AN3061
Rev. 0, 1/2006

Data Acquisition for the MSC711x Utilizing the SC140 Libraries

by Paula Aaronson and Tina Redheendran

1 Introduction

This document describes data acquisition code for the MSC711x using the StarCore™ SC140 libraries and interfacing to the data through the on-device I/O ports. This document with the code listed in **Appendix A** provides a framework for developing additional signal processing on the MSC711x. As a vehicle for understanding, the code examples demonstrate how the MSC711xEVM can interface to a 40 kHz data acquisition system with an analog-to-digital converter (ADC) and digital-to-analog converter (DAC) and perform signal processing on the received data that is sent to a PC.

1.1 Software Overview

The following modules, when combined, form the software required for running the data acquisition code on the MSC711x.

- Ethernet interface
- TDM interface
- DSP algorithm code
  - 40 kHz bandpass filter
  - Signal processing code
- Control Code

This document focuses on the last two modules listed: the DSP algorithm code and the control code. **Figure 1** shows the complete code flow.

CONTENTS

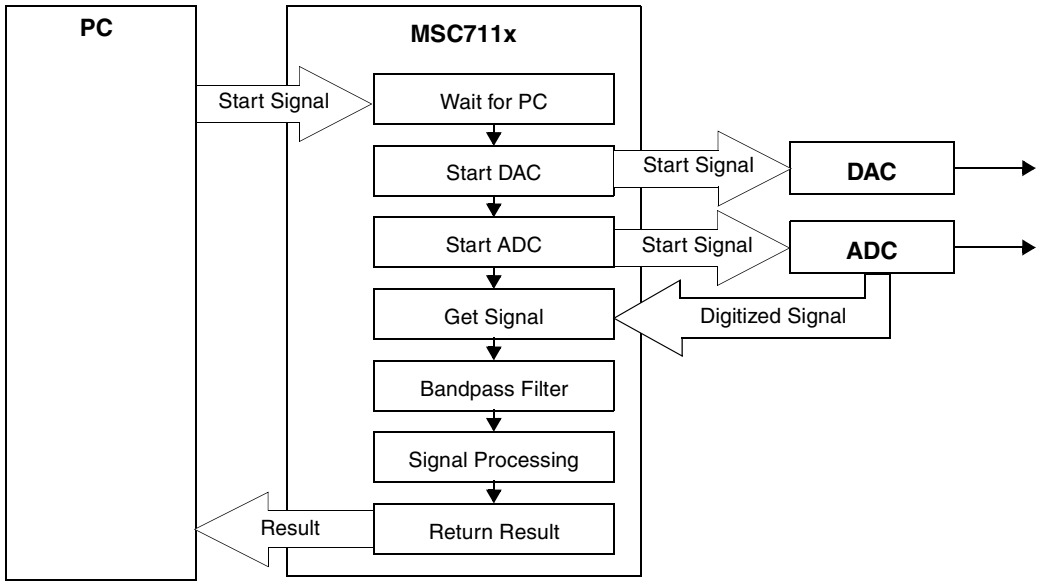

Figure 1. Code Flow

1.2 Ethernet Interface

The MSC711x may include an Ethernet port to connect it remotely to a PC controller where the processed data is used according to the given application. Signals from a PC can trigger the start of the data acquisition and processing. The MSC711x acquires the data and processes it according to commands from the PC. The results of the PC query are returned to the PC where the information is used for the end application and/or displayed. Check the MSC711x device data sheet to verify whether an Ethernet module resides on your MSC711x device. Refer to the *MSC711x Reference Manual* for details on Ethernet programming.

1.3 TDM Interface

The data enters the MSC711x through the TDM port(s) connected to an ADC, and generated data is transferred out of the DSP through the TDM port(s) connected to a DAC. The number of TDM modules differs across MSC711x devices. Check the device data sheet to verify how many TDMs reside on your MSC711x device. Refer to the *MSC711x Reference Manual* and *MSC711x Time-Division Multiplexing (TDM) Usage Examples* (AN2946) for details on TDM programming. The TDM code contains the two following parts:

- *ADC TDM*. When the DSP receives the start signal from the PC, the receive (RX) TDM is triggered and the TDM moves data from the ADC into buffers in the DSP memory.
- *DAC TDM*. When the DSP receives the start signal from the PC, the transmit (TX) TDM is triggered and TDM code generates an output signal and transfers it to the DAC.

The TDM is programmed to place the received data into different buffers according to the amount of data and the rate at which the data is received. The exact number of buffers is determined by the application requirements. While the TDM is filling one buffer, the algorithm code can process the data from the previously filled buffer. This maximizes the throughput of the data acquisition code. The data buffer details are explained in **Section 3**.

1.4 DSP Algorithm Code

The DSP algorithm code processes the received data, including downsampling and a bandpass filter to isolate the signal and filter out any noise. The DSP algorithm code includes any additional signal processing to compute the required result, depending on the needs of the application; this document assumes some unspecified signal processing after the filtering stage. Most of the algorithm code is written in assembly to maximize the processing speed. However, for ease of coding, readability, and maximum future portability, all DSP functions are C-callable. Many DSP signal processing routines are available in the SC140 libraries. The algorithm code is discussed in detail in **Section 3**.

1.5 Control Code

The control code manages the flow of the application by scheduling the sequence of events. For data acquisition, the control code receives the signal from the PC through the Ethernet port, triggers the TDM transmit and receive channels, starts the algorithm code when the data is ready for processing, and sends the result back to the PC though the Ethernet. The control code is written in C and is discussed in detail in **Section 4**.

2 Library Overview

The MSC711x is based on the SC1400 framework and its parallel processing resources, including four MAC units, four arithmetic logic units (ALUs), four bit field units (BFUs), two arithmetic address units, and an efficient five-stage pipeline. Although it is based on a 16-bit orthogonal instruction, the architecture uses variable-length execution sets (VLES). In addition, software written for the four-MAC SC140 can be leveraged for future implementations that contain more or fewer hardware resources.

The SC140x libraries are a collection of software modules covering several functional categories such as mathematics, filtering, frequency domain analysis, and image processing for SC140x DSP customers. The SC140x software library provides C-callable assembly software modules for these library functions optimized for the SC140x cores. The reference manual for each library module provides information about the purpose of the library module and the algorithm used to implement the functionality. The library documentation also describes the inputs and outputs for the modules and any restrictions and limitations. The documentation for each library module describes the C calling syntax to call the assembly module from a C program. Any data structures necessary for use in the library module are described, as well as the hardware core registers used in the assembly implementation of the library module. Also, the performance metrics for the module are listed.

This document uses an FIR library module (`Fir_buffer`) from the SC140x filters library. This function computes a real FIR filter (Direct Form), using real inputs and real coefficients producing a real output. The input sample x is stored in an input buffer. This function requires the inputs to be in the [–1 1) range. The coefficients h are stored in a vector. The filter output result y is stored in an output buffer. Filter-taps M plus one and buffer-length N must be a multiple of four.

The difference equation is:

$$y(j) = \sum_{k=0}^{M-1} h(k) \times (j-k) \qquad 0 \le j \le N-1$$

To take advantage of the parallel processing in the SC1400 core, the input data is read into the delay buffer four samples at a time. The delay buffer is configured into a circular buffer with modulo length equal to the number of delay taps. To use the four ALU in parallel, the length of the input buffer and the delay buffer must be multiples of four. For implementation convenience, four additional words are needed for the delay buffer.

The filter library `Fir_buffer` function prototype is:

```
void SC140xLib_Filters_Fir_buffer (struct stSC140xLib_FIR_IOStruct *);
```

The function receives a pointer to the following structure, which contains pointers to the input and output buffers, the coefficient buffer, and the State variables buffer, as well as the size of these buffers, the current index of the State Variables and the number of Filter taps.

```
typedef struct stSC140xLib_FIR_IOStruct{
       short   *psOutBuffPtr;               //Pointer to output buffer
       short   *psInBuffPtr;                //Pointer to input buffer
       short   *psCoeffPtr;                 //Pointer to coefficient base address
       short   *psStateVarPtr;              //Pointer to State Variable buffer
       short   *psStateVarCurrentPtr;       //Pointer to State Variable current index
       unsigned short usBuffSize;           //Length of the buffer
       unsigned short usFiltTaps;           //Number of taps in the Filter
}stSC140xLib_FIR_IOStruct;
```

This data structure is declared and initialized prior to calling the library function.

The inputs are fractional shorts between [–1 1), with 16-bit input, 16-bit coefficients, and 40-bit accumulation.

```
short * psBuffInPtr                 = pointer to the real input data array with size usBuffSize
short * psCoeffPtr                  = pointer to coefficient buffer
short * psStateVarPtr               = pointer to state variable list: {x(n-1),x(n-2),…}.
short * psStateVarCurrentPtr        = pointer to current point in state variable list
unsigned short usBuffSize           = input/output buffer length
unsigned short usFiltTaps           = taps of fir+1
```

The outputs are fractional shorts between [–1 1).

```
short * psOutBuffPtr                = Pointer to the output data array with size usBuffSize
```

3 DSP Algorithms

This section describes the algorithms used for the main body of the code. Each of these modules is processed in the MSC711x SC1400 core. The current MSC711xEVM has a crystal sampling frequency of 2.5 MHz for the raw data from the ADC. Since the calculations required to do signal processing are proportional to the sampling rate, we want to reduce the sampling rate to the lowest sampling rate possible. To obtain a sampling frequency closer to a necessary oversampling (OVS), the raw data is downsampled by a factor of 8. Therefore, the input to the bandpass filter has a sampling frequency of 312.5 kHz.

3.1 Downsampling

For data acquisition from a 40 kHz source, the Nyquist frequency is 80 kHz. Since the MSC711xEVM samples the raw data from the ADC at 2.5 MHz, the signal is significantly oversampled and the signal can be downsampled without filtering and not violate the Nyquist theorem. Since we are just downsampling rather than decimating, the process can complete in one stage. To implement the downsampling by a factor of 8, every eighth sample is kept, and the other seven samples in between are thrown away. Downsampling a discrete time signal $x(n)$ consists of keeping only every i^{th} value for an input $x(n)$ with output $y(n) = x(i \times n)$ (see **Figure 2**).

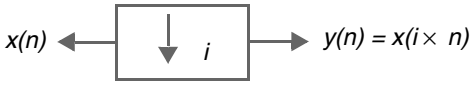

Figure 2. Downsampling Input/Output Diagram

3.2 Bandpass Filter

To pick up only the 40 kHz signal in the data acquisition example, we want a tight bandpass filter around the 40 kHz frequency. The bandpass filter takes the downsampled data and filters it over a 4 kHz bandwidth at a centerpoint of 40 kHz. For the 4 kHz bandwidth with a transition band from the corner frequency to the stop band of only 2 kHz, the filter has 400 taps and a gain of 1 in the passband. The filter chosen is an equiripple FIR filter to minimize the number of coefficients and maintain linear phase. The filter algorithm used is the one implemented in the SC140x filters library `Fir_buffer` function described in **Section 2**, as follows:

$$y(j) = \sum_{k=0}^{M-1} h(k) \times x(j-k) \qquad 0 \le j \le nr - 1$$

The frequency response for the 4 kHz bandwidth filter is shown in **Figure 3**.

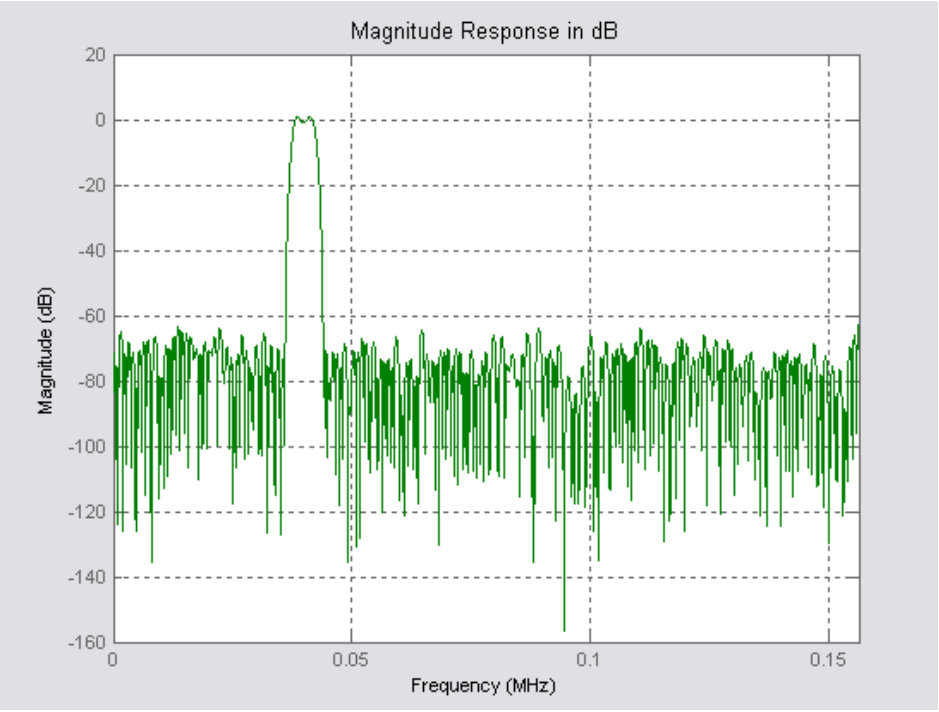

Figure 3. Frequency Response of the Quantized Bandpass Filter

The 400 coefficients for implementing the 4 kHz bandwidth bandpass filter centered at 40 kHz gives a group delay of 200. You must account for this delay if any further processing requires time domain analysis.

3.2.1 Data Input Format

The MSC711xEVM implementation uses a crystal sampling frequency of 2.5 MHz for the raw data from the ADC. To obtain a sampling frequency closer to a necessary oversampling (OVS), the raw data is downsampled by a factor of 8. Therefore, the input to the bandpass filter has a sampling frequency of 312.5 kHz. This provides an input to the bandpass filter of approximately 8 OVS. The input data is 16 bit two's complement integer data.

Note: The ADCs used for the data acquisition example map the incoming signal from 0xFFFF–0x7FFF. Therefore, it is easy to conceptualize the data as 16-bit two's complement integer data. The filter expects the input data to be fractional. Since this example is not dependent on the exact values of the data, this representation is used. For algorithms needing exact precision, you must account for the integer representation rather than a fractional representation.

3.2.2 Coefficient Input Format

There are 400 coefficients for implementing the 4 kHz bandwidth bandpass filter centered at 40 kHz. Only 397 are necessary for the bandpass filter, however, for efficient processing, zeroes are added to pad the end of the file to use all four SC140 ALUs. The coefficient inputs to the bandpass filter function are 16 bit two's complement fractional data.

3.2.3 Output Format

The output from the bandpass filter is the 2.5 MHz sampled data downsampled filtered with the 4 kHz bandwidth bandpass filter centered at 40 kHz. The output data is also 16 bit two's complement integer format.

3.2.4 API

The calling format of the filters is:

```
SC140xLib_Filters_Fir_buffer(&stFir_buffer).
```

The function receives a pointer to the following structure, which contains pointers to the input and output buffers, the coefficient buffer, and the State variables buffer, as well as the size of the buffers, the current index of the State Variables and the number of Filter taps. The structure `struct stSC140xLib_FIR_IOStruct` is populated as follows:

```
stFir_buffer.psInBuffPtr = FilterInput;            //Buffer into the filter
stFir_buffer.psOutBuffPtr = FilterOutput;          //Buffer out of the filter
stFir_buffer.psCoeffPtr = gasCoeffs;               //4kHz bandpass coefficients
stFir_buffer.psStateVarPtr = gasStates             //Current State of Filter;
stFir_buffer.psStateVarCurrentPtr = gasStates      //Current position in States;
stFir_buffer.usFiltTaps = DELAY_TAPS               //Order of the Filter;
stFir_buffer.usBuffSize = FILTER_IN_BUF_SZ         //Size of IO buffers to filter routine;
```

3.3 Signal Processing

The signal processing involved depends on the end application. This simple data acquisition example assumes a MSC711xEVM is used to interface to a data acquisition system and perform signal processing on the received data. However, even this simple example can be expanded for applications requiring more signal processing. For example, robotic control or factory automization could require proximity sensing, localization of objects, size measurement, and liquid or solid levels. The algorithms can be filtering, triangulation, cross-correlation, and adaptive beam forming. Elements to get both range and angle information can be added. As the number of input

elements increases, the position resolution of each object increases. Many other applications would require many other signal processing routines. Signal processing functions such as FFTs/IFFTs, correlation, encoding, decoding, transforms, quantizers, and so forth can be placed into this portion of the code if more processing is required.

4 Control Code

The control code idles until it gets a start signal from the PC via the Ethernet. When the start signal is received, the control code starts the DAC and then starts the ADC. When the TDM indicates it has a full buffer, the control code triggers the filter for that buffer. When all the buffers are filtered, the control code calls the signal processing routine(s). After signal processing, the control code receives the result back from this routine. The control code then triggers the Ethernet to send the computed result to the PC. Then the control code waits for another start signal from the PC and repeats the processing flow. **Figure 4** shows the control code flow for each processing period; the pseudo code for the control code is listed in **Example 1**.

Example 1. Control Pseudo Code

```
While(1){
       While not Start_Signal{
              Wait
       }
       Start DAC
       Start ADC
       For 1-N Buffers{
              Filter Buffer
              Signal Processing
       }
       Signal Processing Completion
       Send Result
}End While(1)
```

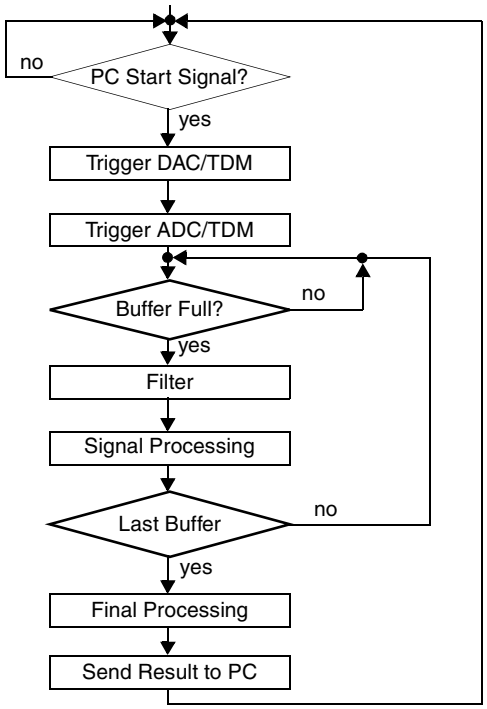

Figure 4. Control Code Flow Chart

The TDM places the received data into different buffers to maximize the throughput of the data acquisition code. The algorithm code processes each buffer while the TDM fills the next buffer. The number and size of the buffers should be chosen carefully. The buffers should not be so large that a long delay is added as the system waits for the buffers to fill. The exact number of buffers should be determined by the application; however, this document assumes that three buffers are required as shown in **Figure 5**.

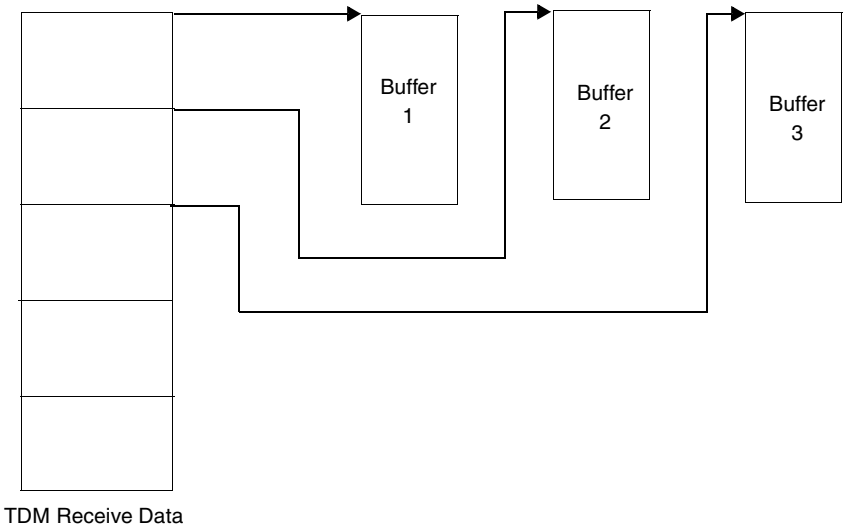

Figure 5. Data Buffer Format

Figure 6 shows the timing for the control code, including the data buffer mechanism. After the start signal is received, the TDM data receive task is divided into three blocks (labeled Get Samples). The data filtering is also divided into three blocks, one for each data buffer. When the data receive and filtering is complete, the signal processing runs to completion and the result is sent to the PC (via the Ethernet). Then the code waits until another start signal is received. Start signals are acknowledged only during this wait period. If a start signal is received while the data is processing (between starting the DAC/ADC and sending the result to the PC), it is ignored.

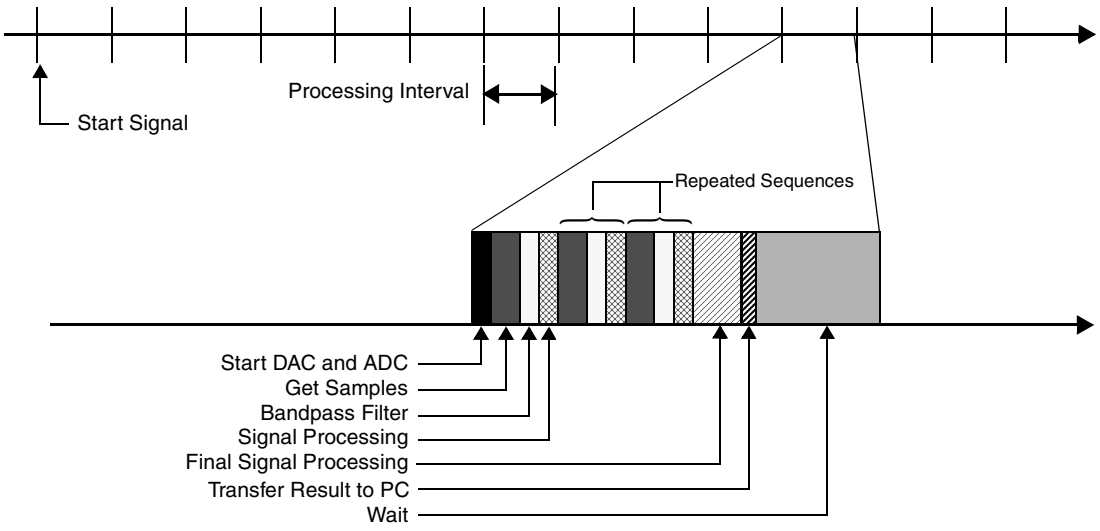

Figure 6. Task Timing

5 Example Application

This section describes an example application using the data acquisition framework. The object of the application is to calculate the distance to a device under test (DUT) using the time of flight of a 20-cycle 40 kHz pulse reflection. The MSC711x receives a start signal from the PC, generates a 20-cycle 40 kHz pulse, and sends it to the DAC. The DAC is connected to a 40 kHz transducer that sends the pulse. The pulse is reflected off the DUT and is received by another 40 kHz transducer connected to the ADC after a delay time of Δt, which is used to calculate the distance to the DUT. The MSC711x receives the reflected return signal from the ADC. When the return pulse is detected, the time of flight is calculated by energy computation and the resulting distance value is returned to the PC. **Table 1** shows the system parameters for the example application.

Table 1. Example Application System Parameters

Parameter	Value
Transducer Frequency	40 kHz
Transmitted Pulse Width	20 cycles
Sampling Frequency	312.5 kHz
Received Pulse Length	157 samples
Maximum Distance	1 m

Figure 7 shows the layout of the example application.

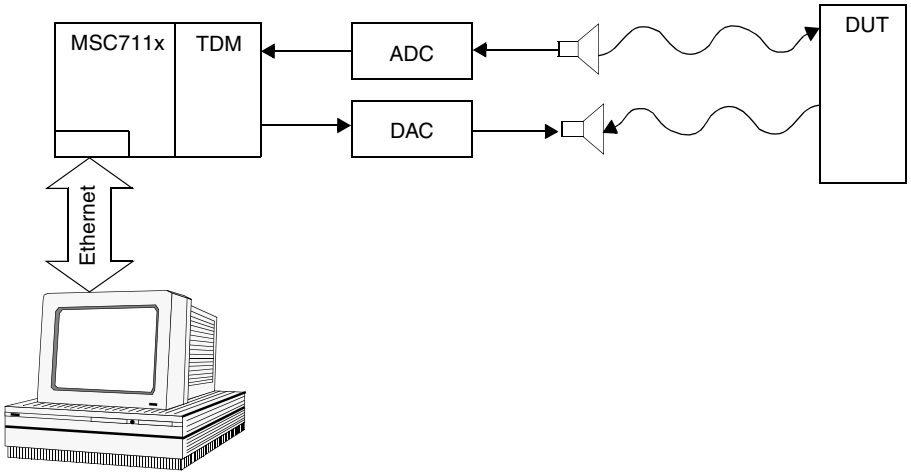

Figure 7. Layout for the Example Application

The downsampling and filter code runs on each buffer. Other signal processing functions can be included that run on each buffer such as FFTs or cross-correlations, to determine the kind of material being detected. However, this document includes only the signal processing needed to calculate the distance of the DUT. The downsampling code and filter described in **Section 3** is valid for this example application. The signal processing for the example application includes code for energy detection (**Section 5.1**) and distance calculation (**Section 5.2**).

5.1 Energy Detection

Each sample of the filtered signal is evaluated to see if it has energy above a given threshold to help screen out false signals from noise and avoid unnecessary computations. The signal is squared and then compared with the threshold. If the signal is above the threshold, the energy from the next 157 samples is added according to the following calculation:

$$Energy = \frac{1}{pulse\ length} \times \Sigma_{pulse\ length}(data)^2$$

The energy is then compared to a minimum energy threshold to determine if a pulse has been detected. If the energy is above the threshold, the index of the start of the received pulse (the start of the 157 samples) is passed to the distance computation function. A typical result is graphed in **Figure 8**. The horizontal line shows the threshold and when the energy goes above this line, the processing stops. If all the samples are evaluated and the pulse is not detected (the energy is not greater than the minimum energy threshold), the energy detection routine returns a negative value to indicate the detection failure. The example code for the energy detection routine is written in assembly and is shown in **Section A.2**.

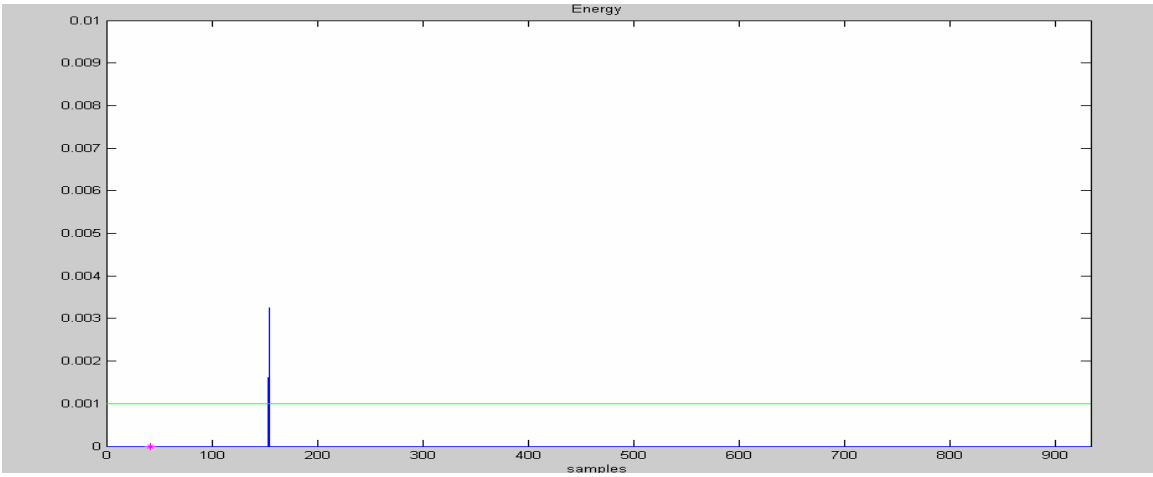

Figure 8. Received Energy

5.1.1 Input Format

The input to the energy detection is the eight OVS filtered data from the bandpass filter. The input data is 16-bit two's complement integer format.

5.1.2 Output Format

The output from the energy detection is the index of the starting sample of the detected pulse. If no pulse is detected, the output is a null character (negative value).

5.1.3 API

The calling format of the energy detection is:

```
Energy_Detect(&stED_buffer).
stED_buffer.sSamples = stED_buffer.sSamples - DELAY_TAPS/2;
```

The function receives a pointer to the following structure, which contains pointers to the filtered data, the number of samples in the buffer, the pulse length, and the two threshold values. The first threshold value determines the minimum energy to indicate the start of a signal. The second threshold determines the minimum energy to indicate the presence of a found signal. The energy detection structure is defined as follows

```
typedef struct stEnergt_Detect_IOStruct{
       short *psInBuffPtr;
       unsigned short usMaxSamples;
       unsigned short usPulseLength;
       unsigned long ulThreshold1;
       unsigned long ulThreshold2;
       short sSamples;
}stEnergt_Detect_IOStruct;
struct stEnergt_Detect_IOStruct
```

and is populated as such:

```
stED_buffer.psInBuffPtr=gasBuffOut;
stED_buffer.usMaxSamples=MaxSamples;
stED_buffer.usPulseLength=PulseLength;
stED_buffer.ulThreshold1=liThreshold1;
stED_buffer.ulThreshold2=liThreshold2;
```

After energy detection, the group delay of the filter must be subtracted from the index of the pulse start because the filter shifts the data by this value. The group delay is half the number of coefficients or DELAY_TAPS/2.

5.2 Distance Computation

When the energy is detected, the distance is calculated by dividing the index of the detected pulse by the sampling frequency to give the time of flight of the reflected pulse. The distance to the DUT is then calculated using the speed of sound. The TDM code must enable the receive and transmit as closely together as possible so that the receive TDM begins at the exact same time as the transmit TDM. Thus the index of the start of the received pulse is an accurate representation of the time of flight of the reflected pulse. For greater precision, an empirical measurement of the offset between the two transducers can be made and subtracted from the distance. However, in this case the time difference was less than the precision of the movement of the DUT. The algorithm is as follows:

$$\Delta t = \frac{index}{sampling\ frequency} = index \times \frac{1}{sampling\ frequency}$$

$$Distance = \frac{c \times \Delta t}{2} = c \times \Delta t \gg 1$$

The example code for the distance computation routine is written in C (see **Section A.3**) because it is such a small section of code and is run only once for each start signal (it is not run for each buffer). However, for more efficient code, convert it to assembly. Because the variables to store the time of flight and the distance are integers and the computation involves division, it is important to ensure that the results do not underflow.

5.2.1 Input Format

The input to the distance computation is the index of the start of the received pulse where energy was detected.

5.2.2 Output Format

The output from the distance computation is the distance to the DUT.

5.2.3 API

The calling format of the distance computation is:

```
long DistCalc(        short siPulseCounter,
                      unsigned long uliSpeedofSound,
                      unsigned long uliInvSampFreq)
```

The function receives the index of the start of the received pulse, the value for the speed of sound being used, and the inverse of the sampling frequency (to avoid the inefficiency of division) and the function returns the distance to the DUT.

5.3 Control Code

The control code for the example application uses the data acquisition control code with the addition of the energy detection and distance computation code. The pseudo control code for the example application shown in **Example 2**.

Example 2. Example Application Control Pseudo Code

```
While(1)
       While not Start_Signal{
              Wait
       }
       Start DAC;
       Start ADC;
       For 1-3 Buffers{
              Filter Buffer
              Signal Processing
       }
       Energy Detection
       Distance Computation
       Send Result
}
```

5.4 Memory Usage

This section looks at the memory sizes of the data buffers. First, the amount of data memory to store all the data from one pulse is calculated. This value is rounded up to a multiple of three to allow the total buffer size to be divided into the three data buffers.

- Distance = 2×1 m = 2 m
- Time is computed using the following equations:
  - Distance/Speed of Sound

    $(2\text{ m})/(343\text{ m/s}) = 5.83 \times 10^{-3}$ s
  - Transmit Pulse Width / Pulse Frequency

    $(20\text{ cyc})/40000\text{ (cyc/s)} = 0.5 \times 10^{-3}$ sec
  - Total time = $5.83 \times 10^{-3} + 0.5 \times 10^{-3} = 6.33 \times 10^{-3}$ sec
- Samples = Time $\times$ Sampling Frequency

$(6.33 \times 10 - 3\ s) \times (312.5\ kHz) = 1978$ samples

- Data memory =
  - 1980 bytes total
  - 3 buffers of 660 bytes

A Code

A.1 Downsampling and Filter Code

The downsampling and filter code for the data acquisition example uses a sampling frequency of 2.5 MHz downsampled by eight giving an approximate eight OVS for the 40 kHz filter. The filter is a 40 kHz bandpass 400 tap equiripple FIR with a gain of one in the passband (±0.5), 4 kHz bandwidth, and down 80 dB in the stop bands.

```
//*****Include files*****
#include "prototype.h"
#include "SC140xLib_Filters.h"              //Filters Library header file

//*****Defines*****
#define BUFFER_SIZE 1980                    //Number of test in and out data
#define DELAY_TAPS   400                    //Number of delay taps
#define FILTER_IN_BUF_SZ   660              //Number of data into/out of filter

//*****Global variables*****
#include "coeff.h"                          //Coefficients
#include "ReceivedSignal40k.h"              //Test data input (real)

short gasBuffOut[BUFFER_SIZE/8];            //Test data output (real)
short gasStates[DELAY_TAPS + 4];

#pragma align gasBuffIn 8
#pragma align gasBuffOut 8
#pragma align gasStates 8
#pragma align gasCoeffs 8

/* ============================================================================
   main() Data Acquisition using SC140x Filters Library Fir_buffer function
============================================================================ */
int main()
{
unsigned short int usiI,usiJ;               //Test counter
short FilterInput[FILTER_IN_BUF_SZ];
short FilterOutput[FILTER_IN_BUF_SZ];       //I/O to Fir_buffer calling routine
short asBuffIn[BUFFER_SIZE/8];

#pragma align FilterInput 8
#pragma align FilterOutput 8
#pragma align gasBuffIn 8

stSC140xLib_FIR_IOStruct stFir_buffer;      //input structure

   //----------------------------
   //Initialization
   //----------------------------
   usiJ=0;
```

```
   for (usiI=0; usiI<DELAY_TAPS+4; usiI++)
    {
        gasStates[usiI] = 0;                //Clear States memory
    }

//Initialize Fir Input Structure
   stFir_buffer.psInBuffPtr=FilterInput;
   stFir_buffer.psOutBuffPtr=FilterOutput;
   stFir_buffer.psCoeffPtr=gasCoeffs;
   stFir_buffer.psStateVarPtr=gasStates;
   stFir_buffer.psStateVarCurrentPtr=gasStates;
   stFir_buffer.usFiltTaps=DELAY_TAPS;
   stFir_buffer.usBuffSize=FILTER_IN_BUF_SZ;

   //----------------------------
   //Call library function
   //----------------------------

for (usiI=0;usiI<BUFFER_SIZE/8/FILTER_IN_BUF_SZ;usiI++){
                                                  //Partition input into 660 size increments
       for (usiJ=0;usiJ<FILTER_IN_BUF_SZ;usiJ++)
       {
              FilterInput[usiJ]=asBuffIn[usiI*FILTER_IN_BUF_SZ+usiJ];    //Downsample by 8
       }

       SC140xLib_Filters_Fir_buffer(&stFir_buffer);//Run Filter

       for (usiJ=0;usiJ<FILTER_IN_BUF_SZ;usiJ++)   //Write output to global output buffer
       {
              gasBuffOut[usiI*FILTER_IN_BUF_SZ+usiJ]=FilterOutput[usiJ];
       }
   }

   //----------------------------
   //Return
   //----------------------------
   return(0);
}
```

A.2 Energy Detection Code

The example code for the energy detection routine is as follows:

```
_Energy_Detect

       push d6              push d7
       dosetup2 SAMP_LOOP   move.l (r0)+,r1      ; r1->indata
       bmclr #$3f,SR.l                           ; no scaling, conv rounding, no sat
       move.w (r0)+,r4                           ; r4=MaxSamples
       move.w (r0)+,r5      tfra r1,r3           ; r5=PulseLength, r3->indata
       move.l (r0)+,d1                           ; d1=Threshold1
       move.l (r0)+,d2      suba r5,r4           ; d2=Threshold2, r4=MaxSamples-PulseLength
       doen2   r4           deca r5              ; r5=PulseLength-1

       move.f (r1)+,d0                           ; get data from input vector
       mpy d0,d0,d3         tfra r1,r2           ; calc sample energy
       cmpgt d1,d3                               ; compare energy to Thresh1

       falign
SAMP_LOOP
```

```
        loopstart2
        bf NO_THRESH1                           ; if samp energy < Thresh1 skip loop
        doensh3 r5
        move.f (r2)+,d0

        falign
        loopstart3
        mac d0,d0,d3          move.f (r2)+,d0   ; calc pulse energy
        loopend3

        cmpgt d2,d3                             ; compare pulse energy to Thresh2
        nop
        ift break PULSE_FOUND                   ; if pulse energy > Thresh2 = pulse found
NO_THRESH1
        move.f (r1)+,d0                         ; get data from input vector
        mpy d0,d0,d3          tfra r1,r2        ; calc sample energy
        cmpgt d1,d3                             ; compare energy to Thresh1
        loopend2

PULSE_FOUND
        suba #2,r1
        suba r3,r1                              ; calc sample of pulse start
        lsra r1                                 ; from start of in buffer
        cmpeqa r1,r4                            ; check if sample = end of buffer
        pop d6                pop d7
        ift move.w #$FFFF,r1                    ; if end of buffer, sample = -1
        move.w r1,(r0)                          ; write output
        rtsd                  move.l (SP-4),d3
_Energy_Detect_end
        move.l d3,SR                            ; restore SR
```

A.3 Distance Computation Code

The example code for the distance computation routine is as follows:

```
{
long lDistance;
unsigned long ulDeltat;

        if (sPulseCounter > 0)
        {
                ulDeltat = (sPulseCounter * uliInvSampFreq);
                lDistance = (ulSpeedofSound * ulDeltat) >> 1;
        }
        else
        {
                lDistance = -1;
        }

        return(lDistance);
}
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN3061
Rev. 0
1/2006