Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Autobuffering, C and FFTs on the ADSP-218x

Submitted 8/2001 by DTL

This application note presents a technique for using autobuffering when bit-reversal is enabled within the C FFT libraries. These techniques are demonstrated in a C-based data acquisition and spectral analysis system which also presents a number of programming techniques such as : setting up hardware and ports in C, setting up FFT buffers on the correct memory boundaries and double-buffering with autobuffering. There is a corresponding VisualDSP project that accompanies this application note - make sure you have downloaded this as well.

The ADSP-218x family of DSPs support serial port DMA functionality with a feature called autobuffering. Autobuffering moves received data from the RX register into memory via a pre-assigned set of DAG registers. Conversely, it moves data from memory to the TX register via another set of DAG registers. The C compiler allows us reserve register sets I2 and I3 for autobuffering. To reserve these registers, use the –reserve switch (for example, -reserve=-I2,I3) in the project options->compile->Additional Options.

The FFT functions in the C-runtime library use an addressing mode available in the DAG1 registers called bit-reversal. This addressing mode is used to scramble/unscramble FFT data which greatly increases the efficiency of the FFT itself. When this mode (bit-reversal) is enabled, all accesses to the DAG1 registers result in a bit-reversed memory access. This means that the addresses stored in the corresponding Ix register are flipped (i.e. low bits are now high bits and visa versa).

If the FFT function has enabled bit-reversal, and a new sample arrives in the RX register while autobuffering is enabled, the autobuffering DAG access will be reversed too! This means that the received word will not end up in the memory buffer that was set aside for autobuffering but rather somewhere else in DM as the destination address was bit-reversed.

Since most applications use autobuffering and bit-reversal, this can present a dilemma. This application note details a technique to avoid this problem without sacrificing performance on the DSP.

This implementation is designed to work in a C environment but can be easily adapted to work in a pure assembly environment.

Before we get into the nuts and bolts, let's discuss what it is we're going to do here. Essentially, we will be turning off autobuffering before the FFT begins and handling the function autobuffering normally handles, manually. Once the FFT function has completed, we will re-enable autobuffering. This approach is done in assembly language so the overhead of manually handling the interrupts during the FFT will be minimal (i.e. 10 instructions/received sample).

Step 1: Modifying the C Interrupt Vector Table

In the C environment, there is a file called 218x_int_tab.asm which contains the interrupt vector table for C interrupts. This file can be found in the VisualDSP\218x\lib\src\libc_src directory. We're going to need to modify this file so copy it to your project directory and include it in your VisualDSP project. Whenever you include a library source file in your project, this code will take precedence over the version stored in the C library file. This way, you can tailor certain components of the C library without having to build an entirely new library file!

Once you have included this file in your VisualDSP project, open it up. You'll see each interrupt vector which is placed in its own section using the .section command. If your LDF file does not these sections, copy them from one of the stock 218x LDF files. The first step is to ensure all of the interrupt vectors have 4 instructions. Because they are explicitly placed in memory using the C library LDF file, they weren't required to be 4 instructions long. But, the stock LDF files group all these sections to together and they will be linked contiguously so if they are not all 4 instructions long, the interrupt table will be misaligned! The short of it is this : if a vector only has 3 instructions, add an RTI; at the end as filler.

```
.section/code IVirqe;
.global ___irqe;
___irqe:
          DM(I4+=M7)=AX1;
          AX1=SIGIRQE;
          JUMP __lib_int_determiner;
          RTI;
```

The next step is to determine which of the 4 types of serial transmissions in which you'll be using autobuffering (i.e. SPORT0 Receive, SPORT0 Transmit, SPORT1 Receive, SPORT1 Transmit). Once this has been determined, locate the appropriate interrupt vectors and replace them with the following (for example):

```
.section/code IVsport0recv;
.global ___sport0recv;
___sport0recv:
          JUMP SPORT0_RX_Interrupt;
          RTI;
          RTI;
          RTI;
```

The function call name is arbitrary but you'll want a unique function call for each of the SPORT vectors you replace. In the example above, we replaced the SPORT0 Receive interrupt. Essentially what we are doing here is setting up the C environment to bypass the C interrupt dispatcher and call our own assembly interrupt.

Step 2 : The Interrupt Service Routine

Once the above has been done for all of the interrupts that you wish to use with autobuffering (which should only be 2 since you can only reserve 2 sets of DAG registers in the C environment - more on this one later), its time to set up the interrupt functions. Create a new text file - this will hold our interrupt service routines which we jump to from the interrupt vectors we wrote in step 1.

Before we get into the code, let's discuss what it is we will be doing here. When autobuffering is enabled, an interrupt is generated after the designated buffer is full and the DAG pointer wraps back to the beginning. The length of the buffer is set using the corresponding L register in the DAG. When autobuffering is not enabled, an interrupt is generated for every sample received. We have one routine for both scenarios so we need to address both in the single ISR.

The code below is fairly simple and well-commented - it should explain itself. You'll need to create a function for each of your interrupts. In our example, we're only using one interrupt so we only modified one interrupt vector in Step 1 and are now only writing a single service routine. You'll need a unique service routine for each interrupt vector.

```
/*   Include Assembly Support and 218x header files */
#include <asm_sprt.h>
#include <def2181.h>


/*   Declare the variables we declared in C as .extern's */
.extern   _Buffer1;
.extern   _Buffer0;
.extern   _db_tgl;

.extern   _Bit_Reversal_On;
.extern   _DoFFT;

/*   Declare section and function name - this must match the
     function you called in your ISR!  */
.section /pm program;
.global SPORT0_RX_Interrupt;
SPORT0_RX_Interrupt:

/*   First, enable secondary register set so we don't corrupt anything
     back in C-world */
     ena SEC_REG;

/*   Second, did we enter this interrupt because we completed an
     autobuffer wrap,    or did we enter it because we disabled
     autobuffering and a new sample came in?  This information is
     stored in the C variable, _Bit_Reversal_On.  */
```

```
ax0 = dm(_Bit_Reversal_On);
ar = pass ax0;

if EQ jump     Autobuffering_Interrupt;

Non_Autobuffering_Interrupt:
/*   At this point, we know that we are not in a standard
     autobuffering interrupt - autobuffering has been
     disabled and we will handle the work of autobuffering
     manually! */

/*   First things first - disable the helpful yet menacing bit
     reversal so we can store the data correctly */
     dis  BIT_REV;

/*   Perform the function that autobuffering normally handles */
     ax0 = rx0;
     ay1 = i2;       /* ay1 = i2 before the access */
     dm(i2,m1) = ax0;
     ax1 = i2;       /* ax1 = i2 after the access */

/*   Check to see if the circular buffer pointer wrapped around, if
     so, this would have generated an autobuffering interrupt so
     service it as such */

     ar = ax1 - ay1;     /* if ax1 < ay1, the pointer wrapped */
     if LT jump Autobuffering_Interrupt;

/*   When an interrupt comes in, the status registers are pushed onto
     the status stack.  When we return from an interrupt, these values
     are automatically restored so although we enabeld secondary
     registers and disabled bit-reversal, we don't have to set them
     back as this will automatically happen during an RTI
     instruction. */
     rti;


/*   This is the autobuffering interrupt.  In this example, we are
     handling the normal autobuffering interrupt in assembly.  If
     you need to call a C function instead, you just need to set
     up the C call correctly - see page 2-71 of the C Compiler
     & Library Manual for ADSP-218x Family DSPs - this section
     is entitle Calling C Functions From Assembly Language
     Programs */

/*   In our autobuffering ISR, we are implementing a simple double-
     buffering scheme to ping-pong between two receive buffers, i.e.
```

```
     receive data into Buffer0 and operate on Buffer1.  Then,
     receive on Buffer1 and operate on Buffer0. */
Autobuffering_Interrupt:
     ax0 = 1;
     dm(_DoFFT) = ax0;

     ax0 = dm(_db_tgl);
     ar = pass ax0;
     IF NE jump Buffer0_Use;

Buffer1_Use:
     i2 = _Buffer1;
     ar = NOT ar;
     dm(_db_tgl) = ar;
     rti;

Buffer0_Use:
     i2 = _Buffer0;
     ar = NOT ar;
     dm(_db_tgl) = ar;
     rti;
```

Step 3 : Setting it all up in C

Its now time to setup the serial ports, setup autobuffering and get the interrupt ready in C.

The first step is to setup the serial ports. Setting up the serial ports from C is quite easy using the macros found in sport.h. The serial ports are set up as memory-mapped C structures.

```
#include <sport.h>
#include <circ.h>

...

// setup 10 Mhz internal SCLK with 80Mhz clock in
sport0.sclkdiv = COMPUTE_SCLKDIV(80000000, 10000000);

// set RFS to be generated ever 20 SCLK periods
sport0.rfsdiv = 20;

// setup sport0 control register
sport0.control.single.isclk   = 1;      // internal sclk
sport0.control.single.irfs    = 1;      // Internal RFS
sport0.control.single.rfsr    = 1;      // RFS required
```

```
sport0.control.single.slen    = 15;      // 16 bit words
```

The second step is to setup autobuffering. These macros are also located in sport.h.

```
// setup autobuffering regisers
sport0.autobuffer.receive_enable   = 1;
                // enable receive autobuffering on sport0
sport0.autobuffer.rmreg            = 1;
                // M3 (=1 by default)
sport0.autobuffer.rireg            = 2;
                // I0 (we need to reserve this)

// Set up DAG register i2 to be used for autobuffering
// and to point to _Buffer0 initially with a length of
// 512
// this macro is found in circ.h
circ_setup(2, Buffer0, 512);
```

Finally, we enable the serial port and setup the interrupt. The function call inclded in the interrupt function is actually just a dummy call. Since we've replaced the interrupt vector for this interrupt, the interrupt dispatcher will never get called. We are using the interrupt function here only to unmask our interrupt.

```
// start sport 0
sport_start(0);

// this interrupt will be generated every time the DAG pointer used
for
// autobuffering wraps
interrupt(SIGSPORT0RECV, SPORT0_Receive_Int);
```

At this point, we're ready to go. Take a look at the corresponding VisualDSP project. You can simulate this by setting up a stream on SPORT0 and using the included in.dat file. In the streams dialog box, set this up (source) as a circular, hexadecimal input file. The destination will be SPORT0.

Conclusion

There are many potential modifications that can be made on the approach. To shorten the amount of time that autobuffering is handled manually, you can set and clear the Bit_Reversal_On variable within the FFT library itself. Bit reversal is not enabled during the entire FFT, just during the scrambling phase. To do this, include the FFT sources in your project and modify them.