DSP 101 Part 4:

Programming Considerations for Real-time I/O

by Noam Levine and David Skolnick

So far, this series has introduced the following topics:

- Part 1 (vol. 31-1): DSP architecture and DSP advantages over traditionally analog circuitry
- Part 2 (vol. 31-2): digital filtering concepts and DSP filtering algorithms
- Part 3 (vol. 31-3): implementation of a finite-impulse- response (FIR) filter algorithm and an overview of a demonstration hardware platform, the ADSP-2181 EZ-Kit Lite™.

Now, we look more closely at DSP programming concerns that are unique to real-time systems. This article focuses on how to develop algorithms for DSP systems with a variety of I/O interfaces.

What does "real-time" mean? In an analog system, every task is performed in "real time" with continuous signals and processing. In a digital signal-processing (DSP) system, signals are represented with sets of samples, i.e., values at discrete points in time. Thus the time for processing a given number of samples in a DSP system can have an arbitrary interpretation in "real time", depending on the sampling rate. The first article in this series introduces the concept of sampling and the Nyquist criterion—that in real-time applications, the sampling frequency must be at least twice the frequency of the highest frequency component of interest in the (analog) signal (Nyquist rate). The time between samples is referred to as the sampling interval. To consider a system as operating in "real time," all processing of a given set of data (one or more samples, depending on the algorithm) must be completed before new data arrives.

This definition of real time implies that, for a processor operating at a given clock rate, the speed and quantity of the input data determines how much processing can be applied to the data without falling behind the data stream. The idea of having a limited amount of time with which to handle data may seem odd to analog designers because this concept does not have a parallel in analog systems. In analog systems, signals are processed continuously. The only penalty in a slow system is limited frequency response. By comparison, digital systems process parts of the signal, enough for very accurate approximations, but only within a limited block of time. Figure 1 shows a comparison. Real-time DSP can be limited by the amount of data or type of processing that can be completed within the algorithm's time budget. For example, a given DSP processor handling data values sampled at, say, 48-kHz (audio signals), has less time to process those data values, including execution of all necessary tasks, than one sampling 8-kHz voice-band data.

In the filter example described earlier in this series, the input sampling rate is 8 kHz. For the DSP in the example to keep up with real-time data, all processing has to be done within a time budget of 1/(8 kHz), or 125 µs. On a 33-MHz digital signal-processor (30 ns per cycle), the time budget provides 125 µs/30 ns, or 4166 instruction cycles, to complete processing and any other required tasks.

Since there is a finite amount of time that can be budgeted to perform any given algorithm, managing time is a central part of DSP system software design. Time management strategy determines how the processor gets notified about events, influences data handling, and shapes processor communications.

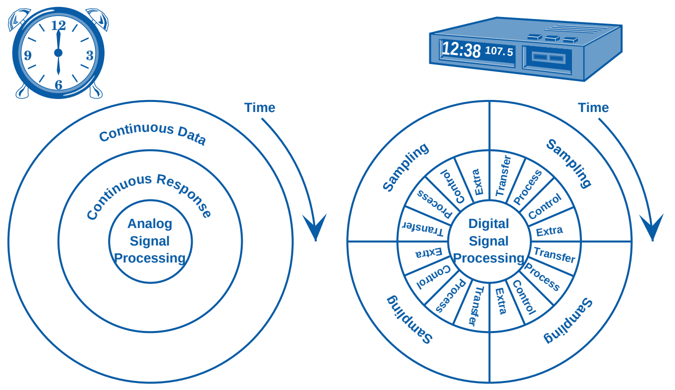

Figure 1. Comparison of analog and digital signal processing. a. Analog: A response value corresponds to each data value at all instants of time. b. Digital: For each sample, the data must be transferred in and processed, an event marks the end of processing (control), and extra time may be necessary for other tasks within the cycle after the designated process occurs.

Event Notification: Interrupts: One can program a DSP to process data using one of several strategies for handling the "event," the arrival of data. A status bit or flag pin could be read periodically to determine whether new data is available. But—"polling" wastes processor cycles. The data may arrive just after the last poll, but it can't make its presence known until the next poll. This makes it difficult to develop real-time systems.

The second strategy is for the data to *interrupt* the processor on arrival. Using interrupts to notify the processor is efficient, though not as easy to program; clock cycles can be wasted during the wait for an interrupt. Nevertheless, event-driven interrupt programming being well-suited to processing real-world signals promptly, most DSPs are designed to deal efficiently with them. In fact, they are designed to respond very quickly to interrupts. The ADSP-2181's response time to an interrupt is about three processor cycles; i.e., within 75 ns the DSP has stopped doing what it was doing and is handling the interrupt event (*vector*).

In many DSP-based systems, the interrupt rates, based on the input data sampling rate, are often totally unrelated to the DSP's clock rate. In the FIR example seen earlier in this series, the processor is interrupted at 125-µs intervals to receive new data.

Interrupt Handling and Interrupt Vectors: Because interrupt processing is such a vital element in DSP systems, processors typically have built-in hardware mechanisms to handle interrupts efficiently. Hard-wired mechanisms are more efficacious than software alone because a DSP's interrupt service routines (ISRs) may have to meet all of the following demands:

- Fast context switching—switch from working on one task and its data (a *context*) to another context without the time loss and complication associated with writing programs to save register contents and chip status information.
- Nested-interrupt handling—handle multiple interrupts of different priorities "simultaneously." The DSP handles one interrupt at a time, but an interrupt of higher priority can take precedence over the handling of a lower-priority interrupt.

- Continue to accept data/record status—while the DSP services an interrupt, events keep on occurring in the real world and data keeps on arriving. To keep up with the "real-world," the DSP must record these events and accept the data—then process them when it has finished servicing the interrupt.

On Analog Devices DSPs, fast context switching is accomplished using two sets of data registers. Only one set is active at a time, containing all the data in process during that context. When servicing an interrupt, the computer can switch from the active to the alternate set without having to temporarily save the data in memory. This facilitates rapid switching between tasks.

To handle multiple interrupts, Analog Devices DSPs record their state for each one. Processor state information is kept on a set of status "stacks" located in the DSP's Program Sequencer. A "stack" consists of a set of hardware registers. Current status information is "pushed" onto the stack when an event occurs. This stack mechanism also allows interrupts to be *nested*; one with higher priority can interrupt one with lower priority.

Two hardware features, interrupt latch and automated I/O, let Analog Devices DSPs stay abreast of the "real world" while processing an interrupt. The latch keeps the DSP from missing important events while servicing an interrupt. The other feature, comprising various forms of automated I/O (including serial ports, DMA, autobuffering, etc.) lets external devices pump data into the DSP's memory without requiring intervention from the DSP. So no data is missed while the DSP is "busy."

When an interrupt request is generated, by an external source or an internal resource, the DSP processor automatically stores its current state of operation, and prepares to execute the interrupt routine. Interrupt routines are dispatched from an interrupt vector table. An interrupt vector table is an area in Program Memory with instruction addresses assigned to particular DSP interrupt functions. For example, in the table below, a Transmit (Tx) interrupt at serial port 1 (SPORT1) of an ADSP-2181 processor will cause the next instruction to be executed at program memory (PM) location 0x0020, followed by the contents of the next three locations, through 0x0023 (the *interrupt routine*). As the 12 items in the table indicate, an ADSP-2181 can handle interrupts from 11 locations (external hardware, DMA ports, and the serial ports) and the processor Reset. The table lists the programmed instructions assigned to each interrupt vector source in memory locations 0x0000 to 0x002F for an FIR filter program.

```
Jump start; nop; nop; nop; /* PM(0x0000-03): Reset vector */
rti; nop; nop; nop;        /* PM(0x0004-07): IRQ2 vector */
rti; nop; nop; nop;        /* PM(0x0008-0B): IRQL1 vector */
rti; nop; nop; nop;        /* PM(0x000C-0F): IRQL0 vector */
ar = dm(stat_flag); ar = pass ar; if eq_rti; jump next_cmd;
                           /* PM(0x0010-13): SPORT0 Tx vector */
jump input_samples; nop; nop; nop;
                           /* PM(0x0014-17): SPORT0 Rx vector */
jump irqe; nop; nop; nop;  /* PM(0x0018-1B): IRQE vector */
rti; nop; nop; nop;        /* PM(0x001C-1F): BDMA vector */
rti; nop; nop; nop;        /* PM(0x0020-23): SPORT1 Tx vector */
rti; nop; nop; nop;        /* PM(0x0024-27): SPORT1 Rx vector */
rti; nop; nop; nop;        /* PM(0x0028-2B): Timer vector */
rti; nop; nop; nop;        /* PM(0x002C-2F): Powerdown vector */
```

Each interrupt vector has four instruction locations. Typically, these instructions will cause the processor to jump to another area of memory in order to process the data, as is shown in the Reset (at 0x0000), SPORT0 Rx (0x0014), and IRQE (0x0018) interrupt vectors. If there are just a few steps—such as reading a value, checking status, or loading memory—that can be done within the four available instruction locations, they are programmed directly, as shown in the SPORT0 Tx vector (0x0010-13). Any unused interrupt vectors call for return from interrupt (rti), with three nop (no operation) instructions.

The nop instructions serve as place holders—instruction space used to ensure that the correct interrupt action lines up with the hardware-specified interrupt vector. The rti instruction at the beginning of each unused vector location is both placeholder and safety valve. If an unused interrupt is mistakenly unmasked or inadvertently triggered, "rti" causes a return to normal execution.

Data I/O

In DSP systems, interrupts are typically generated by the arrival of data or the requirement to provide new output data. Interrupts may occur with each sample, or they may occur after a frame of data has been collected. The differences greatly influence how the DSP algorithm deals with data.

For algorithms that operate on a sample-by-sample basis, DSP software may be required to handle each incoming and outgoing data value. Each DSP serial port incorporates two data I/O registers, a *receive* register (Rx), and a *transmit* register (Tx). When a serial word is received, the port will typically generate a Receive interrupt. The processor stops what it is doing, begins executing code at the interrupt vector location, reads the incoming value from the Rx register into a processor data register, and either operates on that data value or returns to its background task. In the table above, the computer jumps to a program segment, "input_samples", performs whatever instructions are programmed in that segment, and returns from the interrupt, either directly or via a return to the interrupt vector.

To transmit data, the serial port can generate a Transmit interrupt, indicating that new data can be written to the SPORT Tx register. The DSP can then begin code execution at the SPORT Tx interrupt vector and typically transfer a value from a data register to the SPORT Tx register. If data input and output are controlled by the same sampling clock, only one interrupt is necessary. For example, if a program segment is initiated by Receive interrupt timing, new data would be read during the interrupt routine; then either the previously computed result, which is being held in a register, would be transmitted, or a new result would be computed and immediately transmitted—as the final step of the interrupt routine.

All of these mechanisms help a DSP to approach the ability to emulate what an analog system does naturally—continuously process data in real time—but with digital precision and flexibility. In addition, in an efficiently programmed digital system, spare processor cycles left between processing data sets can be used to handle other tasks.

Programming Considerations

In a "real-time" system, processing speed is of the essence. By using SPORT autobuffering, no time is lost to data I/O. Instead, the data management goal is to make sure that the selected address points to the new data.

In the FIR filter example (*Analog Dialogue* 31-3, page 15), a SPORT Receive interrupt request is generated when the input autobuffer is full, meaning that the DSP has received three data words: status, left channel data, and right channel data. Since this simplified application uses single-channel data, only the data value that resides at location rx_buf+1 is used by the algorithm.

Filter Algorithm Expansion In other applications, the data handling can be more involved. For example, if the FIR filter of the example were expanded to a two-channel implementation, the core DSP algorithm code would not have to change. The code relating to data handling, however, would have to be modified to account for a second data stream and a second set of coefficients.

In the filter code, two new buffers in memory would be required to handle both the additional data stream and the additional set of coefficients. The core filter loop may be isolated as a separate "callable" function. This technique lets the same code be used, regardless of the input data values. Benefits of this programming style include readable code, re-usable algorithms, and reduced code size. If a modular approach is not taken, the filter loop would have to be repeated, using additional DSP memory space.

The SPORT Receive interrupt routine would then consist of the setting of pointer and calling the filter. The revised filter routine is shown in the following listing:

```
Filter: cntr = taps - 1;
mr = 0, mx0 = dm(i2,m1), my0 = pm(i5,m5);
                             /* clear accumulator, get first data
                             and coefficient value */
do filt_loop until ce;       /* set-up zero-overhead loop */
filt_loop: mr = mr + mx0*my0(ss), mx0 = dm(i2,m1),
my0 = pm(i5,m5);             /* MAC and two data fetches */
mr = mr + mx0 * my0 (rnd);   /* final multiply, round to 16-bit
                             result */
if mv sat mr;                /* check for overflow*/
rts;                         /* return */
```

It's important to note that the only modifications to the core filter loop were the addition of a label, "Filter:" at the beginning of the routine, and the addition of an "rts" (return from subroutine) instruction at the end. These additions change filter code from a stand-alone routine into a subroutine that can be called from other routines. No longer a single-purpose routine, it has become a re-usable, callable subroutine.

With the core filter set up as a callable subroutine, the two-channel data handling requirements can now be addressed. To simplify some of the programming issues, this example assumes that both the left and right channels use the same filter coefficients.

In the third installment of this series, the entire filter application assembly code was displayed. At the top of the code listing, all of the required memory buffers were declared. To expand the filter application to handle two channels of data, the required new variables and buffers need to be declared. For the incoming data, the buffer declaration,

```
.var/dm/circ_filt_data[taps]; /* input data buffer */
```

would need to be replaced with two buffers, declared as

```
.var/dm/circ_filt1_data[taps]; /* left channel input data buffer */
.var/dm/circ_filt2_data[taps]; /* right channel input data buffer */
```

Because both channels are to have the same filter coefficients applied to them, the data buffers are of equal length.

The filter loop subroutine expects certain data and coefficient values to be accessed using particular address registers. Specifically, address register I2 must point to the oldest data sample, and I5 must point to the proper coefficient value prior to the filter routine being called.

Because the filters for both the left and right channel will be sharing the same memory pointers, there has to be a mechanism for differentiating the two data streams. For the data pointer, I2, two new variables need to be defined, "filter1_ptr" and "filter2_ptr." These locations in memory are going to be used to store address values appropriate for each data stream. The circular buffering capability of the ADSP-2181 is used to ensure that the data pointer is always in the correct place in the buffer whenever the filter is executed. Because the subroutine is now dealing with two buffers, the pointer locations need to be saved when processing for each channel is completed.

To set up the pointers, two variables in data memory need to be declared as follows:

```
.var/dm filter1_ptr;   /* data pointer for left channel data */
.var/dm filter2_ptr;   /* data pointer for right channel data */
```

These variable then need to be initialized with the starting address of each of the data buffers;

```
.init filter1_ptr: ^filt1_data; /* initialize starting point,
                                left channel */
.init filter2_ptr: ^filt2_data; /* initialize starting point,
                                right channel */
```

The DSP assembler software recognizes the symbol "^" to mean "address of." The DSP linker software fills in the appropriate address value. In this way, the pointer variables in the executable program are initialized with the starting addresses of the appropriate memory buffers.

The following listing shows how the FIR Filter interrupt routine uses these new memory elements. The original Filter subroutine from the 3rd installment has been modified to provide two separate channels of filtering. Instead of launching directly into the filter calculation, the routine must first load the appropriate data pointer. The filter routine is then called, and the resulting output is placed in the correct location for transmission.

```
/* ---------------------  FIR Filter ----------------------*/
input_samples:
   ena sec_reg;           /* use shadow register bank */

/* set up for filter 1 */
i2 = dm(filter1_ptr);     /* set data pointer for filter 1 */
ax0 = dm(rx_buf + 1);     /* read left channel data */
dm(i2,m1) = ax0;          /* write new data into delay line,
                          pointer now pointing to oldest data */

call filter;              /* perform the first filter for left
                          channel data */

dm(tx_buf+1) = mr1;       /* write left-channel output data */
dm(filter1_ptr) = i2;     /* save updated filter1 data pointer */

/* set up for filter 2 */
i2 = dm(filter2_ptr);     /* set data pointer for filter 2 */
ax0 = dm(rx_buf + 2);     /* read right channel data */
dm(i2,m1) = ax0;          /* write new data into delay line,
                          pointer now pointing to oldest data */

call filter;              /* perform the filter again for the
                          right channel data */

dm(tx_buf+2) = mr1;       /* write right channel output data */
dm(filter2_ptr) = i2;     /* save updated filter2 data pointer */

rti;                      /* return from interrupt */
```

Because the core filter algorithm no longer handles data I/O, this subroutine can be expanded to more channels of filtering by merely adding more pointer variables and declaring more buffer space (as long as sufficient memory exists!) Similarly, different coefficients can be used for the two filters by setting up variables that contain coefficient-buffer pointer information. In either case, the filter algorithm does not need to be altered. By using this style of modular programming, the user can build up a library of callable

DSP functions. Differences for particular systems can thus be reduced to data-handling issues rather than the development of new algorithms. While this programming style does not necessarily allow the algorithm to perform its task more quickly, the system designer has more flexibility in establishing how data flows through the system.

Real-Time Interface Issues: So far, we have examined how real-time programming in embedded systems relies on rapid interrupt response, efficient data handling, and fast program execution. In addition, the flow of data into and out of the processor also influences how well the system will work in a real-time embedded environment.

The primary data flows into and out of a digital signal processor can be both parallel and serial. Parallel transfers are typically at least as wide as the native data word of the processor's architecture (16 bits for an ADSP-2100 Family processor, 32 bits for the SHARC®). Parallel transfers occur via the external memory bus or external host interface bus of the processor. Serial data transfers require considerably fewer interconnections; they are frequently used to communicate with data converters.

Serial Interface: Ease of hardware interfacing is an important element of efficient DSP system implementation. The ADSP-2181 EZ-Kit Lite system uses an AD1847 serial codec (COder/DECoder). Serial codecs permit data transfers via a serial port (SPORT) on the DSP. This serial port is not an RS-232 PC-style asynchronous serial port; it is a 5-wire synchronous interface that passes bit-clock, Receive-data, Transmit-data, and frame-synchronization signals. Major benefits of serial interfaces are low pin count and ease of hardware hookup. The AD1847 requires only 4 signals to interface to the DSP: serial clock, Receive data, Transmit data, and Receive frame-synchronization signal. The serial data stream is time-division multiplexed (TDM), meaning that the same physical line can carry more than one type of information in serial order. In the case of the AD1847 application on EZ-Kit Lite, initiated in the last issue, the serial line carries both left- and right-channel audio information, along with codec control and status information. As noted earlier, the processor has various means for handling this data. SPORT Interrupts are generated automatically by the serial port hardware for either Receive or Transmit data and for either a single word or a block of words (Figure 2).

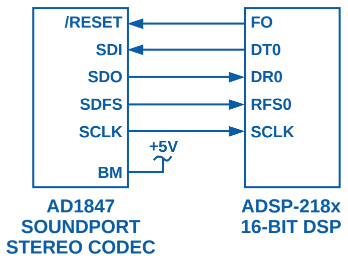

Figure 2. Serial interfacing between digital signal processor and I/O device.

Parallel Interface: Even with a serial bit clock running as fast as the DSP processor, a serial interface trades data transfer speed for simplicity of wiring, transferring a data word at a fraction of the DSP processor speed. For system performance that requires higher data rates, a parallel interface can be used. When interfacing in parallel, the DSP exercises its external data and address busses to read or write data to a peripheral device. On the ADSP-2181, the buses can interface with up to 16 bits of data.

Parallel data transfer is always faster than serial transfers. The DSP can perform an external access every processor cycle, but this requires really fast parallel peripherals that can keep up with it, such as fast SRAM chips. Parallel data transfers with other entities usually occur at less than one per processor cycle.

Interrupt handling is different for the serial and parallel interfaces. Since the external data bus of the DSP processor is a general-purpose entity handling all sorts of data, it does not have dedicated signal lines for interrupt generation and control; however, other DSP resources are available. On the ADSP-2181, several external hardware interrupt lines, such as the one for I/O memory select, are available for triggering by an external device, such as an A/D converter or codec. Such an interface is shown in Figure 3, involving a parallel device and the ADSP-2181 DSP.

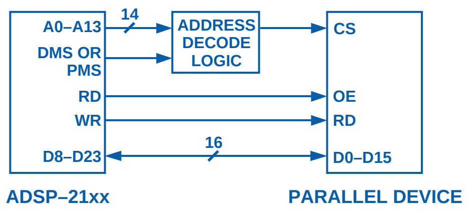

Figure 3. Parallel I/O interfacing for a DSP.

When responding to the interrupt for parallel data, the processor reads the appropriate source and typically places that data value in memory, by executing instructions similar to those shown here:

```
irq2_svc: ax0 = IO(ad_converter); dm(i2,m1) = ax0; rti;
```

"ad_converter" is a previously defined address in I/O space.

REVIEW AND PREVIEW

The goal of this article has been to detail the programming concerns that DSP developers face when handling I/O and other events in real-time systems. Issues introduced include real-time data (samples and frames), interrupts and interrupt-handling, automated I/O, and generalizing routines to make callable subroutines. This brief article could not do justice to the many levels of detail associated with each of these topics. Further information is available in the references below. Future topics in this series will continue to build on this application. The next article will add more features to our growing example program and describe software validation (i.e., debugging) techniques.

REFERENCES

ADSP-2100 Family Assembler Tools & Simulator Manual. Consult your local Analog Devices Sales Office.
ADSP-2100 Family User's Manual. Analog Devices. Free.

Many valuable publications can be found in the Design Support area of our Web site under Product Documentation. A useful bookmark is:

http://www.analog.com/support/product_documentation/dsp_prdoc.html