Contents

Use cases and notes

When you either need high precision or the ability to perform logic based on responses received from commands sent, this library of code is well-suited for that purpose. This is what Scriptable Base Comm code excels with:

- Send a certain command for long periods of time (without the need of a console).
- Monitor the responses and perform any action if a certain field has changed.
- Gather logs (acceptable in Sniffer format) without the need of a Sniffer box while sending and receiving commands.
- Be able to use it on both Linux and Windows platforms.

However, if you simply want to send maybe a command or two or at a fixed auto repeating interval and don't really care about long-term testing then look to the 'Base Comm' app developed by Greg Topel which is more suited towards that.

Couple examples of good use cases:

- A lot of what reliability team does: moving an actuator back on forth based on a duty cycle, gradually ramping up a motor with various RPMs for month's long testing.
- Reproduce obscure bugs in responses like a Push Report causing phantom errors by flooding a PTC board with Push Commands and then monitoring the Push Report data to see if a byte was shifted in. See issue here.

XML files: how to read and use them

What are XML files? How are they used for base communication?

XML (Extensible Markup Language) files are used for organizing and structuring data by defining a rule or patterns for encoding data in both human and machine-readable manner. Read more here.

In our case, these XML files hold the command and response structures of all defined messages between the M4 (Console) and base microcontrollers. This is particularly flexible as different bases have different command structures to accommodate for certain features in the code and it makes it easy to create new commands and responses (if done correctly).

Scriptable Base Comm takes in these files (which you specify at the top of main.cpp) and parses them into a tree-like data structure holding all these message structures for you to load in and start scripting away.

How do I read an XML file for base communication?

To first find the relevant XML file you are interested in, look at instructions included in README.md within `src/xml`. All messages are arranged in format of command 'nodes' containing information about a message packet. Let's go through an example node:

```
<Commands>
  <SearchTag>20:F0:04</SearchTag>
  <TargetID>20</TargetID>
  <SenderID>F0</SenderID>
  <MsgValue>04</MsgValue>
  <MsgName>IC_Statistics_Request</MsgName>
  <PackLen>07</PackLen>
  <DataNames>command</DataNames>
  <DataTypes>byte</DataTypes>
  <DataSizes>1</DataSizes>
  <DataDetails>1 = clear statistics
2 = send statistics</DataDetails>
  <DefaultData>00</DefaultData>
  <IsBootModeCmd>False</IsBootModeCmd>
  <DuplicateCmd />
  <CmdNotes />
</Commands>
```

TAKEN FROM SRC/XML/ DTCOMMANDSTMEV.XML

You'll notice each message is enclosed within the root element of <Commands> and </Commands>. Inside each command element there is the actual structure itself: a search tag for quickly searching messages, a target and sender ID to figure out where to send a message to and where it's coming from, a message value assigning a unique ID to each message, the message name itself, and the actual data types as well. The representation for Target and Sender IDs is as shown below (taken from PTC docs found here):

Module Identification (ID) Numbers:

System Controller (Console) = 0xF0 (can be any unused address)
Test Engineering Control = 0xBC
PTC-HS Motor Controller = 0x10 (aka PTC-HV = High Voltage)
PTC-LS Controller = 0x11 (aka PTC-LV = Low Voltage)
Incline Controller = 0x20
Adjustable Cushioning Controller = 0x21
Eeprom HS Controller = 0x40
Eeprom LS Controller = 0x41

To read the data fields, you start by locating what field you want to modify. In the IC_Statistics_Request (TX), there is only one configurable field which is titled 'command'. Then you'll see it's a variable of type "byte" which corresponds to an unsigned 8-bit integer based on the type conversion found at the top of main.cpp. Its data size reflects the size of the field in bytes, which in this case is only 1 byte. The data details are typically unused (used in Base Comm), unless you explicitly access the code to do so and lastly the default

data is what this field is initialized to when these structures are loaded in. Let's look at a slightly harder example:

```
<Commands>
  <SearchTag>F0:10:87</SearchTag>
  <TargetID>F0</TargetID>
  <SenderID>10</SenderID>
  <MsgValue>87</MsgValue>
  <MsgName>Accumulated_Data_Report</MsgName>
  <PackLen>0C</PackLen>
  <DataNames>status,address,data</DataNames>
  <DataTypes>byte,byte,long</DataTypes>
  <DataSizes>1,1,4</DataSizes>
  <DataDetails>0 = no read or not valid
1 = good data read,data address in eeprom,data value</DataDetails>
  <DefaultData>00,00,00,00,00,00</DefaultData>
  <IsBootModeCmd>False</IsBootModeCmd>
  <DuplicateCmd />
  <CmdNotes />
</Commands>
```

SRC/XML/ DTCOMMANDSTMEV.XML

From the IDs, we know it's going to the console (0xF0) from the high side (0x10) so its a response (RX) as we are emulating the console with this code. It has three comma-separated data fields with the parsed info below:

Message field "status": byte, 1, 00, 0 = no read or not valid 1 = good data read

Message field "address": byte, 1, 00, data address in eeprom

Message field "data": long, 4, 00 00 00 00, data value

Following the format: "name": DataType, DataSizes, DefaultData, DataDetails

Notice how all data-related items are comma-separated and line up accordingly. If there is a mismatch in these or an incorrect amount of bytes assigned to a data type (ex. assigning 3 bytes to a long) then bad things will happen…

Note: All number fields are reported in hexadecimal EXCEPT the DataSizes which are in decimal base 10.

How do I load messages and use them?

Specify the name of the XML file at the top in main.cpp. After the initialization code in main() that says to "IGNORE" the code, initialize a Message pointer object from the return value of table.findMessage(std::string) which takes in the "MsgName" of a command

node. Uppercase/lowercase does not matter if the string matches an existing command node. ‘table’ is a global MessageTable variable that contains all the command nodes and the member function findMessage returns that command node from the table. Next there the three primary Message object member functions are as follows:

bool setField(string dataName, T data)

comm_error sendMessage()

getField<T>(string dataName)

setField allows you to modify a field of an outgoing command, where you have to specify the DataName (from XML, ignores casing) and templatized data of type ‘T’ to set it to. ‘T’ should be ideally set to the corresponding “DataType”. If the data is not the expected type, it’ll pull the first few bytes or match it to the best of its ability which works for most cases.

To actually send a command, call the member function sendMessage() which returns 0 on success or an error due to timeout or other issues.

For getField<T>(), you will have to know the DataType in advance, however. At the top of main.cpp you’ll see the type translations from the XML to actual code. It can be extremely challenging at times to find the type of a data field (ex. just look at push report…). An easy way to check it with the code is to make use of the getter function getType() found within a MessageField object. See an example below to print all data names and their types:

```
int main (int argc, char **argv)
{
    if (!loadDocument(xmlFile, table)) return 1;

    /* Commenting out communication and logger temporarily */
    // if (!initComm(argc, argv))         return 1;
    // initLogger();

    // Load in the Push Report message structure
    Message* pushReport = table.findMessage("Push_Report");

    // Go through each comma-seperated data and print its name and type
    for (MessageField* data: pushReport->getDataFormat())
    {
```

```
        cout << "DataName: " << data->getName() << " TYPE: " << data->getType()
<<'\n';
    }
}
```

I recommend you look at the documented example later in the docs for reference. What makes this code unique compared to others is the ability to use any base ‘XML’ file which is truly what makes this powerful.

Detailed Interface description

Logs

To see all logging settings go to src/logger/log.h and set the macros at the top to your preferences. The features include directing logging output (ex. to stdout or a .log) the directory of the logs, max number of log files, and a time for automatically rotating logs (currently unimplemented). Note that if you change the LOG_PATH, you must also change it in the LOGS_DIR of the Makefile. The way the rotation is configured is doing a “tail” on the .log file gives you the most recent logs and a “head” on the greatest .log. * gives you the oldest logs. Logs are always cycled based on the configuration specified so there will never be a flood of memory making it great for long-term testing.

Serial Constants

Message timeouts and baud rate can be modified in src/serial/lf_comm.h

Important function documentation

Message* MessageTable::findMessage(std::string name)

Will search for message by name, this is slower than search tag. Case insensitive. Returns Message pointer, or NULL on failure.

```
enum comm_error {
    NONE,           // 0
    BAD_CHECKSUM,   // 1
    TIMEOUT,        // 2
    EMPTY_READ,     // 3
    INVALID_MSG,    // 4
};
```

comm_error enum type declaration used for identifying errors during serial communication.

comm_error Message::sendMessage()

Object-based function for sending a message through the serial COM port. Returns a type of error, usually NONE (enum offset 0) if successful.

bool setField(const std::string& dataName, const T& input)

@tparam T The type of the input value.

@param dataName The identifier of the data field to be set.

@param input The value to set for the data field.

@return true if the field was successfully set, false otherwise.

Sets the value of a field identified by `dataName` to the value provided in `input`. The field type `T` is later casted to the type of DataName with the use of helper functions.

T getField(const std::string& dataName)

Opposite of setField(), takes in a dataName and returns the data. IMPORTANT: must be called using a template specialization (ex. getField**<BYTE>**(...)). Read more here.

uint64_t timeSinceEpoch()

Returns the time in ms from January 1, 1970. Useful for taking time differences to measure duration.

After you have created your script, you can build and run it using the following description (found in the README.md as well):

`make clean` to remove all object files and executables (good to run before `make`)
`make` to build

Windows: `.\main.exe COMXX -v` is the accepted format.
Linux: `./main /dev/ttyUSBX -v`

'X' is a placeholder for a decimal number.
`-v` flag enables Sniffer logs output to `logs/`

Lift actuator code walkthrough

Problem statement: Create a script that will move XT actuators based on a certain Duty Cycle, record diagnostic data, and increment and output a counter. See the code.

```
/**************** XML ****************/
STRING xmlFile = "dtCommandsPNC.xml";
/*************************************/

// Declare relevant command and response 'Message' structures
Message *GC_Watts_Torque_TX;
Message *GC_Watts_Torque_RX;

Message *diagnostic_TX;

// Variables to hold useful information
uint64_t cycleCounter = 0;
float    dutyCycle    = 0.1;
int      period       = 100;    // in seconds

int ON_TIME  = dutyCycle*period;
int OFF_TIME = period - ON_TIME;

// Command configuration parameters
BYTE product_type       = 16;
BYTE actuator_level_max = 50;
BYTE actual_max         = 21;
BYTE actuator_level_min = 0;

void moveActuators(bool& firstCycle)
{
    uint64_t timePre = timeSinceEpoch();    // in ms

    while (timeSinceEpoch() - timePre < ON_TIME*1000)   // while it has been less than ON_TIME seconds..
    {
        GC_Watts_Torque_TX->setField("product_type", 23);   // send a dummy command with another product type to update actuator_level in RX
        GC_Watts_Torque_TX->sendMessage();

        GC_Watts_Torque_TX->setField("product_type", product_type);

        // If the actuator is extended to its maximum stroke
        if (GC_Watts_Torque_RX->getField<BYTE>("actuator_level") == actual_max)
        {
            GC_Watts_Torque_TX->setField("actuator level", actuator_level_min); // Set the target level to minimum
            while (GC_Watts_Torque_RX->getField<BYTE>("actuator_level") != actuator_level_min)  // keep sending actuator to home
                GC_Watts_Torque_TX->sendMessage();  // actually send the message
        }

        // If the actuator is at the home position (minimum stroke)
        else if (GC_Watts_Torque_RX->getField<BYTE>("actuator_level") == actuator_level_min)
        {

            GC_Watts_Torque_TX->setField("actuator level", actuator_level_max);
            while (GC_Watts_Torque_RX->getField<BYTE>("actuator_level") != actual_max)
                GC_Watts_Torque_TX->sendMessage();

            if (firstCycle == true)
            {
                firstCycle = false;
                cycleCounter++;     // if not on boot up, increment counter once actuator if actuator reaches home
            }
        }
    }
}

int main (int argc, char **argv)
{
    if (!loadDocument(xmlFile, table)) return 1;
    if (!initComm(argc, argv))         return 1;
    initLogger();

    // Initialize relevant command and response 'Message' structures
    GC_Watts_Torque_TX = table.findMessage("GC_Watts_Torque");
    GC_Watts_Torque_RX = table.findMessage("GC_Watts_Torque_Report");
    diagnostic_TX      = table.findMessage("Diagnostic_Command");

    bool firstTime = true;

    while (1) // run forever
    {
        moveActuators(firstTime);

        usleep(OFF_TIME*1000000); // STOP for OFF_TIME amount of seconds (1 sec = 10^6 microseconds)

        // For diagnostic reports in logs
        diagnostic_TX->sendMessage();

        // Do useful information with base variables
        cout << "Cycles: " << cycleCounter << '\n';
    }

}
```

With any automation problem, there are always key items that you are interested in monitoring or sending. In this case, I am interested in moving an actuator from the PNC base board and recording some general diagnostic data as well.

Coming into this, I already knew what command structure I need to use from the XML file but if you have no idea, you can try searching the XML file (using CTRL+F) and looking

related terms. For example, if I was interested in the rpm of the motor on a Symbio Treadmill I'll do a CTRL+F within the dtCommandsTMEV.xml and search for rpm. You may get a ton of results, but you'll eventually narrow your result to what you are interested in (ex. actual_rpm inside Push Report and RPM_Report).

Every command sent has a response back (except BOOT commands), so to always update a certain variable in the MessageTable, you have to send a command at a specific polling rate (ex. send diagnostic command every 3 seconds to update diagnostic report data).

Anyway, back to the original problem: I first declare relevant command and response 'Message' structures based on the dtCommandsPNC.xml file, variables to hold useful information (like counters), and some parameters of the data fields I am going to set the command's messages to. I make these variables global just so they can be accessed by my moveActuators() function which takes in a flag indicating if it's the first time at home position (to not increment the counter).

Line-by-line description:

Program execution begins in main() so after ignoring the first three lines, I use table.findMessage() to initialize the command and response structures. I create a local variable to address the starting on home position counter issue and then I have an infinite loop that calls moveActuators().

Inside moveActuators() I immediately get the current time from a fixed offset (Jan 1, 1970) and create another loop to keep the actuators moving until the current time offset (which can be obtained with timeSinceEpoch()) subtracted with the initial time is less than the ON_TIME. Pay close attention to the syntax of how I am calling the setField(), sendMessage(), and getField() functions. In particular, the getField<T>() NEEDS to have a DataType specified. As shown in the code the actuator_level is a byte so I get the getField<BYTE>(...).

After this is completed, the next line in main() is usleep(x) which sleeps for *x* amount of microseconds. Then I send a diagnostic command to add a diagnostic report within the logs. Finally, I output the number of cycles to the terminal (these could be outputted to a GUI, file, etc.) and the loop repeats.