EMBEDDED DATA COLLECTION AND ANALYSIS USING MATLAB

Introduction

Embedded control and measurement systems can often benefit from high-level algorithm development using tools such as Matlab. In order to accomplish this, it is necessary to export data from the embedded system to a PC. While Mathworks offers add-on packages that facilitate embedded algorithm development, these packages can be expensive. Often, a simple way of collecting data from an embedded system for basic analysis is all that is required.

This appnote and supporting source code provides a simple portable framework for accomplishing the real-time transport of data between the embedded system and a PC running Matlab using a serial transport such as RS-232.

Requirements

The implementation of the architecture described in this appnote has been tested on the MAX35103EVKIT2 evaluation kitasdfasdf, but it could be easily ported to other platforms. The MAX35103EVKIT2 is recommended for initial evaluation and reference.

This appnote assumes that the user as a basic understanding of Matlab, Matlab MEX, the C language, and the Win32 API. Experience with IAR ARM and Microsoft Visual C++ is also helpful.

The following tools are required for full evaluation:

- Maxim MAX35103EVKIT2 Evaluation Kit
- Microsoft Visual C++
- Matlab (no additional packages required)
- IAR Embedded Workbench for ARM

The Microsoft Visual C++ Community Edition is available for free download from the Microsoft Web site. IAR ARM is available for evaluation from IAR Systems. The MAX35103EVKIT2 is available from Maxim Integrated as well as many electronic distributors such as Digikey and Mouser. Matlab is available from Mathworks, Inc.

Example Implementation using the MAX35103EVKIT2

The MAX35103EVKIT2 was chosen for the initial implementation of the framework described in this appnote. The board consists of a MAX32620 (ARM Cortex M4) and MAX35103 (ultrasonic time-to-digital converter). Together these components enable the collection of liquid flow measurements through an ultrasonic flow body. These measurements are formatted and transmitted to the host PC running Matlab.

A basic understanding about the MAX35103EVKIT2 and the MAX35103 will be helpful in understanding the data communicated by the framework and the format of the host/target protocol packets. Please refer to the MAX35103EVKIT2 documentation for details specific to the embedded platform.

The data transmitted by the MAX35103EVKIT2 embedded target to the host PC running Matlab is the output of the MAX35103 time-to-digital converter. This data stream is an array of time measurements between ultrasonic pulse transmission and reception. The **collect.m** Matlab script (detailed later in this

document) offers a quick starting point for collecting data from the MAX35103EVKIT2 PCB. **Figure 1** below shows how to retrieve and access the first five upstream time measurements from the MAX35103EVKIT2 board.

Command Window

```
>> s1 = collect(6,10,600);
fx >> plot_flow(s1)
```

FIGURE 1 - MATLAB DATA COLLECTION EXAMPLE USING COLLECT.M AND ANALYZE_SAMPLES.M

plot_flow.m plots fluid flow data collected by the MAX35103EVKIT2 board and compares two popular methods of flow calculation. *Figure 1* below is a Matlab plot of flow data over the period of five minutes. The data shows a change in flow as a gate valve is turned on by hand. For more information regarding this data, please see the MAX35103EVKIT2 documentation.
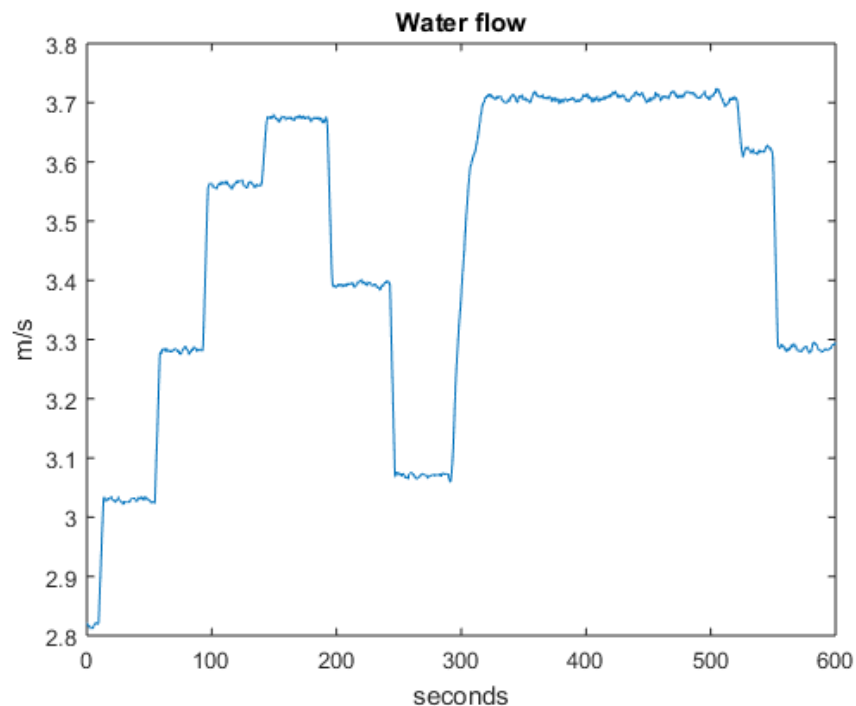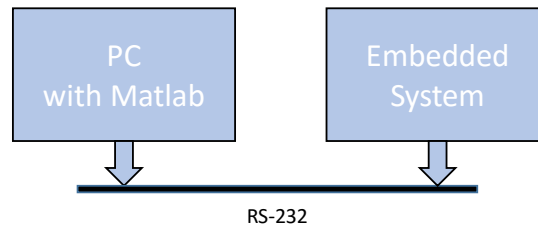

FIGURE 2 - MATLAB PLOT OF WATER FLOW DATA

System Architecture

Data collection using this this framework requires a host PC capable of running Matlab and a target embedded system with a serial interface. The framework provided here specifically supports RS-232, but could easily be ported to support other interfaces.

The framework spans both the PC host and the embedded target. C code is compiled for the target embedded system as well as the host system. The current host implementation requires a Win32 platform, but this could be ported to another OS such as Linux.

The embedded target hardware must provide a serial interface of some kind. The current framework implementation supports UART but is designed to be portable to other transports. The target microcontroller must have enough resources to support the framework data and code requirements but also enough throughput to move data without excessively impacting the performance of the embedded system. An RTOS is not required by the framework, but it does not preclude the use of one.

The current framework implementation was developed with a 96MHz Cortex-M4 processor and could move data across the UART at near maximum throughput with little processor overhead. Smaller systems might require adjustments to the framework to work acceptably.

Figure 4 below depicts the software components on both the host and the target. Components in blue are C language modules that run on the host. Components in red are C language modules that run on the target. Components in green are common to both domains. Matlab scripts, in purple, are standard m-scripts that interface with an application specific interface. The embedded framework implementation described here is built to run on the Maxim MAX35103EVKIT2, an ultrasonic water flow measurement platform.

Components in grey are external platform specific components.

FIGURE 4 - SYSTEM ARCHITECTURE

C/C++ and 4GL languages that can interface with Win32 DLL's, like C# or Python, are supported via the "Win32 DLL I/F" module. This allows for easy support of a custom host application that may not require Matlab.

The "COM" module implements the base host/target protocol. This is where you'll find data packet definitions specific to the embedded application. The "Serialize" module implements escape-based packetization of binary data. Both modules are easily portable via callbacks that interface with communications API's (host) or embedded peripherals (embedded target).

The definitions in the "COM" module drive the implementations in the "Host-side Protocol" and the "Device-side Protocol" components. These implementations are where commands/responses specific to the host and target are implemented and generally have a lot of commonality.

The following sections will detail each major architectural module from top to bottom beginning with the host.

PC Host Architecture

The host side of the architecture depicted in *Figure 4* consists primarily of Matlab and OS specific interfaces and is described in detail in the following sections.

Matlab Scripts

At the top of the host stack are the Matlab scripts that perform application specific data collection and control. The script *collect.m*, show in *Code Listing 1,* is a simple example of how to use the Matlab MEX interface to open, set parameters, and collect data from the MAX35103EVKIT2 Evaluation Kit.

```matlab
function samples = collect( comport, sample_rate_hz, sample_time_seconds )

    % this function collects and returns data from the MAX35103EVKIT2 running the
    % matlab example firmware

    % sample_rate_hz is the desired sampling rate in Hz.  Valid range is
    % from 10Hz to 100Hz

    h_flow = svflow('open',comport);
    if( h_flow )
        svflow('start',h_flow,sample_rate_hz);
        samples = svflow('get_samples',h_flow,sample_rate_hz*sample_time_seconds);
        svflow('stop',h_flow);
        svflow('close',h_flow);
    else
        samples = [];
    end
end
```

CODE LISTING 1 - COLLECT.M

Only one public function can exist in a MEX interface module. In this case, it is *svflow()*. This function is how Matlab scripts call into the MEX module. The name of the function is arbitrary, but 'svflow' was chosen as moniker for the overall host/target protocol as implemented on MAX35103EVKIT2 (a water flow measurement platform).

The first argument to *svflow()* is a text string that indicates a sub-function to call. The second argument is the object handled referencing a specific flow object. This object is returned by s*vflow('open',…)*. This is the basic method that the framework uses to accommodate MEX while supporting an object-oriented architecture.

The *collect.m* script above, calls the sub-function 'open' to open COM6 on the Windows host machine. Next, 'start' is called to specify a sample and begin sample collection. 'get_samples' is then called to collect samples at the defined sampling rate. When this synchronous collection is complete, the flow object is stopped and closed. The script returns the object, samples, which contains all the data captured by the MAX35103 on the evaluation board.

Matlab MEX Interface

The Matlab MEX interface component for the MAX35103EVKIT2 is implemented in a set of C language files. It has access to internal Matlab functions and exposes a standard interface that Matlab scripts can call. All MEX specific functionality is contained in mex.c which provides a Matlab specific wrapper to the core protocol functionality implemented in flow.c (which also serves as the Win32 DLL interface) and serialize.c/com.c which are common to the host and the embedded target.

The module is created by using the Matlab mex() function. The *compile.m* script in **Code Listing 2** compiles the host C files into a form usable by Matlab. The output of this command is the Matlab MEX executable file. Matlab must have been previously configured to use a native toolchain. See the Mathworks website for information about how to set up a toolchain for compiling MEX modules.

```
mex -g -output svflow -I'dll' -I'..' dll/mex.c dll/svflow.c ../serialize.c ../com.c
```

CODE LISTING 2 – COMPILE.M

The Matlab MEX interface specific to the MAX35103EVKIT2 are implemented in mex.c. Matlab requires all MEX modules to implement the function mexFunction() as the sole interface to the functionality provided by the module. To provide a way for a single MEX module to provide multiple object-oriented member functions, a sub-function mechanism is used. In *Code Listing 3*, mexFunction() references a function call table is used to dispatch sub-functions. The call table itself is shown in *Code Listing 4*

```c
for (i = 0; i < ARRAY_COUNT(s_function_table); i++)
{
    if (!lstrcmpA(s_function_table[i].p_name, func))
    {
        s_function_table[i].p_func(nlhs, p_lhs, nrhs - 1, p_rhs + 1);
        return;
    }
}
```
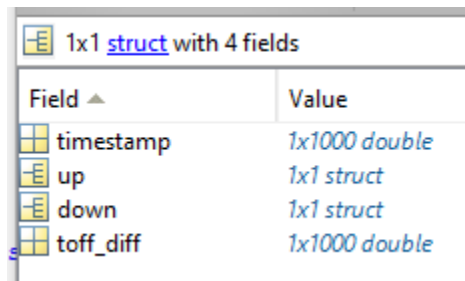
CODE LISTING 3 – MEXFUNCTION() SUB-FUNCTION DISPATCH

```
static const function_table_t s_function_table[] =
{
    { "get_samples", mex_get_samples },
    { "open", mex_open },
    { "close", mex_close },
    { "start", mex_start },
    { "stop", mex_stop }
};
```

CODE LISTING 4 - SUB-FUNCTION CALL TABLE

The mex_* functions referenced in the call table are thin wrappers to the Win32 DLL functions detailed in the next section.

The Matlab MEX interface also formats the data received by the embedded target in a form compatible with the double matrix-oriented nature of Matlab. The top-level object returned by the Matlab MEX Interface is a Matlab struct with the following fields:
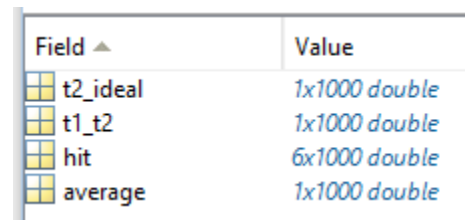
Field ▲	Value
1x1 struct with 4 fields	
timestamp	1x1000 double
up	1x1 struct
down	1x1 struct
toff_diff	1x1000 double

FIGURE 5 - TOP-LEVEL MATLAB DATA OBJECT

The size of the double arrays *timestamp* and *toff_diff* is variable.

The *up* and *down* members are Matlab structs with the following format:

Field ▲	Value
t2_ideal	1x1000 double
t1_t2	1x1000 double
hit	6x1000 double
average	1x1000 double

FIGURE 6 - MATLAB STRUCT CONTAINING MAX35013 TIME MEASUREMENTS

Again, the length of each array is variable. This data corresponds directly with the data output by the MAX35103 on the MAX35103EVKIT2 Evaluation board.

In mex.c, the fuction *mex_get_samples()* formats the data received by the embedded target using Matlab mx* functions.

```
static void mex_get_samples(int nlhs, mxArray *p_lhs[], int nrhs, const mxArray *p_rhs[])
{
    char * sample_fieldnames[] =
    {
        "timestamp",
        "up",
        "down",
        "toff_diff"
    };

    svflow_sample_t sample;

    void **pp = (void*)mxGetData(p_rhs[0]);
    uint32_t sample_count = (uint32_t)mxGetScalar(p_rhs[1]);

    mxArray *p_sample_struct = mxCreateStructMatrix( 1, 1, ARRAY_COUNT(sample_fieldnames),
sample_fieldnames );

    mxArray *p_timestamp = mxCreateNumericMatrix( 1, sample_count, mxDOUBLE_CLASS, mxREAL );
    mxSetField( p_sample_struct, 0, "timestamp", p_timestamp );

    mxSetField( p_sample_struct, 0, "up", create_direction_struct( sample_count, &sample.up ) );
    mxSetField( p_sample_struct, 0, "down", create_direction_struct( sample_count, &sample.down ) );

    mxArray *p_toff_diff = mxCreateNumericMatrix( 1, sample_count, mxDOUBLE_CLASS, mxREAL );
    mxSetField( p_sample_struct, 0, "toff_diff", p_toff_diff );

    sample.p_timestamp = (double_t*)mxGetData( p_timestamp );
    sample.p_tof_diff = (double_t*)mxGetData( p_toff_diff );

    svflow_get_samples( *pp, &sample, sample_count );

    p_lhs[0] = p_sample_struct;
}
```

CODE LISTING 5 - MEX_GET_SAMPLES() IN MEX.C

The '*up*' and '*down*' struct members are constructed in the *create_direction_struct()* function in mex.c

Win32 DLL Interface

The Win32 DLL interface is implemented in svflow.c (which also contains the much of the protocol and platform specific code). The source code package associated with this appnote contains a Visual Studio project that can be used to build the DLL. However, the DLL isn't required by Matlab. It is provided simply to aid those interested in writing custom data analysis code in a language that can interface with DLL's.

*Code Listing 6*Code Listing 6 below shows the DLL interface function supported by the MAX35103EVKIT2. The functions correspond exactly to the sub-functions called in the Matlab script shown in *Code Listing 1*

```
void* svflow_open( uint32_t comport);
void svflow_close(void *pv_context);

uint32_t svflow_get_samples(void *pv_context, flow_sample_t
*p_flow_sample, uint32_t sample_count);

void svflow_start( void *pv_context, float_t sample_rate_hz );
void svflow_stop( void *pv_context );
```

CODE LISTING 6 - FLOW.H

svflow_open () returns an opaque flow communications context object associated with the given Win32 COM port or NULL if an error occurred.

svflow_close() closes communications and frees resources using the context object returned by sv*flow_open()*.

svflow_start() tells the embedded target to begin collecting flow samples at the specified sampling rate.

svflow_stop() tells the embedded target to end data collection.

While these functions are specific to the MAX35103EVKIT2, they could easily be replaced by functions appropriate to other embedded applications.

Host Protocol

The protocol used by the host and the embedded target are built on common definitions and functions in com.c/h and serialize.c/h. Protocols supported by the architecture generally consist of command/response and indication events. The host protocol is implemented in svflow.c with dependences on com.c and serialize.c—which are common to the host and the embedded target.

The host-side protocol uses com_* functions to issue commands and decode responses and indications. For example, in **Code Listing 7**, com_tx() is used to send a 'com_host_start_sampling_t' command packet to the embedded target.

It's worthwhile to note that all protocol functions are singled-threaded blocking calls.

```c
void svflow_start( void *pv_context, float_t sample_rate_hz )
{
    context_t *p_context = (context_t*)pv_context;
    if( p_context )
    {
        com_host_start_sampling_t cmd;
        cmd.sample_rate_hz = sample_rate_hz;
        com_tx( &p_context->com, &cmd, COM_ID_HOST_START_SAMPLING,
                sizeof( com_host_start_sampling_t ) );
    }
}
```

CODE LISTING 7 - FLOW_START() IN FLOW.C

The host protocol module also defines data types that correspond with data types that transit the communications link, but are not identical to them. This difference allows some useful decoupling of the data types required by this module and the modules above. Specifically, it decouples the packet format (concise, single precision floats) from the data format used to accommodate Matlab (verbose, matrix-oriented doubles). This means that translation code must exist in flow.c as can be seen in the serialize callback function in **Code Listing 8**

```c
static bool serialize_cb(void *pv_context, const void *pv_data, uint16_t length)
{
    context_t * p_context = (context_t *)pv_context;
    const com_union_t *p_packet = (const com_union_t*)pv_data;

    if (p_packet->hdr.id == COM_ID_DEVICE_FLOW_SAMPLE )
    {
        com_device_flow_sample_t *p_com_sample =
            (com_device_flow_sample_t*)&p_packet->flow_sample;
        if (!p_context->sample_ndx )
        {
            p_context->time_offset = p_com_sample->timestamp;
        }

        svflow_sample_t *p_flow_sample = p_context->p_flow_sample;
        uint32_t ndx = p_context->sample_ndx;

        direction( &p_flow_sample->up, &p_com_sample->up, ndx );
        direction( &p_flow_sample->down, &p_com_sample->down, ndx );

        p_flow_sample->p_timestamp[ndx] = ( (double_t)( p_com_sample->timestamp -
            p_context->time_offset ) ) / 96000000.0;
        p_flow_sample->p_tof_diff[ndx] = p_com_sample->tof_diff;

        p_context->sample_ndx++;
        if( p_context->sample_ndx >= p_context->sample_count )
            return true;
    }
    return false;
}
```

CODE LISTING 8 - DATA TRANSLATION IN FLOW.C

svflow.c also contains initialization and callback functions required to use comports on a Win32 platform as seen in *Code Listing 9* below.

```c
static uint16_t uart_write(com_t *p_com, void *pv, uint16_t length)
{
    DWORD written;
    context_t *p_context = (context_t*)p_com;
    WriteFile(p_context->hComm, pv, length, &written, NULL);
    return (uint16_t)written;
}

static uint16_t uart_read(com_t *p_com, void *pv, uint16_t length)
{
    DWORD read;
    context_t *p_context = (context_t*)p_com;
    ReadFile(p_context->hComm, pv, length, &read, NULL);
    return (uint16_t)read;
}
```

CODE LISTING 9 - WIN32 SERIAL PORT CALLBACKS IN FLOW.C

The COM module implements the routines that abstract the serial transport. com_init() initializes the abstraction object and com_read() is called to de-serialize and dispatch specific commands/responses and indications.

Embedded Target Architecture

The embedded target architecture is conceptually simple and mirrors the host-side architecture excluding the platform and Matlab specific components.

The embedded application and device-side protocol components implement platform and protocol callbacks and configuration specific to the target. In the case of the MAX35103EVKIT2, this implements functions to support the host side svflow_* calls defined in svflow.c. The implementation can be found in board.c.

Like the comport abstraction on the host side, the embedded target has serial port callbacks as can be seen in *Code Listing 10*. The read callback calls a CSL (chip support library) function call to write length bytes to the UART. The return value is the number of bytes actually written. The read callback uses a CSL function call to read all bytes currently available (up to length) on the port.

```c
static uint16_t uart_write(com_t * p_com, void * pv, uint16_t length)
{
    return UART_Write(MXC_UART0, (uint8_t *)pv, length);
}

static uint16_t uart_read(com_t * p_com, void * pv, uint16_t length)
{
    return UART_Read(MXC_UART0, (uint8_t *)pv, length, NULL);
}
```

CODE LISTING 10 - SERIAL PORT CALLBACKS IN MAIN.C

The COM module is used by the embedded target to dispatch commands from the host. com_read() is called from the main application loop and commands are dispatched in serialize_cb() which is listed in *Code Listing 11*.

main.c contains the entirety of the embedded application for flow measurement on MAX35103EVKIT2 and uses the max3510x.c module to interface with the MAX35103 chip. Board.c contains board specific initialization and interrupt dispatch code.

Although this example implementation is specific to the MAX35103EVKIT2, the COM and serialize modules are not platform specific and could be easily ported to most modern microcontrollers and board designs.

```c
static bool serialize_cb(void *pv_context, const void *pv_packet, uint16_t length)
{
    const com_union_t *p_com = (const com_union_t*)pv_packet;
    switch( p_com->hdr.id )
    {
        case COM_ID_HOST_START_SAMPLING:
        {
            if( p_com->start_sampling.sample_rate_hz > 0.0F &&
                p_com->start_sampling.sample_rate_hz <= 200.0F )
            {
                s_sampling_underflow = 0;
                s_sampling_overrun = 0;
                s_sample_state = sample_state_idle;
                s_send_samples = true;
                SYS_SysTick_Config( (uint32_t)((float_t)SYS_SysTick_GetFreq() /
                    p_com->start_sampling.sample_rate_hz), 1);
            }
            break;
        }
        case COM_ID_HOST_STOP_SAMPLING:
        {
            SYS_SysTick_Config( (uint32_t)((float_t)SYS_SysTick_GetFreq() / 10.0F), 1);
            s_send_samples = false;
            break;
        }
    }
    return false;
}
```

CODE LISTING 11 - SERIALIZE_CB IN MAIN.C

Software Package Contents

The target firmware and Windows host software can be downloaded from the Maxim website. It is provided as a zip archive. Extract the archive to a convenient directory on your computer. *Figure 7* below show the directory structure of the target firmware and host software implementation.

```
                          Root
                  Firmware application and
                     protocol modules
        ┌─────────────────┼─────────────────┐
      board              host               IAR
  boards supported   matlab scripts to   IAR ARM project
     by the          comple the MEX I/F      files
   application       and test the firmware
        │                 │
  max35103evkit2         dll
  board support     Win32 DLL and MEX
  for the EV Kit    I/F project and
        │            source files
   ┌────┴────┐
  CSL      max3510x
 MAX32620  interface for the
 chip support  time-to-digital
 library and   convertor
 ARM CMSIS
```
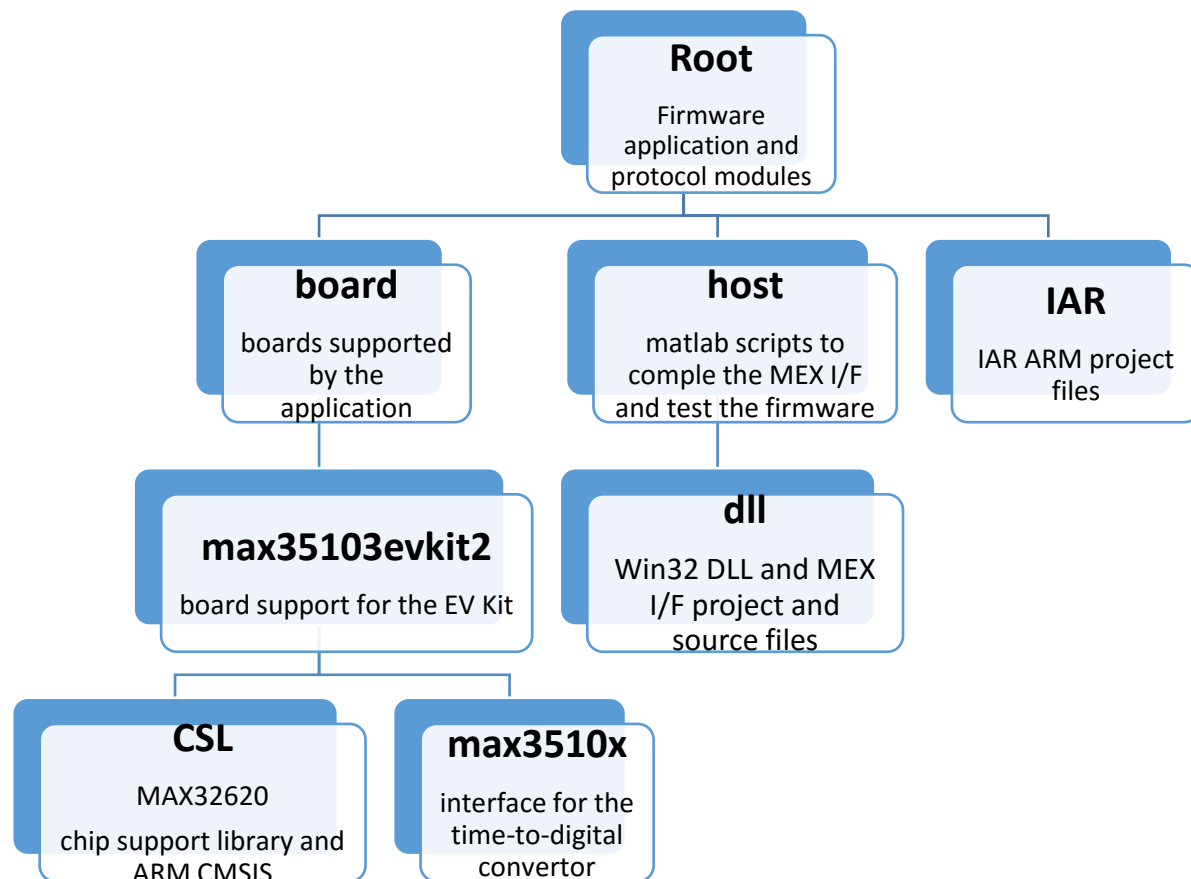
FIGURE 7 - SOFTWARE DIRECTORY STRUCTURE

The root directory contains main()and the COM and Serialize modules. Additionally, transducer.c/h contains parameters specific to the ultrasonic transducers that come with the MAX35103EVKIT2.

The **board** directory contains a sub-directory for each board supported by the Matlab example firmware application. Support for custom user boards could be added here.

The **csl** and **MAX3510x** directories contain code specific the microcontroller and peripherals on the MAX35103EVKIT2.

The **IAR** directory contains the project files used to build and debug the firmware on the MAX35103EVKIT2. New project configurations could be added to support custom user boards. The easiest way to do this would be to copy the project configuration and then modify it to suit the new target.

The **host** and **dll** directories contains all the source necessary to build the Win32 DLL and MEX interface modules. Additionally, you'll find Matlab scripts to compile the MEX interface module alongside the *compile.m* script detailed in ***Code Listing 1***

Building the Target Firmware

The Matlab example firmware can built using IAR ARM. In the **iar** directory, you'll find the IAR project file. Once you load the project, be sure to check that your debugger is configured correctly (see ***Figure 8***). The microcontroller on the MAX35103EVKIT2 can be programmed with any ARM JTAG adapter supported by IAR using the 10-pin ARM header (J1).
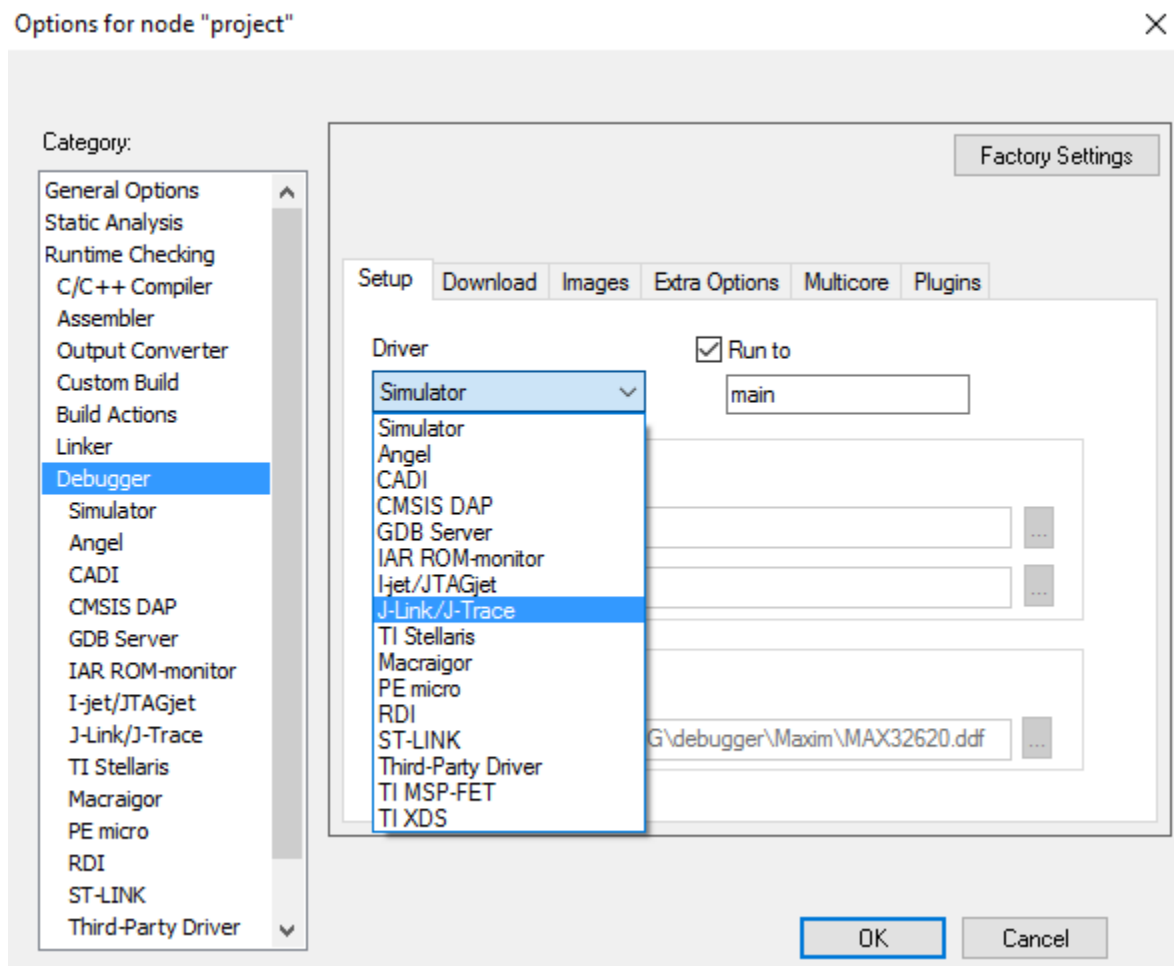
FIGURE 8- IAR DEBUGGER OPTIONS

project.out is the firmware image created when the project is built.
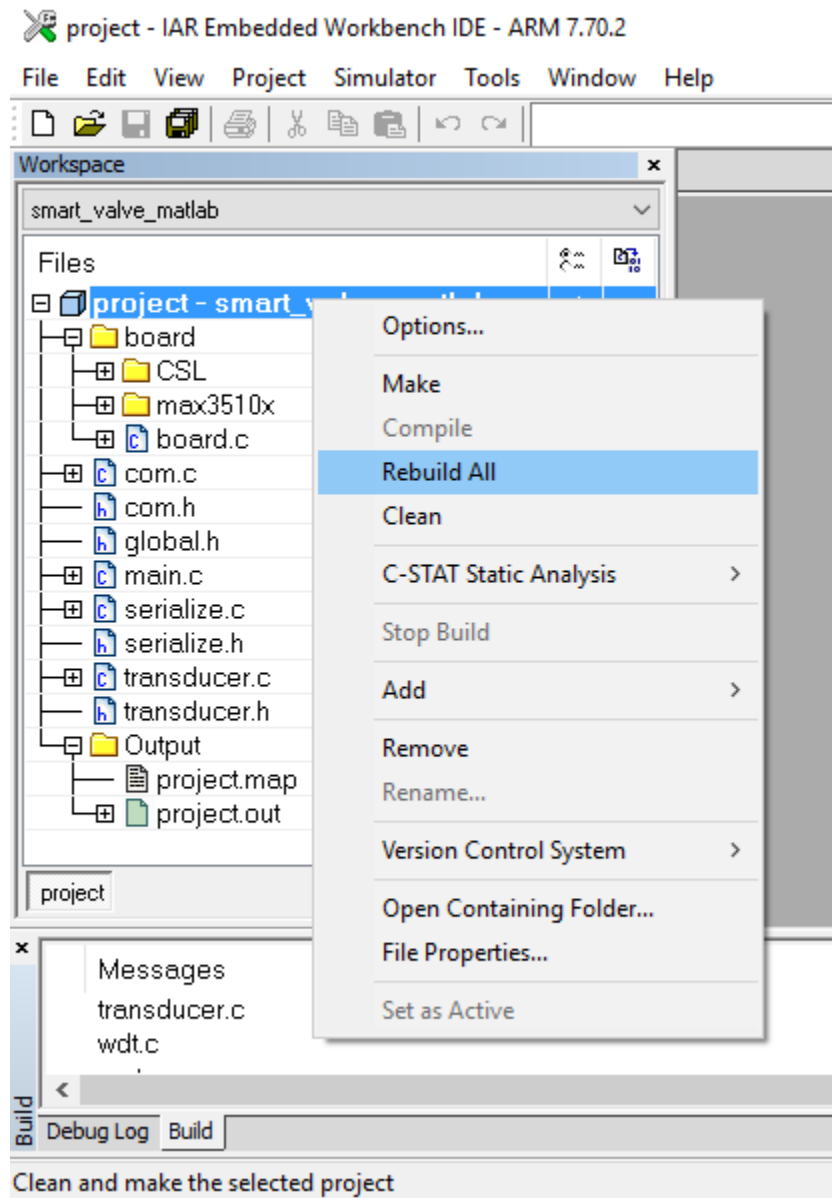

FIGURE 9 - BUILDING THE FIRMWARE WITH IAR

Building the Host Software

The Matlab MEX interface can be built from within Matlab using the *compile.m* script located in the **host** directory as depicted in ***Figure 10***. The output of the build is *flow.mexw64*.

In order to use *compile.m* you must have a Matlab supported C compiler installed. Please refer to the Matworks website for details as this may change from one version of Matlab to the next.

At the time of this writing, the free version of Microsoft Visual C++ can be used by Matlab 2016a to generate MEX files.
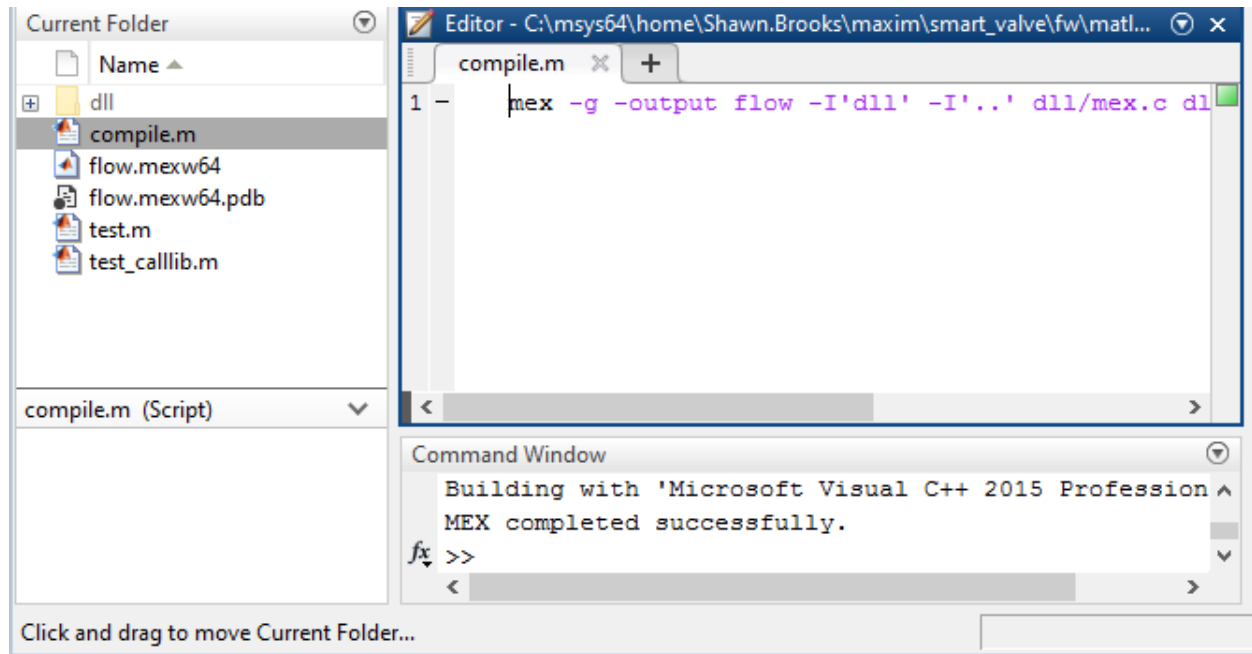
FIGURE 10 - BUILDING THE MEX INTERFACE

The host software can also be built into a DLL module for use by non-matlab programs that know how to talk to DLL's. C# and Python applications (among others) can be supported in this way. The Microsoft Visual C++ project files to build a DLL can be found in the **dll** directory

Hardware Configuration

The MAX35103EVKIT2 PCB must be connected to the ultrasonic flow body as described in the MAX35103EVKIT2 Datasheet. *Figure 11* shows the connections available on the MAX35103EVKIT2 PCB.
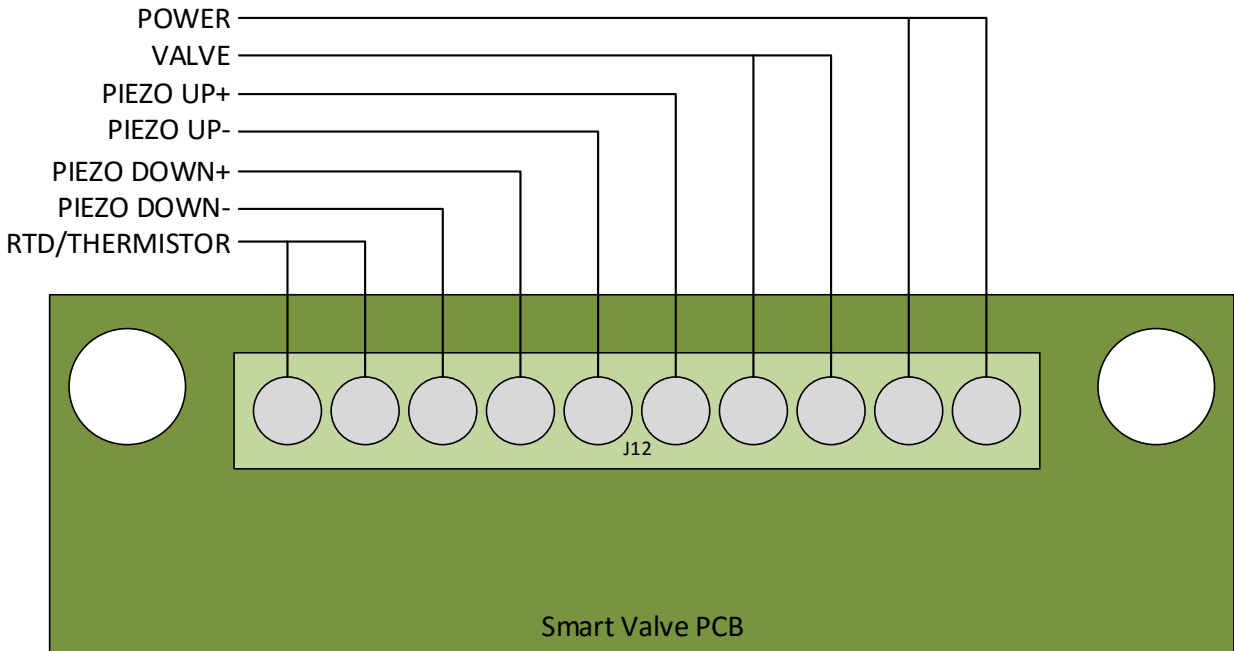
FIGURE 11 – SMART VALVE J12 PINOUT

- POWER should be connected to a 9-24VAC or DC source capable of supplying 100mA.
- VALVE can be left unconnected.
- PIEZOP UP +/- should be connected to one of the flow body transducers
- PIEZO DOWN +/- should be connected to the other flow body transducer.
- RTD/THERMISTOR can be left unconnected.

The rotary switches are not used by the firmware.

Conclusion

Matlab provides a great platform for data analysis and algorithm development. This appnote describes a simple customizable software architecture that can be used to get the data into Matlab without the cost and complexity of commercially available addon modules.