Application Note: 221

Using CMSIS-DSP Algorithms with RTX

Abstract

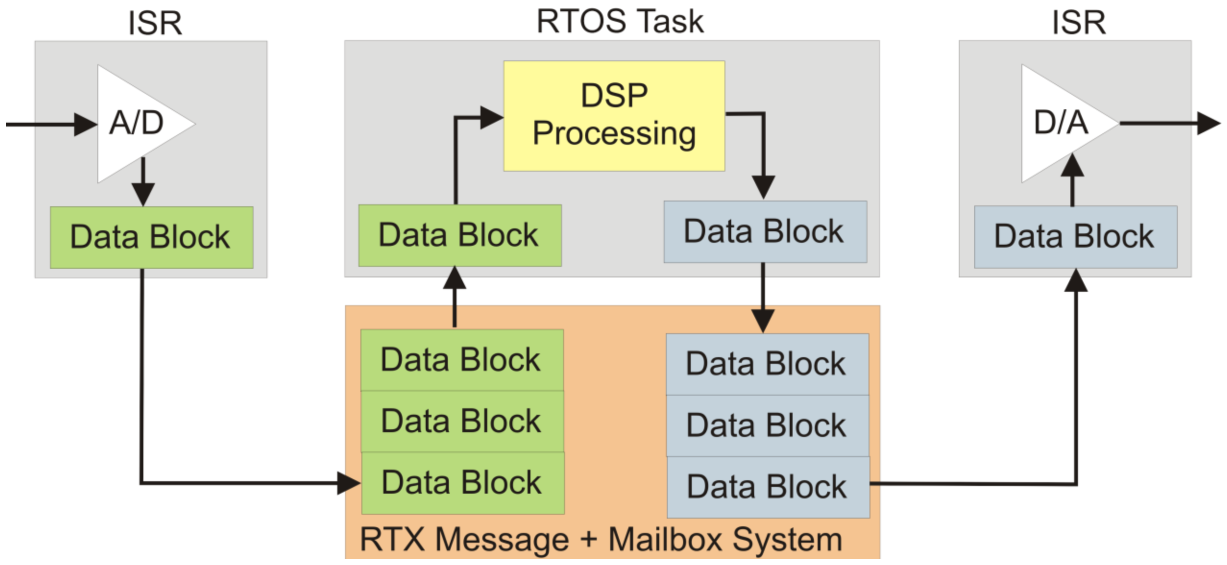
This Application Note describes the development of a digital filter for an analog input signal using the CMSIS-DSP Library and the RTX-RTOS. The application, designed for an NXP LPC1768 device, can be tested with the µVision Simulator and the Logic Analyzer. Due to the CMSIS layer, the application can be ported easily to other Cortex-M3 or Cortex-M4 processor-based devices.

Contents

Introduction

The program example of this Application Note implements a lowpass filter using the DSP block processing mode. Incoming analog signals are converted by the A/D peripheral and are collected into data blocks. These data blocks are queued in RTOS messages, which are sent to a mailbox. These mailbox messages are read by an RTOS task. The task executes the filter algorithm and queues the processed data in RTOS messages, which are sent to a second mailbox. The message data from this second mailbox are then converted back to an analog signal by the D/A peripheral. The RTOS message and mailbox system coordinates the flow of the data blocks through an Interrupt Service Routine (ISR). The CMSIS-DSP Library provides the filter algorithm used in the RTOS task. Since this example application uses the DSP block processing mode, the output signal gets delayed by a few milliseconds.

The application example uses the Keil RTX-RTOS, because this RTOS supports message passing and memory pools that interface between tasks and ISRs. The Keil RTX-RTOS is part of the Keil MDK-ARM Microcontroller Development Kit.



Prerequisites

- Keil MDK-ARM Microcontroller Development Kit.
- CMSIS-DSP Library (binary is included in this project). The library is part of CMSIS Version 2.0, which can be downloaded from **www.onarm.com**.
- A filter design tool, for example one of the *QEDesign* versions. Many other filter design tools are available on the market to generate filter coefficients.

Evaluation versions of the software above are sufficient to test and follow the Application Note example.

Program Implementation

The µVision example project **DSP_APP.UVPROJ** implements a digital lowpass filter. The filter is defined with the following characteristics:

Sampling Frequency [Hz]:	32000
Passband Edge Frequency [Hz]:	3200
Stopband Edge Frequency [Hz]:	9600
Passband Ripple [dB]:	0.1
Stopband Ripple [dB]:	60

Characteristics for the different filter types:

FIR Filter Length:	15
IIR Order Estimation:	4

Several variants for Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters are included in the project. In µVision, the filter variants can be selected through the project targets described below:

- **SIM FIR FLOAT32** - configuration for an FIR filter using the floating-point number format.
- **SIM FIR Q31** - configuration for an FIR filter using the fixed-point number format Q31.
- **SIM FIR Q15** - configuration for an FIR filter using the fixed-point number format Q15.
- **SIM IIR FLOAT32** - configuration for an IIR filter using the floating-point number format.
- **SIM IIR Q31** - configuration for an IIR filter using the fixed-point number format Q31.
- **SIM IIR Q15** - configuration for an IIR filter using the fixed-point number format Q15.

The example code that is important for processing the signals is split into different modules:

Module	Description
ADC.C	Hardware abstraction layer for the A/D converter.
DAC.C	Hardware abstraction layer for the D/A converter.
DSP_APP.C	Main application module, which initializes the timer, A/D and D/A converter, and starts the RTX kernel. The file contains the task to process the data and the ISR to handle the sampling of the input and output data.
DSP_IIR.C	Definition file for the IIR filter structure and state buffers. Copy the filter coefficients to this file. Functions to initialize the IIR data structure and functions to process the data are declared in this file.
DSP_IIR.H	Header file for IIR filters.
DSP_FIR.C	Definition file for the FIR filter structure and state buffers. Copy the filter coefficients to this file. Functions to initialize the FIR data structure and functions to process the data are declared in this file.
DSP_FIR.H	Header file for FIR filters.
ARM_CORTEXM3_MATH.LIB	The CMSIS-DSP Library, which implements the DSP algorithms.
ARM_MATH.H	Header file for ARM_CORTEXM3_MATH.LIB.

The most important functions of the main module **DSP_APP.C** are described briefly below:

- **main**: is the main function and initializes A/D and D/A converter, and starts the RTX kernel. It initializes the Timer2 to trigger at 32 kHz. The timer clock rate, #define **TimerFreq**, has to be set equal to the filter Sampling Frequency (see previous page).
- **init**: is the task that initializes the message pool **DSP_MsgPool** and the mailboxes **mbx_SigMod** and **mbx_TimIrq**. It starts the Timer2 and the task **SigMod**.
- **TIMER2_IRQHandler**: is the ISR triggered by the Timer2 interrupt. The ISR collects the values from the A/D converter and stores the data in message buffers. The ISR also sends the output data, which have been computed by the task **SigMod**, to the D/A converter.
- **SigMod**: is the RTX-RTOS task that computes the data by applying the filter function of the CMSIS-DSP Library and stores the computed values in messages for further processing. All message buffers in the program contain 256 samples. The buffers size can be adjusted through the #define **DSP_BLOCKSIZE**.

Additional modules and functions, which are not relevant to understand the DSP implementation, have not been explained in this document. They are related to device configuration settings or peripherals.

Port the Filter Coefficients to the Application

The project includes two files, **FIR_32K_filterCoeff_QED.flt** and **IIR_32K_filterCoeff_QED.flt,** containing the filter coefficients. After the filter coefficients have been generated with a filter design tool, copy the coefficients to the example application.

For FIR filter coefficients, open the file **DSP_FIR.C** in µVision and copy the coefficients to the data structure.

For IIR filter coefficients, open the file **DSP_IIR.C** in µVision and copy the coefficients to the data structure.

DSP Processing

The code snippets used in this document have been simplified for demonstration purposes. Error handling and overflow checks have been omitted. The IIR Q31filter variant is used to explain the code.

The application can be split logically into three parts:

1. Sampling input data – processed by the ISR **TIMER2_IRQHandler**.
2. Filtering the sampled data – processed by the task **SigMod**.
3. Sampling output data – processed by the ISR **TIMER2_IRQHandler**.

Using CMSIS-DSP Algorithms with RTX

The message buffers for the sampled and processed data are arrays with the size **DSP_BLOCKSIZE,** in this example set to 256. The message pool contains 10 messages. The mailboxes have been configured to store different amounts of messages.

```
/*----------------------------------------------------------------------------------
   defines & typedefs for messages
 *---------------------------------------------------------------------------------*/
typedef struct _DSP_MsgType  {
  q31_t     Sample [DSP_BLOCKSIZE];
} DSP_MsgType;

_declare_box (DSP_MsgPool, sizeof(DSP_MsgType), 10);   // Message Pool of 10 messages
os_mbx_declare(mbx_SigMod, 8);                         // declare SigMod RTX mailbox
os_mbx_declare(mbx_TimIrq, 2);                         // declare TimerIRQ RTX mailbox

DSP_MsgType *pMsgTimIrqOut;                            // message for SigMod RTX mailbox
DSP_MsgType *pMsgTimIrqIn;                             // message for TimIrq RTX mailbox
```

The ISR **TIMER2_IRQHandler** samples the incoming A/D data and converts them to the selected number format, Q31 for this example. The samples are stored in the RTX message **pMsgTimIrqOut**. After storing **DSP_BLOCKKSIZE** samples, **pMsgTimIrqOut** is sent to the mailbox **mbx_SigMod**.

The code below is part of the ISR **TIMER2_IRQHandler**.

```
// ------- signal Input Section --------------------------------------------------
adGdr = LPC_ADC->ADGDR;
if (adGdr & (1UL << 31))  {                            // Data available?

                   // Descale value and move it in positive/negative range.
                   // 12bit Ad = 0xFFF; filter in range is -1.0 < value < 1.0

  tmpFilterIn = ((float32_t)((adGdr >> 4) & 0xFFF) / (0xFFF / 2)) - 1;
  arm_float_to_q31(&tmpFilterIn, &tmp, 1);
  pMsgTimIrqOut->Sample[msgTimIrqOutIdx++] = tmp;

  if (msgTimIrqOutIdx >= DSP_BLOCKSIZE) {
    isr_mbx_send(mbx_SigMod, pMsgTimIrqOut);           // send the message
    pMsgTimIrqOut = _alloc_box (DSP_MsgPool);          // allocate a message
    msgTimIrqOutIdx = 0;
  }
}
```

The task **SigMod** waits for messages sent to the mailbox **mbx_SigMod**. After a message with the sampled A/D data has arrived, a new message for the filtered data is allocated. The filter function, defined in the CMSIS-DSP Library, is executed. The received and the newly allocated message are directly used as parameters in the filter function. On completion of the filter function, the message containing the filtered data is sent to the mailbox **mbx_TimIrq**.

The code below is part of the task **SigMod**.

```
// ------- Task 'Signal Modify' -----------------------------------------------
__task void SigMod (void) {
  uint32_t errCode;
  DSP_MsgType *pMsgIn;                                   // IN  message
  DSP_MsgType *pMsgOut;                                  // OUT message

  for (;;) {
    os_mbx_wait (mbx_SigMod, (void **)&pMsgIn, 0xFFFF);  // Wait for a message

    pMsgOut = _alloc_box (DSP_MsgPool);                  // allocate a message
    iirExec_q31 (pMsgIn->Sample, pMsgOut->Sample);       // execute filter function
    errCode = os_mbx_send (mbx_TimIrq, pMsgOut, 0xFFFF); // send message to TimIrq

    errCode =  _free_box (DSP_MsgPool, pMsgIn);          // free message memory allocation
  }
}
```

The mailbox **mbx_TimIrq** queues the messages sent from task **SigMod**. The ISR **TIMER2_IRQHandler** reads the message from the mailbox **mbx_TimIrq**. Each sample in the message is converted to a format that fits the D/A converter. Then, the data is sent to the D/A converter. After processing **DSP_BLOCKKSIZE** samples, the message memory is freed.

The code below is part of the ISR **TIMER2_IRQHandler**.

```
// ------- signal Output Section ----------------------------------------------
if (pMsgTimIrqIn == NULL)  {
  isr_mbx_receive(mbx_TimIrq, (void **)&pMsgTimIrqIn);
  msgTimIrqInIdx = 0;
}
if (pMsgTimIrqIn != NULL)  {
  tmp = pMsgTimIrqIn->Sample[msgTimIrqInIdx++];
  arm_q31_to_float(&tmp, &tmpFilterOut, 1);

                 // move value in positive range and scale it.
                 // 10bit DA = 0x3FF. Filter OUT range is -1.0 < value < 1.0
  LPC_DAC->DACR = (((uint32_t)((tmpFilterOut + 1) * (0x03FF / 2))) & 0x03FF) <<  6;

  if (msgTimIrqInIdx >= DSP_BLOCKSIZE)  {
    errCode = _free_box (DSP_MsgPool, pMsgTimIrqIn);  // free message memory
    pMsgTimIrqIn = NULL;
  }
}
```

Debug the Application

The application can be tested with the µVision Simulator and the Logic Analyzer. µVision incorporates a C-type scripting language to simulate analog input signals for the A/D converter. The VTREG **AIN2** is linked to the input pin of the A/D converter. The code to generate input signals can be found in the debug command file **DBG_SIM.INI**.

```
/*-------------------------------------------------------------------------------
  Generate mixed sine wave signal
    a   = amplitude of mixed sine wave
    f1  = frequency of sine wave 1
    f2  = frequency of sine wave 2
 *-------------------------------------------------------------------------------*/
signal void SineMix (float a, float f1, float f2)  {
  float   sin1;
  float   sin2;
  float   w;

  for (;;) {                                          // do forever

    w    = 2 * 3.1415926 * f1;
    sin1 = __sin ( ((double)STATES / CCLK) * w);
    w    = 2 * 3.1415926 * f2;
    sin2 = __sin ( ((double)STATES / CCLK) * w);

    AIN2 =  (((sin1 + sin2) * a)) + 1.6;              // set analog value
    swatch (0.00001);                                 // in 10 uSec resolution

  }                                                   // end do forever
}
```

Toolbox buttons that start and stop generating sine waves are defined in the file **DBG_SIM.INI**.

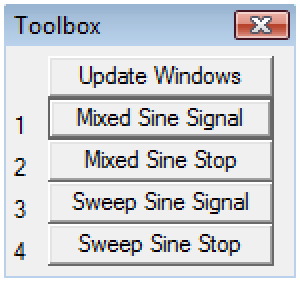
```
DEFINE BUTTON "Mixed Sine Signal", "SineMix   (1.0, 2000.0, 14000.0)"
DEFINE BUTTON "Mixed Sine Stop",   "SIGNAL KILL SineMix"
DEFINE BUTTON "Sweep Sine Signal", "SineSweep (1.0, 2500.0, 9700.0)"
DEFINE BUTTON "Sweep Sine Stop",   "SIGNAL KILL SineSweep"
```

The following symbols are used in the Logic Analyzer to test the application:

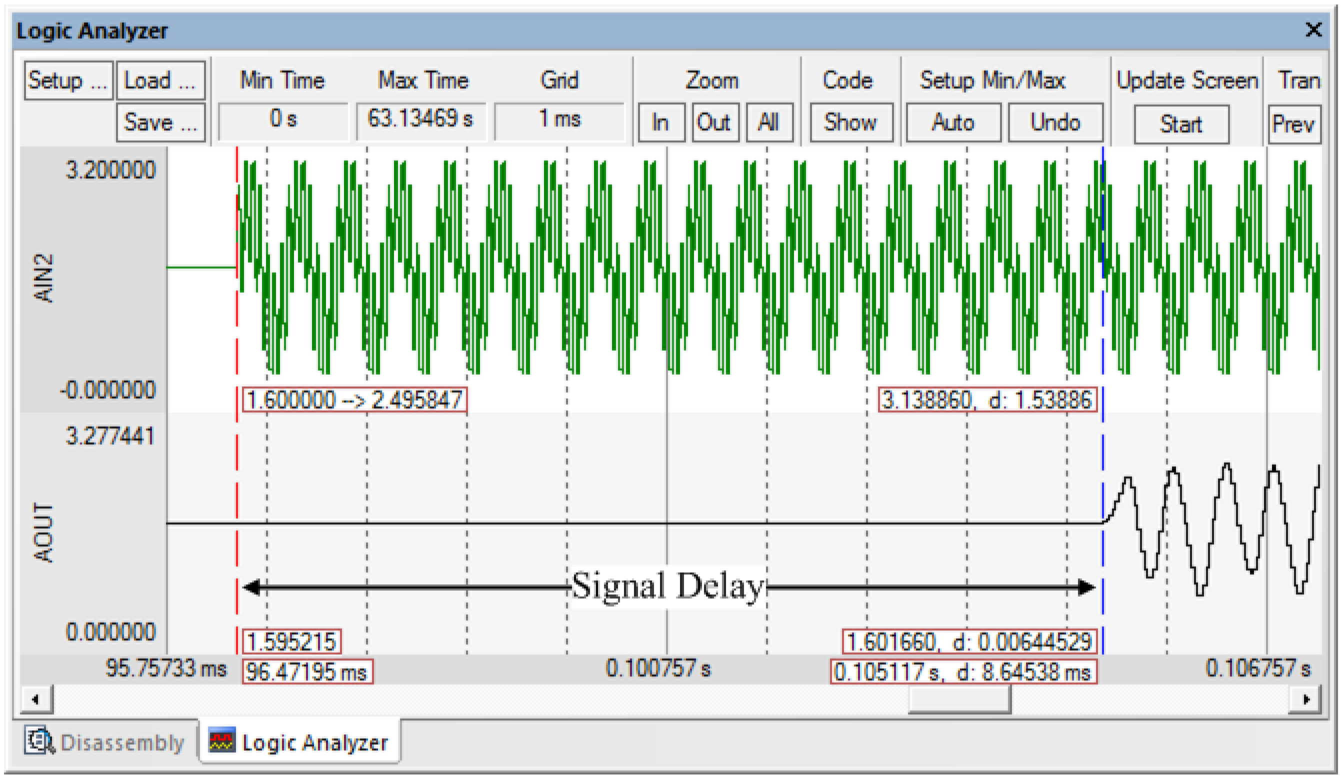
Symbol	Description
AIN2	VTREG AIN2, which is linked to the input pin of the A/D converter
AOUT	VTREG AOUT, which is linked to the output pin of the D/A converter.

To debug the program:

1. Start the µVision Debugger with **Debug – Start/Stop Debug Session**.
2. Open **View – Analysis Windows - Logic Analyzer**
3. Click **Debug – Run** to continue debugging the program.
4. Open **View – Toolbox Window** and click the buttons to start generating the sine wave signals. The output can be viewed in the Logic Analyzer.



Mixed Sine Signal output for target **SIM IIR Q31**



Conclusion

The CMSIS-DSP Library and the RTX-RTOS make the implementation of DSP algorithms straightforward. The RTX-RTOS message and mailbox system supports the DSP block processing offered by the CMSIS-DSP Library. The CMSIS-DSP Library offers a common set of DSP algorithms. Only minor code changes are required for processing data with different filter types. As a consequence, development cycles are shortened and developers can focus on the application design and requirements, without needing to implement their own, basic DSP algorithms.

Revision History

- December 2010: Initial Version