging area to the destination address in NVS,␣
↪then verify it with the CRC32


        if(status == MRT_STATUS_OK)
        {
            //TODO - JUMP TO  firmware_exec_partition->image.addr
        }
        else
        {
            //TODO - handle error
        }

    }
    else
```

(continues on next page)

(continued from previous page)

```
    {
        // No new update in staging area, jump to application
        //TODO - JUMP TO firmware_exec_partition->image.addr
    }


}
```

OTA XFer

ota_xfer.h/c provides an ota trasnfer utility. This utility can be used to manage the transfer of an OTA image from a server (or host) to the staging area. It will keep track of which blocks have been received, and which blocks are missing. It will also keep track of the state of the transfer, and verify the image with the provided CRC32.

```
#include "ota.h"
#include "ota_xfer.h"

ota_img_mgr_t img_mgr;
ota_xfer_t xfer;
ota_partition_t* firmware_partition = NULL;


//Callback when block data packet is received
void ota_firmware_block_callback(uint32_t offset, uint8_t *data, uint32_t len)
{
    ota_partition_write_image( firmware_partition, offset, data, len)

    ota_xfer_write_block(&xfer, offset, data, len);

    if(xfer.state == OTA_STATE_FINISHED)
    {
        Reset the device
    }

    //TODO request next block
}

//Call back when ota start packet is received
void ota_xfer_callback(const char* label, const char* strVersion, uint32_t size, uint32_
↪t crc)
{
    //Kick off new transfer
    ota_xfer_init(&xfer, &ota, label, strVersion, size, crc);

    ota_xfer_set_state(&xfer, OTA_STATE_BULK);
}

int main(void)
{
    //Set up staging
    ota_img_mgr_init(&img_mgr);
```

(continues on next page)

(continued from previous page)

```
    ota_dsk_t* stagingDsk = ota_img_mgr_add_dsk(&img_mgr, "staging", spi_flash_write,␣
↪spi_flash_read);
    firmware_partition = ota_dsk_get_partition(stagingDsk, "firmware"); //This will␣
↪return the firmware partition if it exists, or NULL if it does not exist

    if(firmware_partition == NULL)
    {
        //TODO - handle error
    }

    while(1)
    {
        if(xfer.state == OTA_STATE_BULK)
        {

            //Get next missing block
            uint32_t nextBlock = ota_xfer_get_next_missing_block(&xfer);
            if(nextBlock > -1)
            {
                request_image_block(nextBlock * ota->blockSize, ota->blockSize);
            }
        }
    }


}
```

PolyPacket Protocol

Included in this module is a PolyPacket protocol descriptor file *poly/ota-protocol.yml*. This file can be used to generate a PolyPacket protocol for OTA transfers. The protocol can be used by itself or included in a larger protocol as a plugin.

Generating protocol service

For information on generating code for the ota protocol service, see the PolyPacket documentation.

Pushing Images to device

The protocol descriptor includes *Agent* profiles for the *otaHost* and *otaDevice*. The *otaDevice* agent simply simulates a device that can receive OTA images, and can be used for testing. The *otaHost* agent can be used as a utility for reading partitions and trasnfering images to the device.

1. Setup a simulated device with the *otaDevice* agent.

```
poly-packet -i ota-protocol.yml -a otaDevice -c tcp:8020
```

This will start a simulated device that will listen for connections on port 8020.

2. Setup the *otaHost* agent to connect to the device. Run the following command in a new terminal window.

```
poly-packet -i ota-protocol.yml -a otaHost -c tcp:localhost:8020
```

This will start the *otaHost* agent and connect to the device.

```
______     _      ______         _        _
| ___ \   | |     | ___ \       | |      | |
| |_/ /__ | |_   _| |_/ /_ _  ___| | _____| |_
|  __/ _ \| | | | |  __/ _` |/ __| |/ / _ \ __|
| | | (_) | | |_| | | | (_| | (__|   <  __/ |_
\_|  \___/|_|\__, \_|  \__,_|\___|_|\_\___|\__|    [ota protocol]
              __/ |
             |___/

 TCP trying localhost:8020
 TCP Connected
 agent Profile 'otaHost' Loaded
```

3. Run the *discover* command to get a list of partitions on the device.

```
>>> discover
Discovery Complete!

Disk/Partition              Address          Used/Size     Version   CRC            Flags
------------------------------------------------------------------------------------------------------
envm/bootloader             0x00000000         0B/24.0K              0x00000000     ---
envm/firmware               0x00005DC0         0B/232.0K             0x00000000     ---
spi-flash/bootloader        0x00000000         0B/24.0K              0x00000000     ---
spi-flash/firmware          0x00005DC0         0B/232.0K             0x00000000     ---
spi-flash/fpga              0x0003E800         0B/116.0K             0x00000000     ---
```

4. Use the flash command to transfer an image to the desired partition.

```
flash file: firmware.hex, version: 1.0.1, partition: spi-flash/firmware
```

```
>>> flash (file: firmware.hex, version: 1.0.1, partition: spi-flash/firmware, addr: 0)
100% ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

 Flashing Complete
```

Note: The CLI has tab complete which will show available commands and arguments

5. Run the *discover* command again to verify that the image was transferred. The *V* flag on the partition indicates the device has verified the image with the CRC sent by the host.

```
>>> discover
Discovery Complete!

Disk/Partition              Address          Used/Size     Version   CRC            Flags
------------------------------------------------------------------------------------------------------
envm/bootloader             0x00000000         0B/24.0K              0x00000000     ---
envm/firmware               0x00005DC0         0B/232.0K             0x00000000     ---
spi-flash/bootloader        0x00000000         0B/24.0K              0x00000000     ---
spi-flash/firmware          0x00005DC0      23.8K/232.0K   1.0.1     0xC829F23F     -VN
spi-flash/fpga              0x0003E800         0B/116.0K             0x00000000     ---
```

5.2.5 CRC

This module provides utility functions for calculating cyclic redundancy check (CRC) values. Right now only CRC32 is supported as it is the most common, but more will be added as needed

Use

The module can calculate CRCs on a buffer in one run, or in multiple chunks for larger buffers.

Single Chunk

```
#include "Utilities/CRC/crc32.h"


uint32_t crc = crc32(buffer, buffer_length);
```

Multiple Chunks

```
#include "Utilities/CRC/crc32.h"

crc32_ctx_t crc_ctx;

crc32_init(&crc_ctx);

crc32_update(&crc_ctx, buf0, buf0_len);
crc32_update(&crc_ctx, buf1, buf1_len);
crc32_update(&crc_ctx, buf2, buf2_len);

uint32_t crc = crc32_result(&crc_ctx);
```

5.2.6 ByteFifo

This module provides a simple byte byte_fifo in pure C. Unless there are heavy resource constraints, it is recommended to use the regular Fifo module.

byte_fifos can be defined staticly or initiallized dynamicly

dynamic example:

```
#include "Modules/Utilities/byte_fifo.h"

byte_fifo_t my_fifo;

uint8_t myBuf[64];

int main(void)
{
    //creates a byte_fifo that can store 64 uin16_t
    byte_fifo_init(&my_fifo, 64);
```

(continues on next page)

(continued from previous page)

```
    uint16_t myData = 0;
    for(int i =0; i < 64; i++)
    {
        myData++;
        byte_fifo_push(myData); //
    }

    byte_fifo_pop_buf(&my_fifo, myBuf, 64);

    return 0;
}
```

static example:

```
#include "Modules/Utilities/byte_fifo.h"

byte_fifo_DEF(my_fifo, 64); //Expands to:
/*
uint8_t my_fifo_data[64];
byte_fifo_t my_fifo = {
    .mBuffer = my_fifo_data,
    .mHead = 0,
    .mTail = 0,
    .mMaxLen = 64,
    .mCount = 0,
};
*/

uint8_t myBuf[64];

int main(void)
{
    uint8_t myData = 0;
    for(int i =0; i < 64; i++)
    {
        myData++;
        byte_fifo_push(myData); //
    }

    byte_fifo_pop_buf(&my_fifo, myBuf, 64);

    return 0;
}
```

The main benefit of the static define is that it uses an array of 'type' to hold the data. This can help with debugging when the type is a struct.

5.2.7 COBS

Module for Consistent Overhead Byte Stuffing https://en.wikipedia.org/wiki/Consistent_Overhead_Byte_Stuffing

Consistent Overhead Byte Stuffing (COBS) is an algorithm for encoding data bytes that results in efficient, reliable, unambiguous packet framing regardless of packet content, thus making it easy for receiving applications to recover from malformed packets. It employs a particular byte value, typically zero, to serve as a packet delimiter (a special value that indicates the boundary between packets). When zero is used as a delimiter, the algorithm replaces each zero data byte with a non-zero value so that no zero data bytes will appear in the packet and thus be misinterpreted as packet boundaries.

cobs.c/cobs.h

These provide the basic cobs utility for encoding/decoding a buffer of data.

cobs_fifo.c / cobs_fifo.h

This is a fifo which uses cobs encoding to keep track of 'frames' inside of the fifo. a frame is a single buffer of data.

Working with frames

You can push/pop entire frames with the fifo

```
cobs_fifo_t fifo;
uint8_t buf[32];            //tmp buffer
int len;

cobs_fio_init(&fifo, 256); // create a cobs fifo that can store 256 bytes

uint8_t frameA[] = { 0x11, 0x22, 0x00, 0x33};
uint8_t frameB[] = { 0x12, 0x34};

cobs_fifo_push_frame(&fifo, frameA, 4); //push frame A into fifo. The frame is encoded␣
↪as it is pushed into the fifo
//fifo->mNextLen is now 6. because frame A has 4 bytes + overhead byte and 1 byte for␣
↪the delimiter

cobs_fifo_push_frame(&fifo, frameB, 2); //push frame B into fifo. The frame is encoded␣
↪as it is pushed into the fifo
//fifo->mNextLen is still 6. because frame A is still the first frame in the buffer

len = cobs_pop_frame(&fifo, buf, 32); //pop and decode next frame from fifo
//fifo->mNextLen is now 4 because the next frame is frame B (2 bytes + 1 overhead + 1␣
↪delimiter)

len = cobs_pop_frame(&fifo, &buf[len], 32); //pop and decode next frame from fifo
//len will be the size of frame B decoded (2 bytes), buf = [0x12, 0x34]
```

Working with Raw Bytes

if you are sending or receiving bytes over serial and need to handle encoded data, you can use the _buf functions instead of _frame

Sender

```
uint8_t frameA[] = { 0x11, 0x22, 0x00, 0x33};
uint8_t buf[256];

cobs_fifo_push_frame(&fifo, frameA, 4); //push frame A into fifo. The frame is encoded␣
↪as it is pushed into the fifo


int len = cobs_fifo_pop_buf(&fifo, buf, 32 ); // pop the encoded frame for sending over␣
↪serial
//len is 6 (4 data bytes + 1 overhead + 1 delimiter )

//write delimited data to serial
uart_tx(buf, len); // buf will be [ 0x03, 0x11, 0x22, 0x02, 0x33, 0x00]
```

Receiver

```
uint8_t buf[256];

int len = uart_rx(buf, 256); // using buf from send example [ 0x03, 0x11, 0x22, 0x02,␣
↪0x33, 0x00]

cobs_fifo_push_buf(&fifo, buf, len); //push raw data into fifo


len = cobs_fifo_pop_frame(&fifo, buf, 32 ); // pop and decode next frame
//len = 4, buf = [ 0x11, 0x22, 0x00, 0x33 ]
```

utility-json

5.2.8 PolyPacket

This Module Contains the back-end C code for protocols and services generated using the PolyPacket tool.

To generate services for this module, install the PolyPacket Tool:

```
pip3 install polypacket
```

Packing

This section describes how packets are serialized. Each packet contains a header and an optional data section. If a packet contains no fields (for instance an Ack) there is no data section and the Data Len is 0.

Header

Byte	0	1	2	3	4	5	6	7
Field	typeId	reserved	Data Len		Token		Checksum	
Type	uint8	uint8	uint16		uint16		uint16	

When a packet contains fields, the fields are serialized as field blocks and placed in the data section.

Field Block

Simple (single value) fields contain a typeId and the value. The parser determines the type of the value by looking up the typeId in the field descriptor dictionary

Byte	0	1: 1+ (n/127)	•	•	•
Field	typeId	Array Len (n)	Value[0]	Value[1]	Value[2]
Type	uint8	varsize	DataType	DataType	DataType

If the field is an array/ string, it contains a Length and all of the values present in the array. The Length indicates the number of values present in the array. Again we get the size of each value by looking up the typeId in the field descriptor dictionary

varSize

The varSize type stores a number between 0 and 2^28, but uses the least amount of bytes required. each byte contains 7 value bits, and one 'continue' bit. to read the value, you shift in the lower 7 bits, if the highest bit is set, then the value is continued on the second byte. This repeats until you get a 0 for the 'continue' bit.

```
/*    Variable Size value packing
*    These functions are used for packing and reading variable sized values
*    This allows effecient packing of small values with the flexibility to still use␣
↪larger values (up to 2^28). anything under 7bits is not affected
*    each packed byte represents 7bits of the value, the most signifacant bit is used to␣
↪indicate if the value is continued on the next byte
*    example 0x0321 would be packed to [0xA1, 0x06]
*            0X21 & 0X80 = 0XA1
*            0x03 << 1 = 0x06 //We shift one bit for each byte to compensate for the bit␣
↪used as the continuation flag
*/

int poly_var_size_pack(uint32_t val, uint8_t* buf)
{
uint8_t  tmp = 0;
```

(continues on next page)

(continued from previous page)

```
int idx =0;

do{
    tmp = val & 0x7F;
    val >>= 7;
    if(val > 0)
    {
    tmp |= 0x80;
    }

    buf[idx++] = tmp;

} while(val > 0);

return idx;
}
```

5.2.9 Fifo

This module provides a Generic 'type' fifo in pure C. It is not as effecient as a typed fifo, but it provides the flexibity of storing different types and structs.

Fifos can be defined staticly or initiallized dynamicly

dynamic example:

```
#include "Modules/Utilities/fifo.h"

fifo_t myFifo;

uint16_t myBuf[64];

int main(void)
{
    //creates a fifo that can store 64 uin16_t
    fifo_init(&myFifo, 64, sizeof(uint16_t));

    uint16_t myData = 0;
    for(int i =0; i < 64; i++)
    {
        myData++;
        fifo_push(&myData); //
    }

    fifo_pop_buf(&myFifo, myBuf, 64);

    return 0;
}
```

static example:

```c
#include "Modules/Utilities/fifo.h"

FIFO_DEF(myFifo, 64, uint16_t); //Expands to:
/*
uint16_t myFifo_data[64];
fifo_t myFifo = {
    .mBuffer = myFifo_data,
    .mHead = 0,
    .mTail = 0,
    .mMaxLen = 64,
    .mCount = 0,
    .mObjSize = sizeof(uin16_t)
};
*/

uint16_t myBuf[64];

int main(void)
{
    uint16_t myData = 0;
    for(int i =0; i < 64; i++)
    {
        myData++;
        fifo_push(&myData); //
    }

    fifo_pop_buf(&myFifo, myBuf, 64);

    return 0;
}
```

Note: The main benefit of the static define is that it uses an array of 'type' to hold the data. This can help with debugging when the type is a struct.

5.3 Devices

5.3.1 Memory

FL-S Series NOR Flash Memory

Infineon's FL-S serial Flash memory provides fast quad SPI NOR Flash memory with densities from 128 Mb to 1 Gb for high-performance embedded systems. The FL-S family is AEC-Q100 qualified and supports PPAP for automotive customers at extended temperature ranges of -40°C to +125°C.

The FL-S family brings read speeds in single, dual, and quad I/O modes up to 133 MHz SDR (single data rate), and up to 80 MHz DDR (double data rate) delivering read bandwidth of up to 80 Mbps. Industry-leading programming performance (up to 1.08 Mbps) speeds manufacturing throughput and lowers programming costs dramatically.

Device-Eeprom

SpiFlash

Datasheet: http://www.adestotech.com/wp-content/uploads/DS-AT25SF041_044.pdf

Driver for spi flash device.

5.3.2 Displays

ST727A

Device Driver for SSD1306 based oled displays

ERCxxLcd

Datasheet: https://www.mikrocontroller.net/attachment/10245/SED1565.pdf

Requires: Modules/Utilities/GFX/MonoGfx

This module is a driver for the ERC monochromatic lcd displays driven by the SED15xx driver IC

This module handles mapping the pixels to the device pages/rows/columns in a logical order. So byte 0 of the buffer represents the first 8 pixels on the first row (top) of the display, and continues until it wraps at the end of the row

The lcd buffer stores pixel data 'sideways' 0[4] = byte 0, bit 4

lcd ram:

0[0] 1[0] 2[0] 3[0] 4[0] 5[0] 6[0] 7[0] ….. 0[1] 1[1] 2[1] 3[1] 4[1] 5[1] 6[1] 7[1] ….. 0[2] 1[2] 2[2] 3[2] 4[2] 5[2] 6[2] 7[2] ….. 0[3] 1[3] 2[3] 3[3] 4[3] 5[3] 6[3] 7[3] ….. 0[4] 1[4] 2[4] 3[4] 4[4] 5[4] 6[4] 7[4] ….. 0[5] 1[5] 2[5] 3[5] 4[5] 5[5] 6[5] 7[5] ….. 0[6] 1[6] 2[6] 3[6] 4[6] 5[6] 6[6] 7[6] ….. 0[7] 1[7] 2[7] 3[7] 4[7] 5[7] 6[7] 7[7] …..

local buffer:

0[7] 0[6] 0[5] 0[4] 0[3] 0[2] 0[1] 0[0] , 1[7] 1[6] 1[5] 1[4] 1[3] 1[2] 1[1] 1[0] , 2[7] 2[6] 2[5] 2[4] 2[3] 2[2] 2[1] 2[0] , 3[7] 3[6] 3[5] 3[4] 3[3] 3[2] 3[1] 3[0] , 4[7] 4[6] 4[5] 4[4] 4[3] 4[2] 4[1] 4[0] , 5[7] 5[6] 5[5] 5[4] 5[3] 5[2] 5[1] 5[0] , 6[7] 6[6] 6[5] 6[4] 6[3] 6[2] 6[1] 6[0] , 7[7] 7[6] 7[5] 7[4] 7[3] 7[2] 7[1] 7[0] ,

So before writing to the device, we take a 'block' which is 8 pixels by 8 pixels, and rotate it to match the lcd ram

Tri-Color E-ink display

Datasheet: https://www.waveshare.com/w/upload/9/9e/1.54inch-e-paper-b-specification.pdf

Requires: Modules/Utilities/GFX/MonoGfx

driver for 1.5" tri-color e-ink display

5.3.3 IO

opex

- Generated with MrT Device Utility
- Bus: I2C, SPI
- RegMap: Register Map
- Datasheet: https://www.st.com/resou...
- DigiKey: 497-18052-2-ND
- I2C Address: 0x42

Description

Driver for MCU running custom GPIO expander firmware

Updating Registers

If changes are made to the device.yml file, the code can be updated using *mrtutils*

```
mrt-device -i doc/device.yml -o .
```

Usage

Configure GPIO

```
opex_t exp;

io_init_i2c(&exp, I2C1);            // Initialize expander on I2C1

io_gpio_cfg_t cfg;

cfg.mDIR = IO_GPIO_X_CFG_DIR_OUT;

io_cfg_gpio(&exp, 0, &cfg);         // Configure GPIO 0 to be an output

cfg.mDIR = IO_GPIO_X_CFG_DIR_IN;
cfg.mPP = 1;
cfg.mIRQ = IO_GPIO_X_CFG_IRQ_FALLING

io_cfg_gpio(&exp, 1, &cfg);         // Configure GPIO 1 to be an input with PUSH/Pull ON,␣
↪and a falling trigger for IRQ

io_set_gpio(&exp, 1, LOW);          // Sets GPIO output to LOW. Since it is configured as␣
↪an input, this enables the internal pulldown resistor
```

Set GPIO

```
io_set_gpio(&exp, 0, HIGH);       // Sets GPIO 0 High
```

Configure IRQ

```
io_cfg_irq(&exp, IO_IRQ_POLAR_LOW, 12)                                    //Configure IRQ to␣
↪pull GPIO 12 low when triggered
```

Register Map

Name	Address	Type	Access	Default	Description
GPIO_IN	0x00	uint32	R	0x00000000	Input values for gpio 0-25
GPIO_OUT	0x04	uint32	RW	0x00000000	Output values for gpio 0-15
GPIO_DDR	0x08	uint32	R	0x00000000	Direction Register for GPIO
IRQ_SRC	0x0C	uint32	R	0x00000000	latching Interrupt source mask. indicates souce of IRQ resets on
ADC_0_VAL	0x10	uint16	R	0x0000	Output of ADC 0
ADC_1_VAL	0x12	uint16	R	0x0000	Output of ADC 1
ADC_2_VAL	0x14	uint16	R	0x0000	Output of ADC 2
ADC_3_VAL	0x16	uint16	R	0x0000	Output of ADC 3
ADC_4_VAL	0x18	uint16	R	0x0000	Output of ADC 4
PWM_0_VAL	0x1A	uint16	W	0x0000	PWM value for ch 0
PWM_1_VAL	0x1C	uint16	W	0x0000	PWM value for ch 1
PWM_2_VAL	0x1E	uint16	W	0x0000	PWM value for ch 2
PWM_3_VAL	0x20	uint16	W	0x0000	PWM value for ch 3
PWM_4_VAL	0x22	uint16	W	0x0000	PWM value for ch 4
PWM_5_VAL	0x24	uint16	W	0x0000	PWM value for ch 5
GPIO_0_CFG	0x26	uint8	RW	0x00	Configuration for GPIO 0
GPIO_1_CFG	0x27	uint8	RW	0x00	Configuration for GPIO 1
GPIO_2_CFG	0x28	uint8	RW	0x00	Configuration for GPIO 2
GPIO_3_CFG	0x29	uint8	RW	0x00	Configuration for GPIO 3
GPIO_4_CFG	0x2A	uint8	RW	0x00	Configuration for GPIO 4
GPIO_5_CFG	0x2B	uint8	RW	0x00	Configuration for GPIO 5
GPIO_6_CFG	0x2C	uint8	RW	0x00	Configuration for GPIO 6
GPIO_7_CFG	0x2D	uint8	RW	0x00	Configuration for GPIO 7
GPIO_8_CFG	0x2E	uint8	RW	0x00	Configuration for GPIO 8
GPIO_9_CFG	0x2F	uint8	RW	0x00	Configuration for GPIO 9
GPIO_10_CFG	0x30	uint8	RW	0x00	Configuration for GPIO 10
GPIO_11_CFG	0x31	uint8	RW	0x00	Configuration for GPIO 11
GPIO_12_CFG	0x32	uint8	RW	0x00	Configuration for GPIO 12
GPIO_13_CFG	0x33	uint8	RW	0x00	Configuration for GPIO 13
GPIO_14_CFG	0x34	uint8	RW	0x00	Configuration for GPIO 14
GPIO_15_CFG	0x35	uint8	RW	0x00	Configuration for GPIO 15
GPIO_16_CFG	0x36	uint8	RW	0x00	Configuration for GPIO 16
GPIO_17_CFG	0x37	uint8	RW	0x00	Configuration for GPIO 17
GPIO_18_CFG	0x38	uint8	RW	0x00	Configuration for GPIO 18
GPIO_19_CFG	0x39	uint8	RW	0x00	Configuration for GPIO 19

Table 1 – continued from previous page

Name	Address	Type	Access	Default	Description
GPIO_20_CFG	0x3A	uint8	RW	0x00	Configuration for GPIO 20
GPIO_21_CFG	0x3B	uint8	RW	0x00	Configuration for GPIO 21
GPIO_22_CFG	0x3C	uint8	RW	0x00	Configuration for GPIO 22
GPIO_23_CFG	0x3D	uint8	RW	0x00	Configuration for GPIO 23
GPIO_24_CFG	0x3E	uint8	RW	0x00	Configuration for GPIO 24
GPIO_25_CFG	0x3F	uint8	RW	0x00	Configuration for GPIO 25
IRQ_CFG	0x40	uint16	RW	0x0000	IRQ Configuration
ADC_0_CFG	0x42	uint16	RW	0x0000	Configuration for ADC 0
ADC_1_CFG	0x44	uint16	RW	0x0000	Configuration for ADC 1
ADC_2_CFG	0x46	uint16	RW	0x0000	Configuration for ADC 2
ADC_3_CFG	0x48	uint16	RW	0x0000	Configuration for ADC 3
ADC_4_CFG	0x4A	uint16	RW	0x0000	Configuration for ADC 4
PWM_CONFIG	0x4C	uint32	RW	0x00000000	Configuration for PWM
WHO_AM_I	0x50	uint8	RW	0xAB	Device ID
VERSION	0x51	uint32	RW	0x00000000	Version of firmware
EEPROM_MEM	0x70	uint8	RW	0x00	Start address of EEPROM memory on stm8. User can read/write

Registers

GPIO_IN

Address
[0x00]

Input values for gpio 0-25

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	GPIO_IN																															

GPIO_OUT

Address
[0x04]

Output values for gpio 0-15

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	GPIO_OUT																															

GPIO_DDR

Address
[0x08]

Direction Register for GPIO

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	GPIO_DDR																															

IRQ_SRC

Address
[0x0C]

latching Interrupt source mask. indicates souce of IRQ resets on read

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	IRQ_SRC																															

Fields

IRQ_SRC
Source of IRQ

Name	Value	Description
GPIO_0	x01	IRQ triggered by GPIO0
ADC_0	x4000000	IRQ triggered by ADC0
ADC_1	x8000000	IRQ triggered by ADC1
ADC_2	x10000000	IRQ triggered by ADC2
ADC_3	x20000000	IRQ triggered by ADC3
ADC_4	x40000000	IRQ triggered by ADC4

ADC_n_VAL

Address
[—-]

Output of ADC n

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	ADC_0_VAL															

PWM_n_VAL

Address
[—-]

PWM value for ch n

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	PWM_0_VAL															

GPIO_n_CFG

Address
[—-]

Configuration for GPIO n

Bit	7	6	5	4	3	2	1	0
Field	DIR	PP		LL	IRQ		ALT	EN

Flags

PP
Enables Push/Pull on output, and Pull-up on input

ALT
Indicates that GPIO is disabled because pin is being used for an alternate function (PWM, ADC, etc)

EN
Enables GPIO

Fields

DIR
Pin Direction

Name	Value	Description
IN	b0	GPIO is an input
OUT	b1	GPIO is an output

LL
Low Level

Name	Value	Description
LOW	b0	Low level output
HIGH	b1	

IRQ
Interrupt selection

Name	Value	Description
NONE	b00	No interrupt
RISING	b01	Trigger on Rising
FALLING	b10	Trigger on falling
ANY	b11	Trigger on any

IRQ_CFG

Address
[0x40]

IRQ Configuration

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	Enabled	Polarity										Output				

Flags

Enabled
Enables IRQ signal on selected GPIO

Fields

Polarity
Sets polarity of IRQ

Name	Value	Description
ACTIVE_HIGH	b1	GPIO is high when IRQ is pending
ACTIVE_LOW	b0	GPIO is low when IRQ is pending

Output
Selects the GPIO to use for IRQ

ADC_n_CFG

Address
[—-]

Configuration for ADC n

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	Treshold												IRQ			EN

Flags

EN
Enables ADC Channel

Fields

Treshold
IRQ threshold for ADC channel

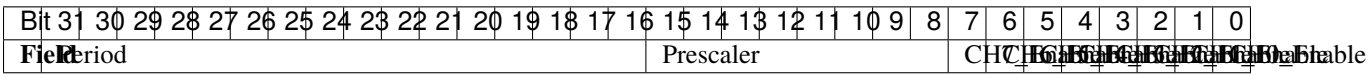
IRQ
Interrupt setting for ADC channel

Name	Value	Description
NONE	b00	No interrupt
RISING	b01	Trigger on Rising
FALLING	b10	Trigger on falling
ANY	b11	Trigger on any

PWM_CONFIG

Address
[0x4C]

Configuration for PWM



Flags

CH0_Enable
Enables PWM channel 0

CH1_Enable
Enables PWM channel 1

CH2_Enable
Enables PWM channel 2

CH3_Enable
Enables PWM channel 3

CH4_Enable
Enables PWM channel 4

CH5_Enable
Enables PWM channel 5

CH6_Enable
Enables PWM channel 6

CH7_Enable
Enables PWM channel 7

Fields

Period
Period for PWM signals

Prescaler
Prescaler for PWM, using 16Mhz clock

Name	Value	Description
PRESCALER_1	b0000	divide clock by 1 (16Mhz)
PRESCALER_2	b0001	divide clock by 2 (8Mhz)
PRESCALER_4	b0010	divide clock by 4 (4Mhz)
PRESCALER_8	b0011	divide clock by 8 (2Mhz)
PRESCALER_16	b0100	divide clock by 16 (1Mhz)
PRESCALER_32	b0101	divide clock by 32 (500Khz)
PRESCALER_64	b0110	divide clock by 64 (250Khz)
PRESCALER_128	b0111	divide clock by 128 (125Khz)
PRESCALER_256	b1000	divide clock by 256 (62.5 Khz)
PRESCALER_512	b1001	divide clock by 512 (31.25 Khz)
PRESCALER_1024	b1010	divide clock by 1024 (1.5625 KHz)
PRESCALER_2048	b1011	divide clock by 2048 ()
PRESCALER_4096	b1100	divide clock by 4096 ()
PRESCALER_8192	b1101	divide clock by 8192 ()
PRESCALER_16384	b1110	divide clock by 16384 ()
PRESCALER_32768	b1111	divide clock by 32768 ()

WHO_AM_I

Address
[0x50]

Default
[0xAB]

Device ID

Bit	7	6	5	4	3	2	1	0
Field								

Fields

ID
ID of device

Name	Value	Description
STM8S003F3	x70	20 pin variant
STM8S003K3	x71	32 pin variant

VERSION

Address
[0x51]

Version of firmware

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	MAJOR								MINOR								PATCH								BUILD							

Fields

MAJOR
Major Version

MINOR
Major Version

PATCH
Major Version

BUILD
Major Version

EEPROM_MEM

Address
[0x70]

Start address of EEPROM memory on stm8. User can read/write up to 128 bytes starting at this address

Bit	7	6	5	4	3	2	1	0
Field	EEPROM_MEM							

5.3.4 MotorDrivers

STSPIN220

- Generated with MrT Device Utility
- Bus: GPIO
- RegMap: Register Map
- Datasheet: https://www.st.com/resou...
- DigiKey: 497-16602-1-ND

Description

Low voltage stepper motor driver

Register Map

Name	Address	Type	Access	Default	Description

Registers

5.3.5 Biometric

ANV401 Fingerprint Sensor

This is the device driver for the ANV401 capacitive fingerprint sensor module.

Example Code

This example is based on an stm32 platform using huart1 for the device, and the irq and reset signals labeled as FINGER_EXTI and FINER_RST

```
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "Devices/Biometric/ANV401-FingerprintSensor/anv401.h"

/* Private variables ---------------------------------------------------------*/
anv401_t fpSensor;
volatile bool fpPresent = false;
volatile bool addNewUser = false;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{

    if(GPIO_Pin == FINGER_EXTI_Pin)
    {
```

(continues on next page)

(continued from previous page)

```
        fpPresent = true;
    }

    if(GPIO_Pin == NEW_USER_BUTTON_Pin)
    {
        addNewUser = true;
    }

}

int main(void)
{
    /* Initialization of HAL, UART, GPIO etc.. */

    //Initialize driver
    anv401_init(&fpSensor, MRT_GPIO(FINGER_EXTI), MRT_GPIO(FINGER_RST)));

    while(1)
    {
        if(addNewUser)
        {
            //Add whoever is touching the sensor as a new user with permission level 3
            anv401_add_user(&fpSensor, 3);

            addNewUser = false;
        }

        if(fpPresent)
        {
            anv401_user_t user = anv401_compare_fingerprint(&fpSensor);

            if(user.mId == ANV401_USER_NONE)
            {
                printf("No Matching User found, Access denied");
            }
            else
            {
                printf("User identified\nId: %04X\nPerm: %d", user.mId, user.mPerm);
            }

            fpPresent = false;
        }
    }

}
```

When the NEW_USER_BUTTON is pressed, the finger currently touching the sensor would be added as a new user. Whenever a finger touches the sensor, it will toggle the EXTI/IRQ signal, and then we can look for a match

5.3.6 Sensors

sht31

- Generated with MrT Device Utility
- Bus: I2C
- RegMap: Register Map
- Datasheet: https://media.digikey.co…
- DigiKey: 1649-1011-1-ND
- I2C Address: 0x44

Description

description

Register Map

Name	Address	Type	Access	Default	Description

Registers

LIS2DH12

- Generated with MrT Device Utility
- Bus: I2C,SPI
- RegMap: Register Map
- Datasheet: http://www.st.com/conten…
- DigiKey: 497-14851-1-ND
- I2C Address: 0x32

Description

MEMS Digital Output Motion Sensor Ultra Low-Power High Performance 3-Axis “Femto” Accelerometer

Register Map

Name	Address	Type	Access	Default	Description
STATUS_AUX	0x07	uint8	R	0x00	n/a
OUT_TEMP	0x0C	uint16	R	0x0000	Temperature sensor data
WHO_AM_I	0x0F	uint8	R	0x33	Device identification register
CTRL0	0x1E	uint8	RW	0x10	Control Register 0
TEMP_CFG	0x1F	uint8	RW	0x07	n/a
CTRL1	0x20	uint8	RW	0x07	Control Register 1
CTRL2	0x21	uint8	RW	0x00	Control Register 2
CTRL3	0x22	uint8	RW	0x00	Control Register 3
CTRL4	0x23	uint8	RW	0x00	Control Register 4
CTRL5	0x24	uint8	RW	0x00	Control Register 5
CTRL6	0x25	uint8	RW	0x00	Control Register 6
REFERENCE	0x26	uint8	RW	0x00	Reference value for interrupt generation
STATUS	0x27	uint8	R	0x00	n/a
OUT_X	0x28	uint16	R	0x0000	X-axis acceleration data
OUT_Y	0x2A	uint16	R	0x0000	Y-axis acceleration data
OUT_Z	0x2C	uint16	R	0x0000	Z-axis acceleration data
FIFO_CTRL	0x2E	uint8	RW	0x00	Fifo Control register
FIFO_SRC	0x2F	uint8	R	0x00	Fifo status register
INT1_CFG	0x30	uint8	RW	0x00	Interrupt 1 config register
INT1_SRC	0x31	uint8	R	0x00	Interrupt 1 source register
INT1_THS	0x32	uint8	RW	0x00	Interrupt 1 threshold register
INT1_DURATION	0x33	uint8	RW	0x00	Interrupt 1 duration register
INT2_CFG	0x34	uint8	RW	0x00	Interrupt 2 config register
INT2_SRC	0x35	uint8	R	0x00	Interrupt 2 source register
INT2_THS	0x36	uint8	RW	0x00	Interrupt 2 threshold register
INT2_DURATION	0x37	uint8	RW	0x00	Interrupt 2 duration register
CLICK_CFG	0x38	uint8	RW	0x00	Click config
CLICK_SRC	0x39	uint8	R	0x00	Click source
CLICK_THS	0x3A	uint8	RW	0x00	Click Threshold
TIME_LIMIT	0x3B	uint8	RW	0x00	Click time limit
TIME_LATENCY	0x3C	uint8	RW	0x00	Click time latency
TIME_WINDOW	0x3D	uint8	RW	0x00	Click time window
ACT_THS	0x3E	uint8	RW	0x00	Activity threshold
ACT_DUR	0x3F	uint8	RW	0x00	Activity duration

Registers

STATUS_AUX

Address
[0x07]

n/a

Bit	7	6	5	4	3	2	1	0
Field	STATUS_AUX							

OUT_TEMP

Address
[0x0C]

Temperature sensor data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	OUT_TEMP															

WHO_AM_I

Address
[0x0F]

Default
[0x33]

Device identification register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

WHO_AM_I
Device identification register

CTRL0

Address
[0x1E]

Default
[0x10]

Control Register 0

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CTRL0
Control Register 0

TEMP_CFG

Address
[0x1F]

Default
[0x07]

n/a

Bit	7	6	5	4	3	2	1	0
Field								

Fields

TEMP_CFG
n/a

CTRL1

Address
[0x20]

Default
[0x07]

Control Register 1

Bit	7	6	5	4	3	2	1	0
Field				LOW_PWR	Z_EN	Y_EN	X_EN	

Flags

X_EN
X-axis enable

Y_EN
Y-axis enable

Z_EN
Z-axis enable

LOW_PWR
Low-power mode enable

Fields

ODR
Data rate selection

Name	Value	Descriptions
PWR_DWN	b0000	Power-down mode
1Hz	b0001	HR/ Normal / Low-power mode (1 Hz)
10Hz	b1000	HR/ Normal / Low-power mode (10 Hz)
25Hz	b1001	HR/ Normal / Low-power mode (25 Hz)
50Hz	b1000000	HR/ Normal / Low-power mode (50 Hz)
100Hz	b1000001	HR/ Normal / Low-power mode (100 Hz)
200Hz	b1001000	HR/ Normal / Low-power mode (200 Hz)
400Hz	b1001001	HR/ Normal / Low-power mode (400 Hz)
1620Hz	b0111	Low-power mode (1.620 kHz)
5376Hz	b0111	HR/ Normal (1.344 kHz) / Low-power mode (5.376 kHz)

CTRL2

Address
[0x21]

Default
[0x00]

Control Register 2

Bit	7	6	5	4	3	2	1	0
Field				FDS	HPCLICK	HP_IA2	HP_IA1	

Flags

HP_IA1
High-pass filter enabled for AOI function on Interrupt 1

HP_IA2
High-pass filter enabled for AOI function on Interrupt 2

HPCLICK
High-pass filter enabled for Click function

FDS
Filtered data selection

CTRL3

Address
[0x22]

Default
[0x00]

Control Register 3

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CTRL3
Control Register 3

CTRL4

Address
[0x23]

Default
[0x00]

Control Register 4

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CTRL4
Control Register 4

CTRL5

Address
[0x24]

Default
[0x00]

Control Register 5

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CTRL5
Control Register 5

CTRL6

Address
[0x25]

Default
[0x00]

Control Register 6

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CTRL6
Control Register 6

REFERENCE

Address
[0x26]

Default
[0x00]

Reference value for interrupt generation

Bit	7	6	5	4	3	2	1	0
Field								

Fields

REFERENCE
Reference value for interrupt generation

STATUS

Address
[0x27]

n/a

Bit	7	6	5	4	3	2	1	0
Field	STATUS							

OUT_X

Address
[0x28]

X-axis acceleration data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	OUT_X															

OUT_Y

Address
[0x2A]

Y-axis acceleration data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	OUT_Y															

OUT_Z

Address
[0x2C]

Z-axis acceleration data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	OUT_Z															

FIFO_CTRL

Address
[0x2E]

Default
[0x00]

Fifo Control register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

FIFO_CTRL
Fifo Control register

FIFO_SRC

Address
[0x2F]

Fifo status register

Bit	7	6	5	4	3	2	1	0
Field	FIFO_SRC							

INT1_CFG

Address
[0x30]

Default
[0x00]

Interrupt 1 config register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

INT1_CFG
Interrupt 1 config register

INT1_SRC

Address
[0x31]

Interrupt 1 source register

Bit	7	6	5	4	3	2	1	0
Field	INT1_SRC							

INT1_THS

Address
[0x32]

Default
[0x00]

Interrupt 1 threshold register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

INT1_THS
Interrupt 1 threshold register

INT1_DURATION

Address
[0x33]

Default
[0x00]

Interrupt 1 duration register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

INT1_DURATION
Interrupt 1 duration register

INT2_CFG

Address
[0x34]

Default
[0x00]

Interrupt 2 config register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

INT2_CFG
Interrupt 2 config register

INT2_SRC

Address
[0x35]

Interrupt 2 source register

Bit	7	6	5	4	3	2	1	0
Field	INT2_SRC							

INT2_THS

Address
[0x36]

Default
[0x00]

Interrupt 2 threshold register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

INT2_THS
Interrupt 2 threshold register

INT2_DURATION

Address
[0x37]

Default
[0x00]

Interrupt 2 duration register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

INT2_DURATION
Interrupt 2 duration register

CLICK_CFG

Address
[0x38]

Default
[0x00]

Click config

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CLICK_CFG
Click config

CLICK_SRC

Address
[0x39]

Click source

Bit	7	6	5	4	3	2	1	0
Field	CLICK_SRC							

CLICK_THS

Address
[0x3A]

Default
[0x00]

Click Threshold

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CLICK_THS
Click Threshold

TIME_LIMIT

Address
[0x3B]

Default
[0x00]

Click time limit

Bit	7	6	5	4	3	2	1	0
Field								

Fields

TIME_LIMIT
Click time limit

TIME_LATENCY

Address
[0x3C]

Default
[0x00]

Click time latency

Bit	7	6	5	4	3	2	1	0
Field								

Fields

TIME_LATENCY
Click time latency

TIME_WINDOW

Address
[0x3D]

Default
[0x00]

Click time window

Bit	7	6	5	4	3	2	1	0
Field								

Fields

TIME_WINDOW
Click time window

ACT_THS

Address
[0x3E]

Default
[0x00]

Activity threshold

Bit	7	6	5	4	3	2	1	0
Field								

Fields

ACT_THS
Activity threshold

ACT_DUR

Address
[0x3F]

Default
[0x00]

Activity duration

Bit	7	6	5	4	3	2	1	0
Field								

Fields

ACT_DUR
Activity duration

HTS221

- Generated with MrT Device Utility
- Bus: I2C
- RegMap: Register Map
- Datasheet: https://www.st.com/conte…
- DigiKey: 497-15382-1-ND
- I2C Address: 0xBE

Description

Humidity and Temperature Sensor

Register Map

Name	Address	Type	Access	Default	Description
WHO_AM_I	0x0F	uint8	R	0xBC	Id Register
AV_CONF	0x10	uint8	RW	0x1B	Humidity and temperature resolution mode
CTRL1	0x20	uint8	RW	0x00	Control register 1
CTRL2	0x21	uint8	RW	0x00	Control register 2
CTRL3	0x22	uint8	RW	0x00	Control register 3
STATUS	0x27	uint8	R	0x00	Status register
HUMIDITY_OUT	0x28	int16	R	0x0000	Relative humidity data
TEMP_OUT	0x2A	int16	R	0x0000	Temperature data
H0_rH_x2	0x30	uint8	R	0x00	Calibration data
H1_rH_x2	0x31	uint8	R	0x00	Calibration data
T0_DEGC_x8	0x32	uint8	R	0x00	Calibration data
T1_DEGC_x8	0x33	uint8	R	0x00	Calibration data
T1T0_MSB	0x35	uint8	R	0x00	Calibration data
H0_T0_OUT	0x36	int16	R	0x0000	Calibration data
H1_T0_OUT	0x3A	int16	R	0x0000	Calibration data
T0_OUT	0x3C	int16	R	0x0000	Calibration data
T1_OUT	0x3E	int16	R	0x0000	Calibration data

Registers

WHO_AM_I

Address
[0x0F]

Default
[0xBC]

Id Register

Bit	7	6	5	4	3	2	1	0
Field								

Fields

WHO_AM_I
Id Register

AV_CONF

Address
[0x10]

Default
[0x1B]

Humidity and temperature resolution mode

Bit	7	6	5	4	3	2	1	0
Field		AVGT			AVGH			

Fields

AVGH
Selects the number of Humidity samples to average for data output

Name	Value	Descriptions
4	b000	4 samples
8	b001	8 samples
16	b010	16 samples
32	b011	32 samples
64	b100	64 samples
128	b101	128 samples
256	b110	256 samples
512	b111	512 samples

AVGT
Selects the number of Temperature samples to average for data output

Name	Value	Descriptions
2	b000	2 samples
4	b001	4 samples
8	b010	8 samples
16	b011	16 samples
32	b100	32 samples
64	b101	64 samples
128	b110	128 samples
256	b111	256 samples

CTRL1

Address
[0x20]

Default
[0x00]

Control register 1

Bit	7	6	5	4	3	2	1	0
Field					BDU	ODR		

Flags

PD
power down mode

BDU
Block Data update. Prevents update until LSB of data is read

Fields

ODR
Selects the Output rate for the sensor data

Name	Value	Descriptions
ONESHOT	b00	readings must be requested
1HZ	b01	1 hz sampling
7HZ	b10	7 hz sampling
12_5HZ	b11	12.5 hz sampling

CTRL2

Address
[0x21]

Default
[0x00]

Control register 2

Bit	7	6	5	4	3	2	1	0
Field						HEATER	ONESHOT	

Flags

BOOT
Reboot memory content

HEATER
Enable intenal heating element

ONESHOT
Start conversion for new data

CTRL3

Address
[0x22]

Default
[0x00]

Control register 3

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CTRL3
Control register 3

STATUS

Address
[0x27]

Default
[0x00]

Status register

Bit	7	6	5	4	3	2	1	0
Field						HUM_READY	TEMP_READY	

Flags

TEMP_READY
indicates that a temperature reading is ready

HUM_READY
indicates that a humidity reading is ready

HUMIDITY_OUT

Address
[0x28]

Relative humidity data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

HUM_OUT
Current ADC reading for humidity sensor

TEMP_OUT

Address
[0x2A]

Temperature data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

TEMP_OUT
Current ADC reading for temperature sensor

H0_rH_x2

Address
[0x30]

Calibration data

Bit	7	6	5	4	3	2	1	0
Field	H0_rH_x2							

H1_rH_x2

Address
[0x31]

Calibration data

Bit	7	6	5	4	3	2	1	0
Field	H1_rH_x2							

T0_DEGC_x8

Address
[0x32]

Calibration data

Bit	7	6	5	4	3	2	1	0
Field	T0_DEGC_x8							

T1_DEGC_x8

Address
[0x33]

Calibration data

Bit	7	6	5	4	3	2	1	0
Field	T1_DEGC_x8							

T1T0_MSB

Address
[0x35]

Calibration data

Bit	7	6	5	4	3	2	1	0
Field	T1T0_MSB							

H0_T0_OUT

Address
[0x36]

Calibration data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	H0_T0_OUT															

H1_T0_OUT

Address
[0x3A]

Calibration data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	H1_T0_OUT															

T0_OUT

Address
[0x3C]

Calibration data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	T0_OUT															

T1_OUT

Address
[0x3E]

Calibration data

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	T1_OUT															

Device-LSM6D

device driver for lsm6d iNEMO inertial module (Accelerometer and Gyroscope)

5.3.7 RF

device-nrf24

driver for nrf24 transceiver

5.3.8 Power

README

This README would normally document whatever steps are necessary to get your application up and running.

What is this repository for?

- Quick summary
- Version
- Learn Markdown

How do I get set up?

- Summary of set up
- Configuration
- Dependencies
- Database configuration
- How to run tests
- Deployment instructions

Contribution guidelines

- Writing tests
- Code review
- Other guidelines

Who do I talk to?

- Repo owner or admin
- Other community or team contact

stc3117

- Generated with MrT Device Utility
- Bus: I2C
- RegMap: Register Map
- Datasheet: https://www.st.com/conte…
- DigiKey: 497-15387-1-ND
- I2C Address: 0xE0

Description

Gas gauge IC with battery charger control

Register Map

Name	Address	Type	Access	Default	Description
MODE	0x00	uint8	RW	0x00	Mode register
CTRL	0x01	uint8	RW	0x00	Control and status register
SOC	0x02	uint16	RW	0x0000	Battery SOC (LSB = 1/512 %)
COUNTER	0x04	uint16	R	0x0000	Number of conversions
CURRENT	0x06	uint16	R	0x0000	Battery current
VOLTAGE	0x08	uint16	R	0x0000	Battery voltage (LSB = 2.2 mV)
TEMPERATURE	0x0A	uint8	R	0x00	Temperature in degrees C (LSB = 1deg C)
AVG_CURRENT	0x0B	uint16	RW	0x0000	Battery average current or SOC change rate
OCV	0x0D	uint16	RW	0x0000	OCV register (LSV = 0.55 mV)
CC_CNF	0x0F	uint16	RW	0x018B	Battery average current or SOC change rate
VM_CNF	0x11	uint16	RW	0x0141	Voltage gas gauge algorithm parameter
ALARM_SOC	0x13	uint8	RW	0x02	SOC alarm level in (LSB = 0.5%)
ALARM_VOLTAGE	0x14	uint8	RW	0xAA	Battery low voltage alarm level (LSB = 17.6 mV)
CURRENT_THRES	0x15	uint8	RW	0x0A	Current threshold for current monitoring (LSB = 47.04 uV)
CMONIT_COUNT	0x16	uint8	R	0x78	Current monitoring counter
CMONIT_MAX	0x17	uint8	RW	0x78	Maximum counter value for current monitoring
ID	0x18	uint8	R	0x16	Part type ID = 16h
CC_ADJ	0x1B	uint16	R	0x0000	Coulomb counter adjustment register
VM_ADJ	0x1D	uint16	R	0x0000	Voltage mode adjustment register

Registers

MODE

Address
[0x00]

Mode register

Bit	7	6	5	4	3	2	1	0
Field	FORCE_VM	FORCE_CC	GG_RUN	ALM_ENA	FORCE_CD	BIBATD_PU	VMODE	

Flags

VMODE
Power saving voltage mode

BIBATD_PU
BATD internal pull-up enable

FORCE_CD
Force CD output high

ALM_ENA
Enable Alarm function

GG_RUN
creates a flag at bit 1 of the DUMMY register

FORCE_CC
Force the relaxation timer to switch to the Coulomb counter (CC) state

FORCE_VM
Force the relaxation timer to switch to voltage mode (VM) state

---

CTRL

Address
[0x01]

Control and status register

Bit	7	6	5	4	3	2	1	0
Field	CTRL							

SOC

Address
[0x02]

Battery SOC (LSB = 1/512 %)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

SOC
Battery SOC (LSB = 1/512 %)

COUNTER

Address
[0x04]

Default
[0x0000]

Number of conversions

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

COUNTER
Number of conversions

CURRENT

Address
[0x06]

Default
[0x0000]

Battery current

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

CURRENT
Battery current

VOLTAGE

Address
[0x08]

Default
[0x0000]

Battery voltage (LSB = 2.2 mV)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

VOLTAGE
Battery voltage (LSB = 2.2 mV)

TEMPERATURE

Address
[0x0A]

Default
[0x00]

Temperature in degrees C (LSB = 1deg C)

Bit	7	6	5	4	3	2	1	0
Field								

Fields

TEMPERATURE
Temperature in degrees C (LSB = 1deg C)

AVG_CURRENT

Address
[0x0B]

Default
[0x0000]

Battery average current or SOC change rate

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AVG_CURRENT
Battery average current or SOC change rate

OCV

Address
[0x0D]

Default
[0x0000]

OCV register (LSV = 0.55 mV)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

OCV
OCV register (LSV = 0.55 mV)

CC_CNF

Address
[0x0F]

Default
[0x018B]

Battery average current or SOC change rate

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

CC_CNF
Battery average current or SOC change rate

VM_CNF

Address
[0x11]

Default
[0x0141]

Voltage gas gauge algorithm parameter

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

VM_CNF
Voltage gas gauge algorithm parameter

ALARM_SOC

Address
[0x13]

Default
[0x02]

SOC alarm level in (LSB = 0.5%)

Bit	7	6	5	4	3	2	1	0
Field								

Fields

ALARM_SOC
SOC alarm level in (LSB = 0.5%)

ALARM_VOLTAGE

Address
[0x14]

Default
[0xAA]

Battery low voltage alarm level (LSB = 17.6 mV)

Bit	7	6	5	4	3	2	1	0
Field								

Fields

ALARM_VOLTAGE
Battery low voltage alarm level (LSB = 17.6 mV)

CURRENT_THRES

Address
[0x15]

Default
[0x0A]

Current threshold for current monitoring (LSB = 47.04 uV)

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CURRENT_THRES
Current threshold for current monitoring (LSB = 47.04 uV)

CMONIT_COUNT

Address
[0x16]

Default
[0x78]

Current monitoring counter

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CMONIT_COUNT
Current monitoring counter

CMONIT_MAX

Address
[0x17]

Default
[0x78]

Maximum counter value for current monitoring

Bit	7	6	5	4	3	2	1	0
Field								

Fields

CMONIT_MAX
Maximum counter value for current monitoring

ID

Address
[0x18]

Default
[0x16]

Part type ID = 16h

Bit	7	6	5	4	3	2	1	0
Field								

Fields

ID
Part type ID = 16h

CC_ADJ

Address
[0x1B]

Default
[0x0000]

Coulomb counter adjustment register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

CC_ADJ
Coulomb counter adjustment register

VM_ADJ

Address
[0x1D]

Default
[0x0000]

Voltage mode adjustment register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

VM_ADJ
Voltage mode adjustment register

BQ28Z

- Generated with MrT Device Utility
- Bus: I2C
- RegMap: Register Map
- Datasheet: http://www.ti.com/lit/ds…
- DigiKey: 296-43394-1-ND
- I2C Address: 0xAA

Description

Battery Fuel Gauge

Register Map

Name	Address	Type	Access	Default	Description
DUMMY	0x00	uint16	R	0xDEAD	dummy register
ManufacturerAccess_ControlStatus	0x00	uint16	RW	0x0000	Control Register
AtRate	0x02	int16	RW	0x0000	Read/Write. The value is a signed integer with
AtRateTimeToEmpty	0x04	uint16	R	0x0000	This read-only function returns an unsigned int
Temperature	0x06	uint16	R	0x0000	This read-only function returns an unsigned int
Voltage	0x08	uint16	R	0x0000	This read-only function returns an unsigned int
BatteryStatus	0x0A	uint16	R	0x0000	See the Flags register.
Current	0x0C	int16	R	0x0000	This read-only function returns a signed integer
MaxError	0x0E	uint8	R	0x00	This read-word function returns the expected m
RemainingCapacity	0x10	uint16	R	0x0000	This read-only command returns the predicted
FullChargeCapacity	0x12	uint16	R	0x0000	This read-only command returns the predicted
AverageCurrent	0x14	int16	R	0x0000	This read-only function returns a signed integer

Name	Address	Type	Access	Default	Description
AverageTimeToEmpty	0x16	uint16	R	0x0000	Uses average current value with a time constan
AverageTimeToFull	0x18	uint16	R	0x0000	This read-only function returns a unsigned inte
StandbyCurrent	0x1A	int16	R	0x0000	This read-only function returns a signed integer
StandbyTimeToEmpty	0x1C	uint16	R	0x0000	This read-only function returns a unsigned inte
MaxLoadCurrent	0x1E	int16	R	0x0000	This read-only function returns a signed integer
MaxLoadTimeToEmpty	0x20	uint16	R	0x0000	This read-only function returns a unsigned inte
AveragePower	0x22	int16	R	0x0000	This read-only function returns a signed integer
BTPDischargeSet	0x24	int16	RW	0x0000	This command sets the OperationStatusA BTP
BTPChargeSet	0x26	int16	RW	0x0000	This command clears the OperationStatusA BT
InternalTemperature	0x28	uint16	R	0x0000	This read-only function returns an unsigned int
CycleCount	0x2A	uint16	R	0x0000	This read-only function returns an unsigned int
RelativeStateOfCharge	0x2C	uint8	R	0x00	This read-only function returns an unsigned int
StateOfHealth	0x2E	uint8	R	0x00	This read-only function returns an unsigned int
ChargeVoltage	0x30	uint16	R	0x0000	Returns the desired charging voltage in mV to t
ChargeCurrent	0x32	uint16	R	0x0000	Returns the desired charging current in mA to t
DesignCapacity	0x3C	uint16	R	0x0000	In SEALED and UNSEALED access This com
AltManufacturerAccess	0x3E	uint16	R	0x0000	MAC Data block command
MACData	0x40	uint16	R	0x0000	MAC Data block
SafetyAlert	0x50	uint32	R	0x00000000	This command returns the SafetyAlert flags on
MACDataSum	0x60	uint8	R	0x00	MAC Data block checksum
MACDataLen	0x61	uint8	R	0x00	MAC Data block length

Registers

DUMMY

Address
[0x00]

Default
[0xDEAD]

dummy register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field														BIT1	BIT0	

Flags

BIT0
creates a flag at bit 0 of the DUMMY register

BIT1
creates a flag at bit 1 of the DUMMY register

Fields

REMAINING
creates a 14 bit field using the remaing bits

Name	Address	Description
MIN	x00	creates a macro for the minimum 14 bit value
MAX	x3fff	creates a macro for the maximum 14 bit value

ManufacturerAccess_ControlStatus

Address
[0x00]

Default
[0x0000]

Control Register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	SECURITY_Mode		AUTHCALM			CheckSumValid		BTP_INT				LDMD	R_DIS	VOK	QMax	

Flags

AUTHCALM
Automatic Calibration Mode

CheckSumValid
Checksum Valid

BTP_INT
Battery Trip Point Interrupt. Setting and clearing this bit depends on various conditions

LDMD
LOAD Mode

R_DIS
Resistance Updates

VOK
Voltage OK for QMax Update

QMax
QMax Updates. This bit toggles after every QMax update.

Fields

SECURITY_Mode
Security Mode

Name	Address	Description
Reserved	b00	Reserved
Full_Access	b01	Full Access
Unsealed	b10	Unsealed
Sealed	b11	Sealed

AtRate

Address
[0x02]

Default
[0x0000]

Read/Write. The value is a signed integer with the negative value indicating a discharge current value. The default value is 0 and forces AtRateTimeToEmpty() to return 65535.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AtRate
Read/Write. The value is a signed integer with the negative value indicating a discharge current value. The default value is 0 and forces AtRateTimeToEmpty() to return 65535.

AtRateTimeToEmpty

Address
[0x04]

Default
[0x0000]

This read-only function returns an unsigned integer value to predict remaining operating time based on battery discharge at the AtRate() value in minutes with a range of 0 to 65534. A value of 65535 indicates AtRate() = 0. The gas gauge updates the AtRateTimeToEmpty() within 1 s after the system sets the AtRate() value. The gas gauge updates these parameters every 1 s. The commands are used in NORMAL mode.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AtRateTimeToEmpty
This read-only function returns an unsigned integer value to predict remaining operating time based on battery discharge at the AtRate() value in minutes with a range of 0 to 65534. A value of 65535 indicates AtRate() = 0. The gas gauge updates the AtRateTimeToEmpty() within 1 s after the system sets the AtRate() value. The gas gauge updates these parameters every 1 s. The commands are used in NORMAL mode.

Temperature

Address
[0x06]

Default
[0x0000]

This read-only function returns an unsigned integer value of temperature in units (0.1 k) measured by the gas gauge and is used for the gauging algorithm. It reports either InternalTemperature() or external thermistor temperature depending on the setting of the TEMPS bit in Pack configuration.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

Temperature
This read-only function returns an unsigned integer value of temperature in units (0.1 k) measured by the gas gauge and is used for the gauging algorithm. It reports either InternalTemperature() or external thermistor temperature depending on the setting of the TEMPS bit in Pack configuration.

Voltage

Address
[0x08]

Default
[0x0000]

This read-only function returns an unsigned integer value of the measured cell pack in mV with a range of 0 12000 mV.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

Voltage
This read-only function returns an unsigned integer value of the measured cell pack in mV with a range of 0 12000 mV.

BatteryStatus

Address
[0x0A]

Default
[0x0000]

See the Flags register.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	TCA		OTA	TDA		RCA	RTA	INIT	DSG	FC	FD	Error_Code				

Flags

FD
Fully Discharged

FC
Fully Charged

DSG
Discharging

INIT
Initialization

RTA
Remaining Time Alarm

RCA
Remaining Capacity Alarm

TDA
Terminate Discharge Alarm

OTA
Overtemperature Alarm

TCA
Terminate Charge Alarm

OCA
Overcharged Alarm

Fields

Error_Code
Error Code

Name	Address	Description
OK	b0000	OK
Busy	b0001	Busy
Reserved_Command	b0010	Reserved_Command
Unsupported_Command	b0011	Unsupported_Command
AccessDenied	b0100	AccessDenied
Overflow_Underflow	b0101	Overflow_Underflow
BadSize	b0110	BadSize
UnknownError	b0111	UnknownError

Current

Address
[0x0C]

Default
[0x0000]

This read-only function returns a signed integer value that is the instantaneous current flow through the sense resistor. The value is updated every 1 s. Units are mA.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

Current
This read-only function returns a signed integer value that is the instantaneous current flow through the sense resistor. The value is updated every 1 s. Units are mA.

MaxError

Address
[0x0E]

Default
[0x00]

This read-word function returns the expected margin of error

Bit	7	6	5	4	3	2	1	0
Field								

Fields

MaxError
This read-word function returns the expected margin of error

RemainingCapacity

Address
[0x10]

Default
[0x0000]

This read-only command returns the predicted remaining capacity based on rate (per configured Load Select) temperature present depth-of-discharge and stored impedance. Values are reported in mAh.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

RemainingCapacity
This read-only command returns the predicted remaining capacity based on rate (per configured Load Select) temperature present depth-of-discharge and stored impedance. Values are reported in mAh.

FullChargeCapacity

Address
[0x12]

Default
[0x0000]

This read-only command returns the predicted capacity of the battery at full charge based on rate (per configured Load Select) temperature present depth-of-discharge and stored impedance. Values are reported in mAh.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

FullChargeCapacity
This read-only command returns the predicted capacity of the battery at full charge based on rate (per configured Load Select) temperature present depth-of-discharge and stored impedance. Values are reported in mAh.

AverageCurrent

Address
[0x14]

Default
[0x0000]

This read-only function returns a signed integer value that is the average current flow through the sense resistor. The value is updated every 1 s. Units are mA.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AverageCurrent
This read-only function returns a signed integer value that is the average current flow through the sense resistor. The value is updated every 1 s. Units are mA.

AverageTimeToEmpty

Address
[0x16]

Default
[0x0000]

Uses average current value with a time constant of 15 s for this method. A value of 65535 means the battery is not being discharged.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AverageTimeToEmpty
Uses average current value with a time constant of 15 s for this method. A value of 65535 means the battery is not being discharged.

AverageTimeToFull

Address
[0x18]

Default
[0x0000]

This read-only function returns a unsigned integer value predicting time to reach full charge for the battery in units of minutes based on AverageCurrent(). The computation accounts for the taper current time extension from linear TTF computation based on a fixed AverageCurrent() rate of charge accumulation. A value of 65535 indicates the battery is not being charged.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AverageTimeToFull
This read-only function returns a unsigned integer value predicting time to reach full charge for the battery in units of minutes based on AverageCurrent(). The computation accounts for the taper current time extension from linear TTF computation based on a fixed AverageCurrent() rate of charge accumulation. A value of 65535 indicates the battery is not being charged.

StandbyCurrent

Address
[0x1A]

Default
[0x0000]

This read-only function returns a signed integer value of measured standby current through the sense resistor. The StandbyCurrent() is an adaptive measurement. Initially it will report the standby current programmed in initial standby and after several seconds in standby mode will report the measured standby. The register value is updated every 1 s when measured current is above the deadband and is less than or equal to 2 × initial standby. The first and last values that meet these criteria are not averaged in since they may not be stable values. To approximate to a 1-min time constant each new value of StandbyCurrent() is computed by taking approximate 93% weight of the last standby current and approximate 7% of the current measured average current.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

StandbyCurrent
This read-only function returns a signed integer value of measured standby current through the sense resistor. The StandbyCurrent() is an adaptive measurement. Initially it will report the standby current programmed in initial standby and after several seconds in standby mode will report the measured standby. The register value is updated every 1 s when measured current is above the deadband and is less than or equal to 2 × initial standby. The first and last values that meet these criteria are not averaged in since they may not be stable values. To approximate to a 1-min time constant each new value of StandbyCurrent() is computed by taking approximate 93% weight of the last standby current and approximate 7% of the current measured average current.

StandbyTimeToEmpty

Address
[0x1C]

Default
[0x0000]

This read-only function returns a unsigned integer value predicting remaining battery life at standby rate of discharge in units of minutes. The computation uses Nominal Available Capacity (NAC) for the calculation. A value of 65535 indicates the battery is not being discharged.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

StandbyTimeToEmpty
This read-only function returns a unsigned integer value predicting remaining battery life at standby rate of discharge in units of minutes. The computation uses Nominal Available Capacity (NAC) for the calculation. A value of 65535 indicates the battery is not being discharged.

MaxLoadCurrent

Address
[0x1E]

Default
[0x0000]

This read-only function returns a signed integer value in units of mA of maximum load conditions. The MaxLoadCurrent() is an adaptive measurement which is initially reported as the maximum load current programmed in initial Max Load Current register. If the measured current is ever greater than the initial Max Load Current then the MaxLoadCurrent() updates to the new current. MaxLoadCurrent() is reduced to the average of the previous value and initial Max Load Current whenever the battery is charged to full after a previous discharge to an SOC of less than 50%. This will prevent the reported value from maintaining an unusually high value.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

MaxLoadCurrent
This read-only function returns a signed integer value in units of mA of maximum load conditions. The MaxLoadCurrent() is an adaptive measurement which is initially reported as the maximum load current programmed in initial Max Load Current register. If the measured current is ever greater than the initial Max Load Current then the MaxLoadCurrent() updates to the new current. MaxLoadCurrent() is reduced to the average of the previous value and initial Max Load Current whenever the battery is charged to full after a previous discharge to an SOC of less than 50%. This will prevent the reported value from maintaining an unusually high value.

MaxLoadTimeToEmpty

Address
[0x20]

Default
[0x0000]

This read-only function returns a unsigned integer value predicting remaining battery life at the maximum discharge load current rate in units of minutes. A value of 65535 indicates that the battery is not being discharged.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

MaxLoadTimeToEmpty
This read-only function returns a unsigned integer value predicting remaining battery life at the maximum discharge load current rate in units of minutes. A value of 65535 indicates that the battery is not being discharged.

AveragePower

Address
[0x22]

Default
[0x0000]

This read-only function returns a signed integer value of average power during battery charging and discharging. It is negative during discharge and positive during charge. A value of 0 indicates that the battery is not being discharged. The value is reported in units of mW.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AveragePower
This read-only function returns a signed integer value of average power during battery charging and discharging. It is negative during discharge and positive during charge. A value of 0 indicates that the battery is not being discharged. The value is reported in units of mW.

BTPDischargeSet

Address
[0x24]

Default
[0x0000]

This command sets the OperationStatusA BTP_INT and the BTP_INT pin will be asserted when the RemCap drops below the set threshold in DF register.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

BTPDischargeSet
This command sets the OperationStatusA BTP_INT and the BTP_INT pin will be asserted when the RemCap drops below the set threshold in DF register.

BTPChargeSet

Address
[0x26]

Default
[0x0000]

This command clears the OperationStatusA BTP_INT and the BTP_INT pin will be deasserted.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

BTPChargeSet
This command clears the OperationStatusA BTP_INT and the BTP_INT pin will be deasserted.

InternalTemperature

Address
[0x28]

Default
[0x0000]

This read-only function returns an unsigned integer value of the measured internal temperature of the device in 0.1-k units measured by the gas gauge.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

InternalTemperature
This read-only function returns an unsigned integer value of the measured internal temperature of the device in 0.1-k units measured by the gas gauge.

CycleCount

Address
[0x2A]

Default
[0x0000]

This read-only function returns an unsigned integer value of the number of cycles the battery has experienced a discharge (range 0 to 65535). One cycle occurs when accumulated discharge greater than or equal to CC threshold.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

CycleCount
This read-only function returns an unsigned integer value of the number of cycles the battery has experienced a discharge (range 0 to 65535). One cycle occurs when accumulated discharge greater than or equal to CC threshold.

RelativeStateOfCharge

Address
[0x2C]

Default
[0x00]

This read-only function returns an unsigned integer value of the predicted remaining battery capacity expressed as percentage of FullChargeCapacity() with a range of 0% to 100%.

Bit	7	6	5	4	3	2	1	0
Field								

Fields

RelativeStateOfCharge
This read-only function returns an unsigned integer value of the predicted remaining battery capacity expressed as percentage of FullChargeCapacity() with a range of 0% to 100%.

StateOfHealth

Address
[0x2E]

Default
[0x00]

This read-only function returns an unsigned integer value expressed as a percentage of the ratio of predicted FCC (25C SoH Load Rate) over the DesignCapacity(). The range is 0x00 to 0x64 for 0% to 100% respectively.

Bit	7	6	5	4	3	2	1	0
Field								

Fields

StateOfHealth
This read-only function returns an unsigned integer value expressed as a percentage of the ratio of predicted FCC (25C SoH Load Rate) over the DesignCapacity(). The range is 0x00 to 0x64 for 0% to 100% respectively.

ChargeVoltage

Address
[0x30]

Default
[0x0000]

Returns the desired charging voltage in mV to the charger

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

ChargeVoltage
Returns the desired charging voltage in mV to the charger

ChargeCurrent

Address
[0x32]

Default
[0x0000]

Returns the desired charging current in mA to the charger

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

ChargeCurrent
Returns the desired charging current in mA to the charger

DesignCapacity

Address
[0x3C]

Default
[0x0000]

In SEALED and UNSEALED access This command returns the value stored in Design Capacity and is expressed in mAh. This is intended to be a theoretical or nominal capacity of a new pack but should have no bearing on the operation of the gas gauge functionality.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

DesignCapacity
In SEALED and UNSEALED access This command returns the value stored in Design Capacity and is expressed in mAh. This is intended to be a theoretical or nominal capacity of a new pack but should have no bearing on the operation of the gas gauge functionality.

AltManufacturerAccess

Address
[0x3E]

Default
[0x0000]

MAC Data block command

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

AltManufacturerAccess
MAC Data block command

MACData

Address
[0x40]

Default
[0x0000]

MAC Data block

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

MACData
MAC Data block

SafetyAlert

Address
[0x50]

Default
[0x00000000]

This command returns the SafetyAlert flags on AltManufacturerAccess or MACData.

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field					UTD	UTC			CTOS			PTOS						OTD	OTC		ASCD		ASCC		AOLD		OCD			OCC	COV	CUV

Flags

UTD
Undertemperature During Discharge

UTC
Undertemperature During Charge

CTOS
Charge Timeout Suspend

PTOS
Precharge Timeout Suspend

OTD
Overtemperature During Discharge

OTC
Overtemperature During Charge

ASCD
Short-Circuit During Discharge

ASCC
Short-Circuit During Charge

AOLD
Overload During Discharge

OCD
Overcurrent During Discharge

OCC
Overcurrent During Charge

COV
Cell Overvoltage

CUV
Cell Undervoltage

MACDataSum

Address
[0x60]

Default
[0x00]

MAC Data block checksum

Bit	7	6	5	4	3	2	1	0
Field								

Fields

MACDataSum
MAC Data block checksum

MACDataLen

Address
[0x61]

Default
[0x00]

MAC Data block length

Bit	7	6	5	4	3	2	1	0
Field								

Fields

MACDataLen
MAC Data block length

5.3.9 Audio

wm8731

- Generated with MrT Device Utility
- Bus: I2C, SPI
- RegMap: Register Map
- Datasheet: https://statics.cirrus.c…
- DigiKey: WM8731CSEFL-ND
- I2C Address: 0x34

Description

Aduio codec

Register Map

Name	Address	Type	Access	Default	Description
LEFT_IN	0x00	uint16	W	0x0097	Left line in control
RIGHT_IN	0x01	uint16	W	0x0097	Right line in control
LEFT_OUT	0x02	uint16	W	0x0079	Left Headphone Out control
RIGHT_OUT	0x03	uint16	W	0x0079	Right Headphone Out control
AN_PATH	0x04	uint16	W	0x000A	analog audio path control
DIG_PATH	0x05	uint16	W	0x0008	Digital audio path control
POWER_DWN	0x06	uint16	W	0x009F	Power Down control
DIG_IFACE	0x07	uint16	W	0x009F	Digital audio interface format
SAMPLE	0x08	uint16	W	0x0000	Sampling control
ACTIVE	0x09	uint16	W	0x0000	Active Control
RESET	0x0F	uint16	W	0x0FFF	Reset control

Registers

LEFT_IN

Address
[0x00]

Default
[0x0097]

Left line in control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field							LRINBOTH	MUTE			VOLUME					

Flags

MUTE
Mutes Left input

LRINBOTH
Left to Right Channel Line Input Volume and Mute Data Load Control

Fields

VOLUME
Volume control for Left input in 1.5dB steps range -34.5dB -> +12dB

Name	Value	Descriptions
MIN	b00000	-34.5dB
0dB	b10101	0db Gain
MAX	b11111	+12dB
STEP	b00001	1.5dB Step

RIGHT_IN

Address
[0x01]

Default
[0x0097]

Right line in control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field							LRINBOTH	MUTE			VOLUME					

Flags

MUTE
Mutes Right input

LRINBOTH
Left to Right Channel Line Input Volume and Mute Data Load Control

Fields

VOLUME
Volume control for right input in 1.5dB steps range -34.5dB -> +12dB

Name	Value	Descriptions
MIN	b00000	minimum -34.5dB
0dB	b10101	0db Gain
MAX	b11111	maximum +12dB
STEP	b00001	1.5dB Step

LEFT_OUT

Address
[0x02]

Default
[0x0079]

Left Headphone Out control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

LEFT_OUT
Left Headphone Out control

RIGHT_OUT

Address
[0x03]

Default
[0x0079]

Right Headphone Out control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

RIGHT_OUT
Right Headphone Out control

AN_PATH

Address
[0x04]

Default
[0x000A]

analog audio path control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field								SIDEATT		SIDETONE	DAC-SEL	BY-PASS	IN-SEL	MUTEMIC	MIC-BOOST	

Flags

MICBOOST
Microphone Input Level Boost

MUTEMIC
Mute Mic input to ADC

INSEL
Selects input between Mic and Line-in

BYPASS
Combines Line-in signal to Output

DACSEL
DAC Select

SIDETONE
Combines Mic signal to Output

Fields

SIDEATT
Side Tone attenuation

Name	Value	Descriptions
6dB	b00	6dB of attenuation
9dB	b01	9dB of attenuation
12dB	b10	12dB of attenuation
15dB	b11	15dB of attenuation

DIG_PATH

Address
[0x05]

Default
[0x0008]

Digital audio path control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field											HPOR	DACMU	DEEMP		ADCHPD	

Flags

ADCHPD
ADC High Pass Filter

DACMU
DAC Soft Mute

HPOR
Store dc offset when High Pass Filter disabled

Fields

DEEMP
De-emphasis Control

Name	Value	Descriptions
DIS	b00	Disable
32kHz	b01	32 kHz
44_1kHz	b10	44.1 kHz
48kHz	b11	48 kHz

POWER_DWN

Address
[0x06]

Default
[0x009F]

Power Down control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field								POWEROFF	PEAK-OUTPD	OS-CPD	OUTPD	DACPD	AD-CPD	MICPD	LINEINPD	

Flags

LINEINPD
Line Input Power Down

MICPD
Microphone Input an Bias PowerDown

ADCPD
ADC Power Dow

DACPD
DAC Power Down

OUTPD
Powers down ALL outputs including digital

OSCPD
Oscillator Power Down

CLKOUTPD
CLKOUT power down

POWEROFF
POWEROFF mode

DIG_IFACE

Address
[0x07]

Default
[0x009F]

Digital audio interface format

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field								BLCK-INV	MAS-TER_MODE	LR-SWAP	LRP	IWL		FORMAT		

Flags

BLCKINV
Inverts the bit clock

MASTER_MODE
Enables Master mode

LRSWAP
Swaps LR clock polarity

LRP
DACLRC phase control (in left, right or I2S modes)

Fields

IWL
Word Length. Audio data size

Name	Value	Descriptions
32BIT	b11	32 bit sample size
24BIT	b10	24 bit sample size
20BIT	b01	20 bit sample size
16BIT	b00	16 bit sample size

FORMAT
Selects digital audio format

Name	Value	Descriptions
RIGHT_JUST	b00	MSB-First right justified
LEFT_JUST	b01	MSB-first left justified
I2S	b10	I2S format. MSB-First left -1 justified
DSP	b11	DSP Mode. frame sync + 2 data packed words

SAMPLE

Address
[0x08]

Default
[0x0000]

Sampling control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field																

Fields

SAMPLE
Sampling control

ACTIVE

Address
[0x09]

Default
[0x0000]

Active Control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field															Enable	

Flags

Enable
Enables Digital Audio interface

RESET

Address
[0x0F]

Default
[0x0FFF]

Reset control

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field								RESET								

Fields

RESET
Setting to 0 resets the device

5.3.10 FPGA

Spartan6

Datasheet: https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf

Driver for configuring Spartan 6 FPGA using an 8 bit selectmap interface

5.3.11 RegDevice

This module provides a generic driver for accessing register based devices. It supports devices on both I2C and SPI buses. Since most register based devices use the same access scheme, this provides a consistent base for device drivers.

mrt-device

The recomendded method for creating device drivers based on this module,is to use the mrt-device which is part of the mrt-utils toolset. This provides a very consistent usage of the regdev module, and also creates an easily parseable device file as a byproduct. This can be used for better documentation as well as a basis for automated testing of hardware.

```
pip3 install mrtutils
```

Step 1: Define device:

Devices are defined with a YAML file.

To generate a blank template: .. code-block:: bash

    mrt-device -t /path/to/file.yml

The descriptor file contains device information such as part numbers, links to datashees, and other relevant information. It also contains definitions of registers and data structures on the device. The entities in the definition are:

registers

`registeres` are individualy addressable memory registers on the device. each register can have the folowing attributes:

addr
: register address on device

type
: register type, (default is uin8_t)

perm
: premissions on register R for read, W for write

desc
: description of register. used for code documentation

default
: default value of the register

fields

`fields` are data fields contained in registers. They are grouped by register and they contain the following attributes:

mask
: this specifies the mask for the field. This is used to mask and shift data to match the field.

vals
: this is a list of possible values and their descriptions for the field.

Note: If a field is defined with a single bit mask, and no values, it is interpretted as a 'flag'. Flag fields have macros generated for setting, clearing, and checking them.

Then fill out the template. example from **`hts221 driver <https://github.com/uprev-mrt/device-hts221`_** :

```
---
name: HTS221
description: Humidity and Temperature Sensor
category: Device
requires: [RegDevice,Platform]
datasheet: https://www.st.com/content/ccc/resource/technical/document/datasheet/4d/9a/9c/
↪ad/25/07/42/34/DM00116291.pdf/files/DM00116291.pdf/jcr:content/translations/en.
↪DM00116291.pdf
mfr: STMicroelectronics
mfr_pn: HTS221TR
digikey_pn: 497-15382-1-ND

prefix: HTS
bus: I2C
i2c_addr: 0xBE

registers:
- WHO_AM_I:      { addr: 0x0F , type: uint8_t, perm: R, desc:  Id Register, default: 0xBC}
- AV_CONF:       { addr: 0x10 , type: uint8_t, perm: RW, desc: Humidity and temperature␣
↪resolution mode}
- CTRL1:         { addr: 0x20 , type: uint8_t, perm: RW, desc: Control register 1}
- CTRL2:         { addr: 0x21 , type: uint8_t, perm: RW, desc: Control register 2}
- CTRL3:         { addr: 0x22 , type: uint8_t, perm: RW, desc: Control register 3}
- STATUS:        { addr: 0x27 , type: uint8_t, perm: R, desc: Status register}
- HUMIDITY_OUT: { addr: 0x28 , type: int16_t, perm: R, desc: Relative humidity data }
- TEMP_OUT:      { addr: 0x2A , type: int16_t, perm: R, desc: Temperature data}

- H0_rH_x2:      { addr: 0x30 , type: uint8_t, perm: R, desc: Calibration data}
- H1_rH_x2:      { addr: 0x31 , type: uint8_t, perm: R, desc: Calibration data}
- T0_DEGC_x8:    { addr: 0x32 , type: uint8_t, perm: R, desc: Calibration data}
- T1_DEGC_x8:    { addr: 0x33 , type: uint8_t, perm: R, desc: Calibration data}
- T1T0_MSB:      { addr: 0x35 , type: uint8_t, perm: R, desc: Calibration data}
- H0_T0_OUT:     { addr: 0x36 , type: int16_t, perm: R, desc: Calibration data}
- H1_T0_OUT:     { addr: 0x3A , type: int16_t, perm: R, desc: Calibration data}
- T0_OUT:        { addr: 0x3C , type: int16_t, perm: R, desc: Calibration data}
- T1_OUT:        { addr: 0x3E , type: int16_t, perm: R, desc: Calibration data}

fields:
    - STATUS:
        - TEMP_READY: { mask: 0x01, desc: indicates that a temperature reading is ready }
        - HUM_READY: { mask: 0x02, desc: indicates that a humidity reading is ready }

    - CTRL1:
        - ODR:
            mask: 0x07
            vals:
            - ONESHOT: { val: 0, desc: readings must be requested}
            - 1HZ: { val: 1, desc: 1 hz sampling}
            - 7HZ: { val: 2, desc: 7 hz sampling}
            - 12_5HZ: { val: 3, desc: 12.5 hz sampling}
```

Step 2: generate the code

To generate the code, use mrt-device and specify an input and an output path:

```
mrt-device -i device.yaml -o .
```

The tool will generate 3 files (using hts221 as an example):

hts221.h
: header file for driver

hts221.c
: Source file for driver

hts221_dev.h
: Macros generated from device file. this contains macros for addresses, values, masks, and functions for accessing fields/flags in registers.

Step 3: customize

This will provide a good base with access to all of the register. To add more functionality you can add to the code. If you want to ability to modify the device file further, keep your code inside of the 'user code' blocks provided:

```
/*user-block-init-start*/
/*user-block-init-end*/
```

If the device does not follow the normal register access schemes, you can specify your own, and redirect the mrt_regdev_t fRead and fWrite function pointers to them.

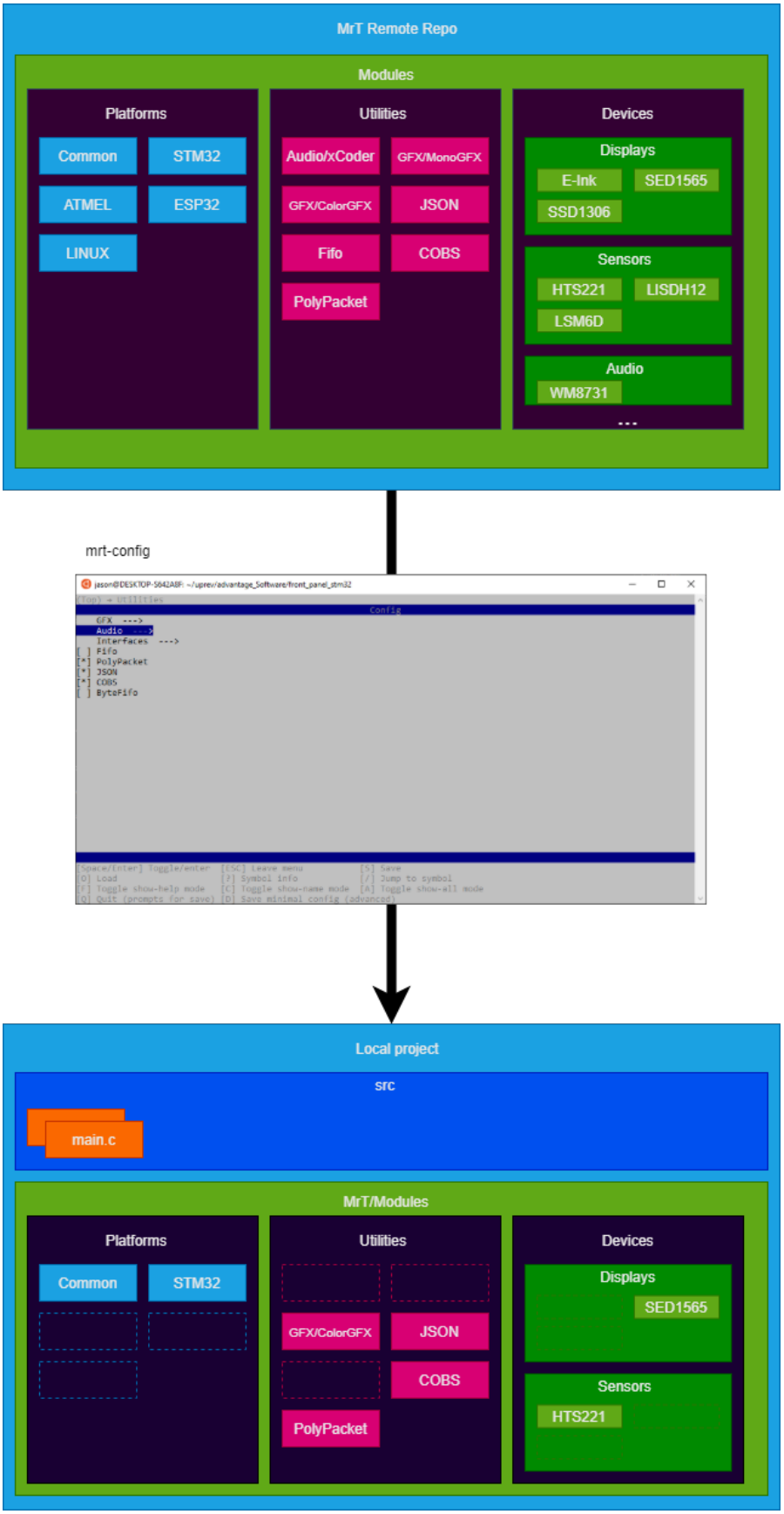
```
/**
*@brief writes buffer to address of device
*@param dev ptr to generic register device
*@param addr address in memory to write
*@param data ptr to data to be written
*param len length of data to write
*@return status (type defined by platform)
*/
mrt_status_t my_write_function(mrt_regdev_t* dev, uint32_t addr, uint8_t* data,int len );

static mrt_status_t hts_init(hts221_t* dev)
{
    /*user-block-init-start*/
    dev->mRegDev.fWrite = my_write_function;
    /*user-block-init-end*/
    return MRT_STATUS_OK;
}
```

CHAPTER

SIX

ARCHITECTURE

At its core `MrT` is just a git repository that contains a bunch of reusable submodules. mrt-config is just a tool that lets you browse submodules from that repo remotely, and add them to your own repo.


MrT Remote Repo
Modules
Platforms
Common
STM32
ATMEL
ESP32
LINUX
Utilities
Audio/xCoder
GFX/MonoGFX
GFX/ColorGFX
JSON
Fifo
COBS
PolyPacket
Devices
Displays
E-Ink
SED1565
SSD1306
Sensors
HTS221
LISDH12
LSM6D
Audio
WM8731
---
mrt-config
Local project
src
main.c
MrT/Modules
Platforms
Common
STM32
Utilities
GFX/ColorGFX
JSON
COBS
PolyPacket
Devices
Displays
SED1565
Sensors
HTS221

6.1 Custom Remotes

By default mrt-config will use UpRev-MrT as the remote repo, but you can actually use any remote repo using the `-r` option. This allows users to maintain custom sets of modules and private repos.

It works by parsing the `.gitmodules` file, so it will work with any repo that has submodules, there are no special files required.

6.2 mrt.yml files

Even though the tool will work on repos without any special files, `mrt.yml` files can extend the functionality. If you run the mrt-doc tool in the root of a repo, it will check all of the submodule paths in that repo for `mrt.yml` files and combine them into a root `mrt.yml` file. The main use for this is to gather all of the requirements for the submodules, so when you select one in the `mrt-config` tool, it can automatically select the dependencies.

CHAPTER

SEVEN

ADDING MODULES

This section covers the information needed for contributors to add modules to the framework

7.1 Creating a Module

`mrt-config` works by grabbing the list of submodules in the main uprev-mrt repo . When you import a module into your project, it adds that submodule to your project using the same relative path it has in the main repo.

So to add a module, you need to create a repo for the module, and then add it as a submodule to the uprev-mrt repo.

Note: Repo names for modules should be all lowercase and hyphenated with the module category as a prefix. example: the `Fifo` module's repo is `utility-fifo`

---

mrt.yml file

Every module should contain an `mrt.yml` file with a name, description, category, and requires field

example from `Fifo` module:

```
---
name: fifo
description: generic fifo utility
category: utility
requires: []
```

Once you have the basic module added, you can begin adding code. The modules structure will vary based on what type of module it is. See below for specifics when adding a *Platform* , *Device* , or *Utility* module

7.2 Platform Modules

Platform modules are meants to abstract any IO operations. This can normally be done by typdefing native platform types to the mrt_xx_t equivalent, and using a macro to pass through operation. In some cases, you may have to get a little creative to make it work, but the macros make the system pretty flexible.

When adding a platform, the header and symbol must be added to `Platforms/Common/mrt_platform.h`

example from `Platforms/Common/mrt_platform.h`

```
...

#if MRT_PLATFORM == MRT_STM32_HAL
    #include "Platforms/STM32/stm32_hal_abstract.h"
    #define MRT_PLATFORM_STRING "STM32_HAL"
    #include "platform_check.h"
#endif

...
```

Then in the header for the module, you can abstract the various IO operations.

7.2.1 Delay Abstraction

- MRT_DELAY_MS(ms)

7.2.2 Uart Abstraction

- typedef xx mrt_uart_handle_t;
- MRT_UART_TX(handle, data, len, timeout)
- MRT_UART_RX(handle, data, len, timeout)

7.2.3 GPIO Abstraction

- typedef xx mrt_gpio_t
- MRT_GPIO_WRITE(pin,val)
- MRT_GPIO_READ(pin)
- MRT_GPIO_PORT_WRITE(port, mask, val)
- MRT_GPIO_PORT_READ(port)

7.2.4 I2C Abstraction

- typedef xx mrt_i2c_handle_t
- MRT_I2C_MASTER_TRANSMIT(handle ,addr,data,len, stop, timeout)
- MRT_I2C_MASTER_RECEIVE(handle ,addr, data, len, stop, timeout)
- MRT_I2C_MEM_WRITE(handle, addr, mem_addr, mem_size, data, len, timeout)
- MRT_I2C_MEM_READ(handle, addr, mem_addr, mem_size, data, len, timeout)

7.2.5 SPI Abstraction

- typedef xx mrt_spi_handle_t
- MRT_SPI_TRANSFER(handle ,tx, rx ,len, timeout)
- MRT_SPI_TRANSMIT(handle, tx, len, timeout)
- MRT_SPI_RECIEVE(handle, tx, len, timeout)

7.2.6 Mutex Abstraction

- MRT_MUTEX_TYPE
- MRT_MUTEX_CREATE(m)
- MRT_MUTEX_LOCK(m)
- MRT_MUTEX_UNLOCK(m)
- MRT_MUTEX_DELETE(m)

7.2.7 printf

- MRT_PRINTF(f_, ...)

Note: Not every function has to be used. Any undefined functions will be defined as NOP() and a warning will be displayed at compile time to let the user know the function is not available on the platform.

7.2.8 Example from `Platforms/Atmel`

```
...

//Delay Abstraction
#define MRT_DELAY_MS(ms) delay_ms(ms)

//Uart Abstraction
typedef struct io_descriptor* mrt_uart_handle_t;
#define MRT_UART_TX(handle, data, len, timeout) io_write(handle, data, len)
#define MRT_UART_RX(handle, data, len, timeout) io_read(handle, data, len)

//GPIO Abstraction
typedef uint8_t mrt_gpio_t;
typedef enum gpio_port mrt_gpio_port_t;
#define MRT_GPIO_WRITE(pin,val) gpio_set_pin_level(pin,val)
#define MRT_GPIO_READ(pin) gpio_get_pin_level(pin)
#define MRT_GPIO_PORT_WRITE(port, mask, val) gpio_set_port_level(port, mask, val)
#define MRT_GPIO_PORT_READ(port) gpio_get_port_level(port)

//printf
#define MRT_PRINTF(f_, ...) printf((f_), __VA_ARGS__)

...
```

7.3 Device Modules

Devices are the most commonly added module type, because every project has unique hardware. The main thing to keep in mind with a Device module, is that all of the IO operations must go through an abstracted platform function. This means you can not use any native IO calls. For instance all GPIO writes must use `MRT_GPIO_WRITE()`, and all UART transmits must use `MRT_UART_TX()` etc.

The `mrtutils` package contains a tool called *mrt-device* that can be used to create device drivers for register based devices.

7.4 Utility Modules

Utilities are the easiest modules to add, because they do not have to interact with hardware. Because these modules can be run on any system, they are all required to have a unit test with 80% code coverage.

CHAPTER

EIGHT

CODING PRACTICES

All of the modules should be written in pure C since the goal is to be reusable across many embedded platforms.

8.1 Documentation

All Modules should include a 'README.rst' file in the root of the modules directory. The README files are automatically combined and updated in the Reference section of this page. If the documentation contains references to other pages or images, they must be in a subdirectory named 'doc'.

Note: *README.md* files are also supported, but rst is preferred

8.2 Code Comments

All public functions should be documented using doxygen style comments:

```
/**
 *@brief Draws a bitmap to the buffer
 *@param gfx ptr to mono_gfx_t descriptor
 *@param x x coord to begin drawing at
 *@param y y coord to begin drawing at
 *@param bmp bitmap to draw
 *@param val pixel value on
 *@return status of operation
 */
mrt_status_t mono_gfx_draw_bmp(mono_gfx_t* gfx, int x, int y,const GFXBmp* bmp, uint8_t␣
↪val);
```

8.3 Unit Tests

The Unit Tester for MrT recursively searches the modules for any file ending with '_UT.cpp', and adds them to the GTest project. To add a Unit test to a module just add a file that ends with _UT.cpp.

Note: To keep projects from trying to compile the Unit test files, they are wrapped with `#ifdef UNIT_TESTING_ENABLED` .. `#endif //UNIT_TESTING_ENABLED`

8.4 Pull Requests

Because modules are typically developed as part of a seperate project, Pull Requests for the module should be reviewed along with the code for that project. There currently is not support for this on Bitbucket Cloud, but I am looking into a solution for this.

CHAPTER

NINE

MRT FRAMEWORK

Modular **R**eusability and **T**esting Framework

MrT is a collection of reusable modules that can be easily integrated into new projects. Each module is designed and maintained according to guidelines and standards to keep consistency. This allows uniform implementation, documentation and testing.

9.1 Modules

There are three types of modules in the *MrT* framework *Platforms*, *Devices*, and *Utilities*

9.1.1 Platforms

Platforms are abstractions for specific platforms. This could be an OS or an MCU family. Each platform contains abstracted interfaces such as GPIO, Uart, SPI, and I2C. This allows the device modules to have a common interface for all platforms. When using a platform module, check the Readme for the module for the integrations steps specific to that platform. Normally these are just the steps to include the *Modules* directory in the projects include path, and define the `MRT_PLATFORM` symbol

9.1.2 Devices

Devices are modules for supporting commonly used ICs in projects. This would include common sensors, flash/eeprom memory, displays, battery charge controllers, etc.

Device modules contain all the logic needed for their operation and communicate using abstracted interfaces from platform modules

9.1.3 Utilities

Utilities are modules that provide a common functionality with no need for abstraction i.e., they do not depend on any specific hardware or platform. These include Fifos, Hashing functions, encoders/decoders, and messaging protocols. Because these do not rely on any hardware, they can be used without a `Platform` module