

# Designing a DSP System

Doc #1.0

## Application Note

**D. SignT**

Digital Signalprocessing Technology

Norbert Nölker & Adolf Klemenz GbR

Gelderner Straße 36

D - 47647 Kerken

phone +49 (0) 2833 / 570 977

fax +49 (0) 2833 / 33 28

email [info@dsignt.de](mailto:info@dsignt.de)

www <http://www.dsignt.de>

## **Designing a DSP System Revision History**

1.0 August 1999 - initial release

---

<b>1 INTRODUCTION</b> .....	<b>4</b>
<b>2 DATA ACQUISITION</b> .....	<b>5</b>
2.1 Serial Interface .....	5
2.2 Parallel Interface.....	7
2.3 Which maximum Data Acquisition Rate is possible ? .....	8
2.3.1 Interrupts .....	8
2.3.2 Busy-Polling.....	11
2.3.3 DMA .....	13
2.3.4 Still too slow ?.....	15
<b>3 DATA OUTPUT</b> .....	<b>17</b>
<b>4 CONTROL INTERFACE</b> .....	<b>18</b>
4.1 Host Interface .....	18
4.2 Asynchronous Communication.....	19
4.3 Bit-programmable I/O Ports .....	19
4.4 Sensors .....	20
4.5 Networks.....	20
4.6 Other I/O Interfaces .....	20
4.7 Control Software .....	21
<b>5 DSP AND MEMORY</b> .....	<b>23</b>
5.1 Memory Organisation.....	23
5.2 Operational Units .....	24
5.3 Data Format.....	26
5.4 Arithmetic .....	27
5.5 Built-in Peripherals.....	27
5.6 Benchmarks.....	28

---

<b>6 COMPILER AND CODE DEVELOPMENT .....</b>	<b>30</b>
6.1 Assembly Language Modules.....	31
6.2 The Influence of the Linker on Execution Performance .....	31
6.3 Real-Time Operating Systems .....	32
6.4 Debugging .....	33
<b>7 CONCLUSION .....</b>	<b>34</b>

## **1 Introduction**

DSPs provide unbeatable performance in processing real-world data. Architecture is optimised to sustain high transfer rates and single-cycle execution of the fundamental signal processing operations. No other processors exist which offer these performance levels at comparable system cost and power requirements.

Signal processing algorithms are the dominating parts of most DSP applications, however lots of secondary tasks have to be executed:

- Data must be acquired and/or output
- Signal processing is controlled by environmental conditions and user settings, thus communication with host systems, user interfaces or sensors is required.
- The results of signal processing must be evaluated and transmitted to other systems for further processing or visualisation or they are used to control actors and machinery.

The impact these secondary tasks have on signal processing execution time is often neglected. If you take a look at typical benchmark code provided by the DSP manufacturers, you will find data readily available in the DSP memory and often all data busses fully occupied by the signal processing data movements. This is far away from reality: In parallel with signal processing, you will have to read A/D converters and store the samples in an acquisition buffer, sample I/O ports, and read or write host controllers or communication devices. These tasks will often cause interrupts, bus locks or bus collisions.

The subject of this application note is to describe typical interface techniques for data acquisition and control, the associated software, and discussion of the influence on DSP operation speed. We will try to clarify dependencies and peculiarities of the different DSP architectures which are not apparent after a first glance at the data sheets. Most areas from data acquisition via control tasks up to code generation will be covered. We will however not discuss typical signal processing algorithms as there is lots of excellent literature available about it, instead we will try to draw your attention to the interaction of signal processing and framework tasks.

## 2 Data Acquisition

Most DSP systems will be used to process some kind of real-world data, either acquired by A/D converters directly connected to the DSP systems or transmitted digitally via point-to-point links or networks.

### 2.1 Serial Interface

The first choice to connect the incoming data stream to the DSP is the synchronous serial port, a built-in peripheral on all general purpose DSPs. Benefits of the serial interface are not only a reduced pin count and less glue-logic, the main advantage is independence from external bus operations. The serial interface operates completely 'on it's own', whereas data acquisition via a parallel bus interface may conflict with other bus transactions.

A serial interface is typically 'calmer' than a parallel interface, no interference from other bus transactions is radiated to the data acquisition devices, and it is easier to de-couple and isolate.

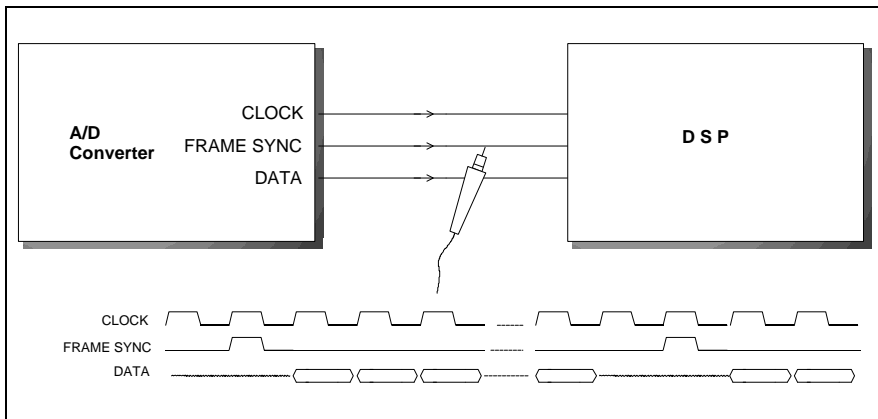


Figure 2-1 Serial Interface

All general purpose DSPs offer at least one serial interface which can be configured via control registers to meet the peripheral's requirements. Polarity, direc-

tion, frame sync type and timing, and data format can be adjusted. Industry standard converters and Codecs can be connected without additional glue logic. You will often find the serial port configuration settings for popular DSPs in the converter's application notes.

Many DSPs offer a multi-channel support for time division multiplex interfaces, like the T1/E1 format. Only pre-selected timeslots will be read to reduce data transfer and relieve the DSP core.

Clocks and frame synchronisation can be generated internally, derived from the DSP system clock, or provided externally, i.e. the serial interface can be operated in master and slave mode. The maximum clock frequency of the serial port limits this interface technique to sampling frequencies below 500 kHz, yielding in a serial clock frequency of 8 MHz for a 16 Bit / 500 KSPS converter. Although modern DSPs are capable of much higher serial clocks, serial interface converters with higher sampling rates are rare. You may however consider to serialise a parallel converter's data via shift registers, especially if opto-isolation is required. In such a case you may use the maximum serial clock of the DSP, which might be as high as 100 MHz on a Texas Instruments C6201 DSP, suitable for a 16 Bit 6.25 MHz data acquisition.

If possible, use serial clocks and framesyncs generated by the DSP itself. This will provide a synchronisation of the multiple frequencies present in the system, and prevent beat frequencies and modulation products which might degrade your system's EMC.

Data transfer from the serial port to the DSP memory can be accomplished by the DSP core itself in busy-polling or interrupt mode, or by DMA. DMA is the preferred choice, because it greatly relieves the DSP core. Please refer to section 2.3 for a comparison of these modes. Most DSPs feature separate internal buses for their built-in peripherals, accessing them will not cause bus conflicts or stalls.

## 2.2 Parallel Interface

High-speed data acquisition usually requires the use of a parallel interface because of the clock limitations of serial interfaces. Because modern DSPs typically provide only one external bus interface, care must be taken to avoid conflicts between the data acquisition and other peripherals or memories connected to the external bus.

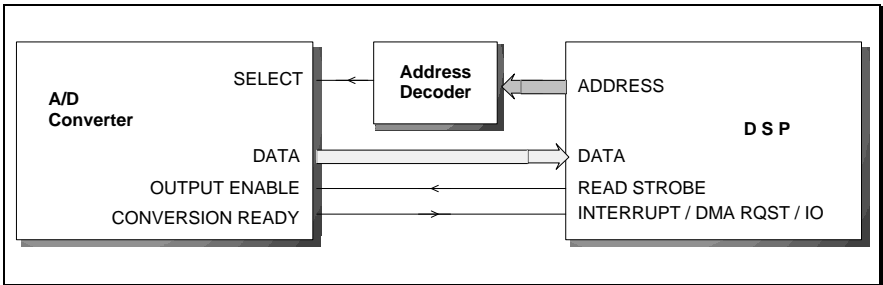


Figure 2-2 Parallel Interface

Let's take a look at a typical scenario: A high-speed ADC operating at 20 MHz sampling frequency should be interfaced via the parallel external bus. Every 50 nsecs, a transfer must occur, otherwise data will be lost. Typical DSP bus cycles are about 30 nsecs, so reading the A/D converter (via DMA) should not present too much problems. However, if the acquisition buffer is located in external memory, two memory accesses must be made in 50 nsecs ! It is a common practice to locate memory and peripherals in different memory banks to simplify address decoding and individually adjust the bus timing. However, often idle cycles are inserted by the DSP's external bus controller if a memory bank boundary is crossed. In the setup described above, an additional idle cycle before or after the converter is read would cause data acquisition to fail! If data acquisition needs not to be continuous, you may buffer the ADC data stream in a FIFO memory.

In the setup described above, you must also make sure that no other peripheral will lock the external bus too long. Imagine a data transfer to a UART with 100 nsecs or more access time: at least one or two ADC samples will be lost!

If ADC data is acquired in busy-polling or interrupt mode, these calculations should be made for less demanding sampling frequencies as well. Even if a 1 MHz A/D converter must be regularly serviced by the DSP core, no other task or interrupt must run longer than 1  $\mu$ sec. On a DSP with 20 nsecs instruction cycle time this is equal to processing 50 instructions. An interrupt service routine, e.g. to read data from a UART and perform flow control, might easily use that much instructions. If possible, assign the highest priority interrupt to the A/D converter and enable interrupt nesting, so the ADC may interrupt any other interrupt service currently executed.

DMA transfers can help a lot to relieve constraints. If the converter is serviced by DMA, only the bus usage by memory and other peripherals must be observed.

This directly leads to the question:

## 2.3 Which maximum Data Acquisition Rate is possible ?

The following examples assume an A/D converter connected either via a serial or parallel interface. Code samples are simplified because no instructions for buffer maintenance (range checking, wrap around in circular buffers) are included. Real-world code typically requires additional instructions, so the results should be regarded as best-case!

### 2.3.1 Interrupts

An interrupt service routine (ISR) is used to read the converter and store the samples in an acquisition buffer. The interrupt is triggered by the ADC's conversion ready output.

Written in C, the ISR code may be similar to:

```
void ad_interrupt ( void )
{
    buffer [index++] = *(int *)ADC_ADDRESS;
}
```

This looks pretty small and efficient, but take a look at what happens inside the DSP: If the interrupt occurs, first the code stored in the DSP execution pipeline

will be flushed, the current program counter is saved on the system stack, then the program counter will be loaded with a value from the interrupt vector table - the start address of the ISR. Next a context save is required, i.e. the current processor status and all registers which are modified in the ISR must be pushed onto the stack. Then the ISR code is executed, the context is restored and the program counter is loaded with the value on top of the stack. The execution pipeline has to be filled again, finally processing is continued where it had been interrupted.

Written in 'pseudo-code', the ISR will look like this:

```
void ad_interrupt ( void )
{
    save context (status register, registers used in ISR);
    load address register with 'buffer' start address;
    load address register with global constant 'ADC_ADDRESS';
    load index register with global variable 'index';
    load general purpose register with data from ADC;
    store general purpose register at buffer+index;
    increment index;
    store the updated index register in global variable;
    restore context;
    return from interrupt;
}
```

As you can see, two address registers, an index register and a general purpose register are modified inside the ISR. Including a status register save and restore, context save and restore will require 5 instructions each. There is no chance for parallel execution except auto-incrementing the index, so the ISR will require 17 instructions. We still have to add several cycles for pipeline flush and re-fill and interrupt vector fetch, resulting in a typical runtime of approximately 24 cycles. A 'model' DSP with 20 nsecs instruction cycle will require  $24 * 20\text{nsecs} = 480$  nsecs. for interrupt processing, which will limit the reasonable data acquisition rate to 1M samples per second - at least there should remain some time to process the data.

Remember, this is a best-case calculation! If index-boundaries have to be checked, execution time of the ISR will further increase. If any memory stall should occur, probably because the system stack and the program memory are located in the same memory bank, the execution time for context save and re-

store will double. The calculation above assumes an op-code fetch is possible in parallel with every load/store instruction.

Also note that this result cannot be scaled linear with processor speed. A high-speed DSP like the Texas Instruments C6000 has an instruction cycle time of 5 nsecs only, but the execution pipeline is much longer, more cycles will be wasted by pipeline flush and re-fill. Also load instructions on the C6000 have a 4 cycle latency before the data is valid in the register. The C6000 compiler will try to interleave and re-arrange instructions to fill most of these 'delay-slots' with useful instructions, but this is rarely possible on such short program fragments like the ISR. Also more registers are required for interleaving, resulting in longer context save and restore time. If external memory is addressed, the execution time will be dominated by the external bus access time, which often needs to be much longer than 5 nsecs.

Some C compilers do not track the register usage inside the ISR and perform a partial or complete context save and restore. An example is the Analog Devices C Compiler for the Sharc<sup>®</sup> series: You may select between three interrupt modes, the fastest one still requiring 30 cycles overhead. The Texas Instruments C Compiler do track the register usage and save and restore only those registers which are used inside the ISR. Make sure not to call any function inside the ISR, the compiler then can no longer track register usage and a complete context save and restore will be done.

Speeding up interrupt processing by hand-coding in Assembler will not yield too much improvements unless you can reserve address and index registers for exclusive use in the ISR.

Often some DSP instructions are not interruptable, like single cycle hardware loops. A typical instruction that may be assembled as a single cycle hardware loop by the compiler / optimiser may be:

```
for (i=0; i<BUFFERSIZE; i++) buffer[i] = 0;
```

*Do not use interrupt driven data transfer for data acquisition above 1 MHz. Make sure the ADC interrupt has highest priority and nesting is enabled so it can interrupt a currently executed slow interrupt, e.g. a communications routine.*

### 2.3.2 Busy-Polling

Busy-Polling techniques may be used for block-wise, non-continuous data acquisition and simple algorithms like filters. The ADC 'conversion ready' signal is connected to an I/O port and sampled:

```
void main ( void )
{
    for (;;)
    {
        while (!(*(int*)ADC_STATUS_PORT & CONV_READY_FLAG)) ;
        buffer [index++] = *(int*)ADC_ADDRESS;
        signal_processing ();
    }
}
```

The resulting pseudo-code is:

```
void main ( void )
{
    outer_loop:
    load address register with constant ADC_STATUS_PORT;
    inner loop:
    load general purpose register from ADC_STATUS PORT;
    calculate general purpose reg. AND bitmask CONV_READY;
    if result = zero jump inner_loop;

    load address register with constant ADC_ADDRESS;
    load address register with 'buffer' start address;
    load index register with global variable 'index';
    load general purpose register with data from ADC;
    store general purpose register at buffer+index;
    increment index;
    store the updated index register in global variable;

    call signal_processing;

    jump outer loop;
}
```

The jump or branch instructions flush the pipeline and typically require 3 cycles. The resulting time spend inside the loops is 15 cycles (best case), corresponding to 300 nsecs on our model DSP. Assuming 500 nsecs are used for processing, data acquisition speed may be up to 1.25 MHz - not too much an improvement compared to interrupt processing. Some processors offer bit-test instructions or I/O ports whose state can be used as a condition for the branch instruction and thus save a few cycles. Note that these special instructions may not be supported by the C compiler and will require the use of inline assembly statements.

Busy polling yields better results if data acquisition is done block-wise. In this case, addresses and indices remain in registers and do not require a re-load each loop. In this case, data acquisition is non-continuous: no data is acquired during signal-processing:

```
void main ( void )
{
    for (;;)
    {
        /* Data Acquisition Loop */
        for (index=0; index<BLOCKSIZE; index++)
        {
            while (!(*(int*)ADC_STATUS_PORT & CONV_READY_FLAG) );
            buffer [index] = *(int*)ADC_ADDRESS;
        }

        signal_processing();
    }
}
```

Because the address and index registers have to be loaded only once, the resulting execution time of the for data acquisition loop is 7 cycles best-case only, suitable for 7 MHz acquisition speed on our model DSP.

*Busy-Polling is useful for block-wise, non-continuous data acquisition at high speeds. Make sure that no interrupts occur during polling and no DMA transfer locks the external bus too long if busy-polling is driven to it's limits.*

### 2.3.3 DMA

Most DSPs offer a DMA Controller that can be programmed to transfer data to and from memory to memory, or a built-in peripheral to memory without DSP core involvement. The DMA event is triggered by either an external DMA Request signal or by an internal status change, e.g. a serial port receiver full condition.

The DMA controller can automatically increment addresses, sophisticated implementations even maintain buffer structures like ping-pong or circular buffers on their own, some can even be used for sorting and two-dimensional DMA without CPU involvement. Auto-initialisation is required for these advanced features.

Because the DSP core is not involved in data transfer, the only acquisition speed limitation is the maximum serial bit rate, respectively the external bus interface speed.

The signal processing needs to be synchronised to the incoming data stream, this can be accomplished by using 'DMA complete' interrupts or polling of the DMA status. Best results are achieved in block mode. Because the DMA continues to acquire data during signal processing, data acquisition may be continuous:

```
void main ( void )
{
    DMA_source = ADC_ADDRESS;
    DMA_destination = buffer;
    DMA_source_increment = 0;
    DMA_destination_increment = 1;
    DMA_count = BLOCKSIZE;
    DMA_auto_initialise = true;
    DMA_start;

    for (;;)
    {
        while (! dma_complete) ;
        signal_processing ();
    }
}
```

If the DMA controller lacks an auto-initialisation, this has to be performed by the DSP core:

```
while (! dma_complete) ;  
DMA_destination = buffer;  
DMA_count = BLOCKSIZE;  
DMA_restart;
```

*The speed of DMA driven data acquisition is limited by the maximum serial bit rate, respectively the external bus speed only. A DSP with a 30 nsecs external bus cycle time may be used for data acquisition up to 30 MHz if the buffer is located in internal memory. If the DMA access conflicts with other external bus transactions, speed may be lower, e.g. 15 MHz if the buffer is located in external memory too.*

*Carefully check for bus idle cycles and possible bus conflicts to determine the maximum speed.*

As a result, use DMA wherever applicable. Even if acquisition speed is not too demanding, DMA will save plenty of DSP core processing time. This may allow to lower the system clock for power-saving, or give you some spare processing-time for future enhancements.

### 2.3.4 Still too slow ?

Applications exist, where acquisition speeds above 30 MHz are desired. In these cases, the DSP will require some hardware assistance to manage data acquisition. Several techniques exist:

- You may include a hard-wired data acquisition and pre-processing, e.g. in a FPGA, to reduce the sampling frequency before the data is passed to the DSP for final processing.
- In some cases a FIFO memory may be used to buffer the incoming data stream and allow block-read operations. This is especially true if idle cycles or bus conflicts may limit acquisition speed.
- You may use an external ping-pong buffer, controlled by external hardware. The ADC is connected to bank 1 of the buffer, a counter generates incrementing addresses to store the data. Bank 2 is connected to the DSP for processing. If processing and acquisition is complete, the banks are exchanged. Crossbar switches or bus multiplexers can be used for bank-switching.

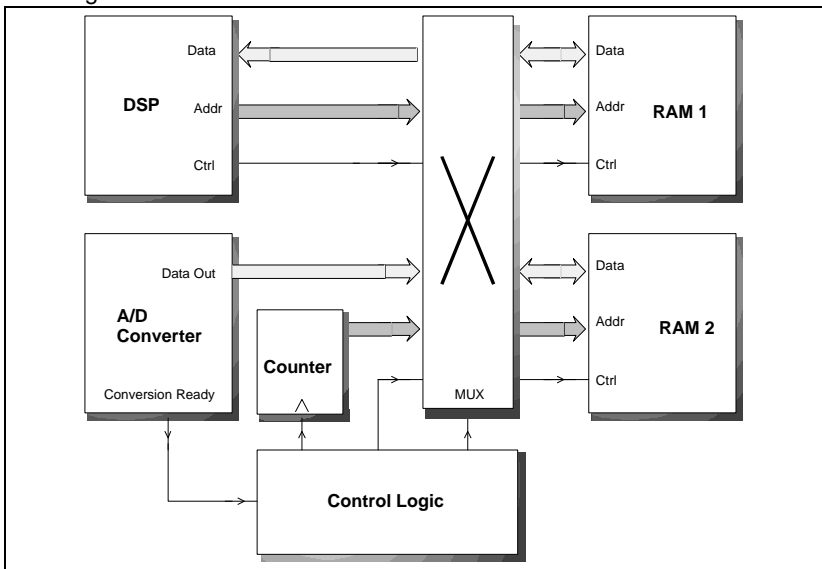
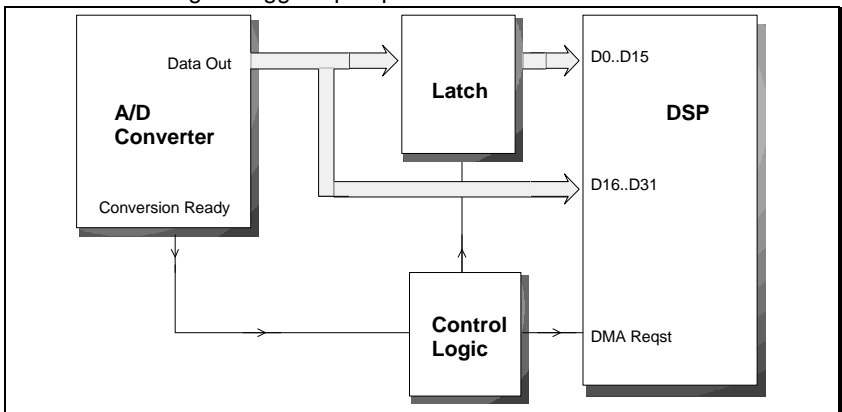


Figure 2-3 External Ping-Pong Buffer

- If multiple converters are used in synchronous sampling mode, you may connect them to a single address and different data bus lines. On a DSP with a 32 bit external bus, you may connect a 16 bit converter to D0..D15 and another one to the same address, D16..D31. Reading both converters will be done in a single cycle. If acquisition speed of both converters may vary, you will need to generate a common 'conversion ready' signal by AND-ing both signals.
- If a single converter with 16 bits or less is used, you may double the transfer rate if every second sample is latched and connected to the upper 16 bits of the DSP's 32 bit data bus. The DSP then reads sample<sub>(N)</sub> on D0..D15 and sample<sub>(N-1)</sub> on D16..D31 in a single bus cycle. The conversion ready signal must be fed through a toggle flip-flop in this case.



*Figure 2-4 Latched Samples*

- Typically, the converter's data width is less than the DSP data bus width. If the ADC uses 2's complement data, connecting it to the DSP's upper data bits will save additional shift or sign extension instructions, i.e. a 2's complement 12 bit converter should be connected to D31..D20. The unconnected bits of the data bus will be undefined if the converter is read. To save masking operations, these bits may be driven low by an additional bus driver. Pull-Down resistors cannot be used because they must be as small as a few hundred ohms to un-load a 50pF bus capacitance fast enough, which will violate the DSP's bus driving capabilities.

### **3 Data Output**

The results of signal processing may be output via D/A converters, passed to other systems for further processing or visualisation, or control any kind of machinery via I/O ports, field busses etc.

For D/A converters, the same interface techniques, data transfer mechanisms and restrictions as described in the Data Acquisition section apply. In some cases, the DSP's built-in timer(s) can be used to generate a PWM output - a low cost D/A solution.

Other output and communication interfaces are described in the following section.

## **4 Control Interface**

Very few DSP systems can operate without external control. Most applications require to pass process control information to the DSP and/or evaluate the results of signal processing. This may be accomplished by a direct connection of a user interface to the DSP (keypad, display), via serial and parallel connections (RS232, 422, SCSI), networks (Ethernet, CAN), or a direct bus interface to a micro-controller or host-system. The choice is usually dictated by the requirements of data throughput, noise immunity and local distance of the systems.

### **4.1 Host Interface**

If a host-controller is located close to the DSP system, you may prefer to use a DSP with a built-in host interface. Two topologies exist: either the host interface uses it's own independent address, data and control signals, or the host may gain control over the DSP's external bus.

The Texas Instruments C5000 and C6000 series for example use an independent host interface. The host can setup DMA transfers to and from DSP memory. Data is transferred via set of host port registers. The de-coupled interface ensures that a slow host will not slow down DSP operation.

Analog Devices Sharc<sup>®</sup> processors include a bus arbitration logic which enables the host to gain control over the DSP external bus interface. The host may initiate DMA transfers from the DSP internal memory via a set of buffer registers or directly access external peripherals and memories. Because the external bus control is passed to the host, a slow host may lock the bus quite long, preventing the DSP from further operation if external memories or peripherals should be accessed.

If the DSP lacks a host interface you must attach a dual-ported memory or a bi-directional latch and control logic to the DSP external bus. An interface with individual transmit and receive buffers plus interrupt / DMA control can be implemented in a small CPLD. Compared to discrete logic, CPLD or FPGA implementations will simplify matching the set-up and hold timing requirements to the DSP bus timing. The latch or dual ported memory de-couples the host and DSP busses and avoid bus locks caused by slow host processors. Transfers may be

DMA or interrupt driven, however, the host cannot initiate DMA transfers as it is possible on DSPs with a built-in host port. A communication protocol must be established to instruct the DSP to setup the requested transfers.

If host communication is not too demanding, a serial interface may be used. Many micros feature a SPI interface which allows data transfers at several MBits/sec. SPI is based on a shift register with parallel load and straightforward to implement in a small CPLD or FPGA.

## 4.2 Asynchronous Communication

A UART can be connected to the DSP for standard asynchronous communication. Typically the DSP bus timing and signals do not match the UART's bus interface, so some glue logic will be required. Only very few DSPs feature a built-in UART. The external UART is prone to cause problems with high speed data acquisition or extensive external memory accesses because of its typically slow access times. Sophisticated UARTs are available with large receive and transmit FIFOs, DMA support and RTS/CTS or Xon/Xoff auto flow control. These features greatly reduce protocol overhead and may save valuable computation time.

For less demanding requirements, the UART can be software-emulated by using the DSP I/O ports and timers. We have implemented this on a Texas Instruments TMS320C31 DSP. At 9600 baud, the UART functions require less than 3% of the DSP processing time, comparable to operating the DSP at 58 MHz instead of its nominal 60 MHz.

## 4.3 Bit-programmable I/O Ports

are used for a direct connection of a user interface. Reading the status of switches, keypads or driving LEDs or LCD interfaces is a common use for I/O ports. Also reading the status of external peripherals, e.g. an ADC's conversion ready signal, is done via I/O. DSPs typically do not have sufficient I/O ports, so additional ports have to be connected to the external bus. Provided that address decoding is fast, latches and buffers may be accessed without wait states and will not cause too much conflicts with other external bus transactions.

If a simple latch without read-back capability is used as an output port, a 'shadow' register must be maintained in the DSP memory to allow manipulation of individual bits.

CPLD and FPGA implementations of I/O ports make possible the integration of advanced features, e.g. bit-change interrupts, to relieve the DSP from control tasks. I/O port devices for micro controllers (PIO) are getting rare and suffer from slow interfaces. These devices should be used with care: the bus adaptation logic required to connect them to a fast DSP may be as costly as a direct implementation of the I/O functionality in a CPLD/FPGA.

#### 4.4 Sensors

Many signal processing applications must take into account the influence of environmental conditions like temperature to perform compensation or supervise limits. Time-keeping is also often required. If these information cannot be supplied by a host system, suitable devices must be connected to the DSP. Often they are accessed in large time intervals only, because of slow variation of the measured quantity. Serial interfaces (SPI, Two-Wire, I<sup>2</sup>C) are preferable and commonly used. The serial interface can be either emulated via I/O ports in software or implemented in hardware, depending on how much time is left for software emulation. Other sensors might require a parallel bus interface. Again, additional glue logic and careful examination of bus conflicts is required.

#### 4.5 Networks

A variety of networks are used for communication in industrial and automotive systems. Mostly field busses like CAN have been used, but more and more Ethernet is making it's way. Most network controllers handle the physical layer and the lower protocol layers. They are typically designed for mass market applications: ISA, PCI and PCMCIA/PC-Card busses. A few are designed to directly interface to commonly used micro controllers.

None of these devices will fit to the external bus of a DSP without glue logic and timing adaptation. Slow bus cycles are common, which might cause bus locks.

#### 4.6 Other I/O Interfaces

FireWire, USB, SCSI to name but a few, may also be used to communicate with other systems. Typically you will not find a DSP with these interfaces as a built-in peripheral, external controllers must be used. Bus adaptation and careful examination of bus performance and conflicts is required.

## 4.7 Control Software

DSP architecture is optimised for signal processing which requires parallel multiply and accumulate, address update and data transfer. When it comes to executing typical control algorithms, parallel processing is no longer possible. The DSP will not operate faster than a 'normal' processor with identical cycle time. Often micro controllers provide special instructions for control tasks, like testing the state of a port bit and branching accordingly. The DSP will have to execute several instructions for this operation, so performance might be even inferior to a fast micro controller. Even if the DSP features bit test instructions, this may not be supported by the C compiler and instead a series of logical operations is performed. This is often overlooked if calculation power requirements are estimated.

It is typical for control code that program flow is not linear, execution depends on the results of previous instructions, conditions have to be evaluated and branches are taken.

A typical control code may be:

```
if (*(int*) UART_STATUS_REGISTER & RX_READY)
{
    /* read command from UART */
    command = *(int*) UART_RECEIVE_REGISTER & 0xFF;

    /* check errors */
    if ( *(int*) UART_STATUS_REGISTER & ERRORS )
    {
        error_handler ();
    }
    else
    {
        /* dispatch command */
        switch (command)
        {
            case CMD1: ....; break;
            case CMD2: ....; break;
            case CMD3: ....; break;
            ....
        }
    }
}
```

Such code leaves little chance for parallel computation or to fill delay slots. The 'if' and 'case: ... break;' instructions will cause multiple branches which disrupt the execution pipeline and cause additional delays. Conditional execution of instructions, offered by some DSPs, may help, but only if the actions taken according to the condition are small enough to be coded in a few instructions. DSPs with an instruction cache might suffer from control code displacing the signal processing code in the cache, thus additional cycles will be wasted on re-entrance to signal processing.

The speed of control code execution cannot be scaled with execution cycle time, fast DSPs will often use a longer instruction pipeline or latencies exist on register loads.

From our experience, control tasks often present more problems than the signal processing itself because of the following reasons:

Highly optimising C compilers tend to perform code re-arrangements to yield a better efficiency. This may cause problems if the peripheral requires a strict access sequence. Several slow communication devices have restrictions on successive write operations and will require a delay between write cycles. As long as you compile without optimisation, the delay constraints are satisfied, but from a certain level of optimisation, successive writes will violate the controller timing. The resulting system misbehaviour will often occur only sporadic, and it's cause is hard to find: as soon as you single step through the program for debugging, everything works fine again.

Polling loops are sometimes reduced to a single read instruction, if the variable to poll has not been declared volatile:

```
while (! (*(volatile int*)PORT_ADDRESS & BITMASK)) ;
```

*Verify your control algorithms with all possible optimiser settings to prevent trouble if the code is recompiled later with different options. Note that the assembler code generated by the compiler will often change dramatically if the debug (-g) option is removed!*

## **5 DSP and Memory**

### **5.1 Memory Organisation**

A key feature of all DSPs is the use of a Harvard architecture, i.e. independent memories and busses are used to access data and program code. This allows simultaneous op-code fetches and data movements in a single instruction cycle. Because most signal processing algorithms require two data operands for each instruction, this architecture has been extended or modified on most designs.

Signal processing is typically based on multiply - accumulate instructions, with a sample being multiplied by a coefficient and the result added to the accumulator. The DSP must be able to read two operands, sample and coefficient, and the op-code from memory in a single cycle. Therefore two data memories and busses and a program memory and bus must exist. Examples for this topology are the Motorola 56K and Texas Instruments C6000 DSPs.

Other DSPs use combined data/program memory, divided into at least two simultaneously accessible banks, and add an instruction cache. If the op-code can be fetched from the cache, two data transfers are possible in a single cycle. Cache strategies differ widely. While some DSPs simply load every instruction into the cache, more sophisticated implementations caches only those instructions which conflict with a dual data move. The simple implementation is prone to cache misses after the signal processing flow has been interrupted by data acquisition interrupts or control code. You may consider to execute your time-critical signal processing tasks once, to make sure that the instructions are stored in the cache, and then freeze the cache contents. Depending on cache size and strategy, this may work for short program segments only.

Some C6000 family members use a data cache to boost performance on systems using external SDRAM memories for data storage. This technique will lower systems costs for applications requiring large data buffers. Block-wise processing is required to yield good performance results.

Analog Devices Sharc<sup>®</sup> DSPs use a dual-ported memory with additional FIFOs as internal memory. This architecture allows simultaneous data moves by the DSP core plus additional DMA transfers in a single cycle. DSPs using a 'standard'

memory will either delay the DMA or the core data transfer on simultaneous access.

To reduce pin count, most DSPs feature only a single external bus interface. External bus access times are often longer than instruction cycle times to allow the use of standard memories with reasonable timing requirements. Modern DSPs directly support the connection of various external memory types like asynchronous SRAM, SDRAM, and synchronous SRAM. The bus-timing is programmable for each of these memory types. Locating program or data in external memory might cause several delays because of access conflicts and idle states. SDRAM interfaces are included on many new DSP designs to lower system costs for applications requiring large memories. As with all DRAMs, crossing page boundaries will add significant delays caused by additional pre-charge cycles. This should be carefully considered if data buffers are linked to external SDRAM.

As noted in the data acquisition section, the external bus address range is often divided into multiple banks with individual chip selects and wait state configuration. It is quite common that additional idle cycles are inserted if a bank boundary is crossed, especially on switching from synchronous to asynchronous memories.

Large internal memory is preferable for demanding applications, however it is extremely costly. Because even internal memory has several restrictions for simultaneous access, a key to system performance is linking data and program memory sections to suitable memory areas. These issues will be discussed in section 6.

## 5.2 Operational Units

Hardware multiplier and accumulator are the key parts of the DSP core. Most designs also feature a barrel shifter which is useful for scaling and bit-manipulations. An ALU is a matter of course. Mostly specialised address generation units are added, which support indexed addressing modes, auto-increment and decrement, often with additional hardware support for bit-reversal (FFT) and circular addressing modes (acquisition buffers). Very few DSPs can use memory operands directly. Most DSPs operate from a register file and require to copy the content of a memory cell to a register before processing.

If you prefer to write your programs in Assembler language, you will have full control over the DSP's architectural features and discover lots of performance

differences, depending on the application: Some offer wide accumulators to support high precision if intermediate results can stay in the accumulator. Some offer limiting and rounding in hardware, while other DSPs will require additional instructions. Specialised instructions may exist to isolate and manipulate bit-fields. However, C compilers are not always capable to take full advantage of these features. Inline assembly instructions or macros are commonly required, at the expense of losing program portability.

As with 'standard' micro processors, Two different approaches exist:

CISC DSPs feature a complex instruction set with special hard-wired instructions to speed-up common signal processing algorithms. An example may be the Texas Instruments C54x series with a sophisticated instruction set for telecommunications, e.g. Viterbi encoding. The Analog Devices Sharc<sup>®</sup> DSPs feature several complex instructions for general purpose arithmetic.

RISC DSPs use a reduced instruction set which can be executed at higher speeds. Complex operations have to be programmed as a series of simple instructions. The higher execution speed at least compensates the increased instruction count.

Popular micro processors also rely on RISC instruction sets to push execution speed beyond several hundred MHz. CISC instructions are internally broken up into a series of RISC instructions. Lots of supplemental functional units exist to achieve high levels of parallelism and minimise pipeline delays, e.g. branch prediction units, pipeline save and restore on function call and return, etc. The Texas Instruments C6000 DSP uses a different approach and moves 'intelligence' away from the DSP core to the Code Generation Tools: The DSP core is uncompromisingly optimised for fast execution. The compiler / optimiser is in charge of distributing execution units for parallel execution, filling delay slots and minimising pipeline effects to achieve optimum performance. This technique yields outstanding results on 'long' algorithms with lots of repetitions, e.g. FFT and convolution. However, it often cannot maintain its performance on 'short' calculations.

### 5.3 Data Format

Each discussion about data formats must start with the compulsory fixed-point vs. floating point debate:

Floating point offers an enormous dynamic range and promises to avoid scaling problems. This is surely true for many applications. But if precision requirements are more demanding, you must observe what happens inside the DSP and the compiler. If additions are executed, floating-point operands are scaled to identical exponents and the mantissas are added. Then the result is normalised again. As you can see, there is no precision improvement compared to a fixed-point DSP with a data word size identical to the floating-point DSP mantissa size. Large accumulators will help to maintain precision, but often the C compiler will store intermediate results in variables. To maintain the high precision, 64 bit floating point formats may have to be used which dramatically increase computation time and storage requirements. If this should be avoided, scaling may be required on a floating-point DSP too.

TI's C6000 series have some peculiarities which deserve special attention. Although the C6201 data bus, internal memory, register file, adders and shifters are 32 bit wide, the multipliers are 16x16 bits only. The 1600 MIPS throughput is based on the assumption that 16 bit data is used. In this case, two 32 bit wide operands are loaded in a single 5 nsecs. instruction cycle. This data is interpreted as four 16 bit operands, word1 D0..D15 being operand1, word1 D16..D31 operand 2 etc. Several instructions exist to multiply the upper and lower halves of the input registers. This is comparable to the MMX or 3DNow! instructions of advanced x86 processors. If 'real' 32 bit data is used, processing speed is limited to 800 MIPS by the two simultaneous data fetches each cycle. 32 bit multiplication must be performed as a sequence of 16 bit operations. Multiplying upper and lower 16 bit words is not supported by ANSI C instructions, inline assembly macros must be used to take advantage of this feature.

Internal data representation might be of concern if data must be interchanged with other systems. Today, IEEE floating point formats are used, some older DSP designs use a different format and require conversion.

## 5.4 Arithmetic

As described, DSPs can perform multiplication, addition and subtraction in a single cycle. All other calculations have to be programmed based on these fundamental operations. The `maths.lib` is part of the ANSI C extensions and is provided with every C compiler. However, performance differences are noticeable. If a division or square root should be calculated, iterations are used. Some DSPs like the Sharc<sup>®</sup> feature instructions to generate a relatively precise seed as a start value for the iteration in a single cycle, which greatly reduces the number of iteration loops.

## 5.5 Built-in Peripherals

Common to all DSPs is a synchronous serial port to interface ADCs, DACs, Codecs and several time-division-multiplex interfaces. Multi-channel mode to support TDM in hardware and reduce data transfer overhead is not available on all DSPs. The synchronous serial port can also be used for inter-processor communication and even for networking if TDM is supported. In TDM mode, a control pin which reflects the state of the transmitter (high-impedance or active) is useful if external bus drivers are required, e.g. for differential transmission lines .

Timers are available on most DSPs too. They can be used to schedule program events, provide sampling clocks, measure or count external events or even generate PWM output.

External DMA requests are mostly used for data acquisition. External to internal memory transfers, like block-wise transfer of large acquisitions buffers into internal memory or code overlays, may not be possible without additional hardware generating DMA requests. Other DSPs can perform these tasks by internally generated DMA triggers.

External interrupts are used for data acquisition and exception handling. Level triggered interrupts require some precautions in software to avoid triggering multiple interrupts from a single event, but allow sharing an interrupt line by multiple open-drain devices. Edge triggered interrupts are easier to handle, but interrupt sharing is not supported. The interrupt response time of a DSP is usually very short, due to it's high execution speed. The time from occurrence of the interrupt

condition to the desired reaction however is determined mostly by the compiler and the DSP interrupt prioritising and nesting capabilities.

I/O ports are often used to read the status of external devices, e.g. an A/D converter's conversion ready output in busy-polling mode. Another common use is synchronising multiple processors. Typically only a few ports are available, often external I/O must be connected.

Some DSPs have specialised peripherals, tailored to mass market applications, e.g. motor control. If these devices match your requirements, they will offer unbeatable price / performance ratios and straightforward system design.

## 5.6 Benchmarks

Benchmarks are intended to reveal the DSP's performance and capabilities. Their 'practical' use should be considered carefully. A typical example is the 1024 points complex FFT:

First, you cannot directly compare a fixed-point and a floating-point implementation. Fixed point FFTs require scaling the input samples or intermediate results to avoid overflow. Imagine a FFT of a full-scale DC signal on a 16 bit machine: all input samples will be 32767. The FFT result should be a single frequency bin at DC with a real-value of  $512 \times 32767$ . Because this number cannot be represented in 16 bit format, the input samples will have to be scaled down 9 bits to avoid overflow, reducing dynamics to 7 Bit, respectively 42 dB! Sophisticated fixed-point algorithms include auto-scaling during FFT computation, but additional cycles are required for this. As a benchmark code, you will find only the simplest and fastest version without any scaling!

FFT results often have to be re-ordered by bit-reversal addressing. If the DSP lacks bit-reversal address generation, lots of additional instruction cycles will be used before the results are prepared for further processing.

Most benchmarks assume data readily available in internal DSP memory. Typically background tasks have to acquire and output data in parallel with processing. If the algorithm occupies all data paths, like a convolution, every background (DMA) transfer will collide with a core data access and additional cycles are required unless dual-ported memories or other buffering methods are used. If data buffers are too large to fit in internal memory, execution speed will be reduced noticeably too.

Beside these, several other side effects exist which may degrade performance under real-world conditions: Interrupt processing may disrupt the instruction cache and delay execution on re-entry. Some optimisations rely on buffer sizes being a power of two. Wrap-around of address pointers in these cases is straightforward by masking the upper address bits by using AND operations. Additional cycles for loop control and compare operations may be required if different buffer sizes are used. Most DSPs include suitable address modification modes, but often these require the use of special buffer start addresses.

## **6 Compiler and Code Development**

More and more user's write their DSP programs in C. This offers several advantages. Algorithms developed and simulated on a workstation can be implemented with minimum re-coding, and program maintenance is simplified. Modern DSPs are designed to support an efficient code generation in C. A large set of universal registers and stack manipulation instructions are essential. RISC implementations and 'orthogonal' instructions sets simplify compiler design.

No DSP compiler currently offers the comfort of the code generation tools used for personal computers. Exception handling, garbage collection and all these nice features are not implemented in favour of reduced memory requirements and better code efficiency. Writing DSP programs in C still requires a good understanding of the DSP's hardware architecture - or plenty of computation power overhead.

Several myths exist about compiler efficiency. Of course there are differences, but lots of misinterpretation is caused by a direct comparison to hand-optimised assembly code: A DSP with a reduced instruction set will require a certain amount of operations to solve a given problem, even if coded in Assembler. The C compiler will use only a few instructions more, so it is honoured for superb efficiency. On a different DSP, the C compiler might produce comparable results, but the specialised instruction set of this DSP may allow to solve the problem with much less Assembler instructions - this Compiler is blamed for bad efficiency.

Making use of a DSP's special features will require the use of assembly language statements (often encapsulated in macros), no matter which DSP you use. You must take the decision weather to write 'pure' and portable C code - and waste some performance, or to optimise performance - resulting in code only maintainable by DSP experts.

## 6.1 Assembly Language Modules

If possible you should use readily available software modules for the time critical parts of your program. Some years ago, optimised assembly language modules have been available from specialised third-parties only, nowadays most DSP manufacturers offer them either as an extension to the ANSI-C compiler or for free download from their Internet sites.

As long as your application is built around standard signal processing functions like FFT, convolution, LMS algorithms, etc., you will often find a suitable implementation in optimised Assembler language. Even highly specialised algorithms like telecommunications codecs are available for free. Most of this code is used for benchmark tests too, so you can be assured to get the optimum performance.

## 6.2 The Influence of the Linker on Execution Performance

In the DSP section we have described the various concepts of memory organisation and layout which are used to guarantee a dual-data access plus op-code fetch in a single cycle. During compilation, memory is allocated by the linker, so it becomes one of the key components to achieve the desired performance. No linker optimisers exist, at least at this stage of code generation you must have a basic understanding of the DSP memory layout and the usage of the various memory sections generated by the compiler. As always in engineering, contradictory requirements exist and a compromise has to be found.

Most signal processing algorithms will expect data and coefficient sets in simultaneously accessible memory banks. This is easily achieved if the DSP's internal memory is large enough. You may then locate the program code in external memory (if no special program memory exists). Only a few memory conflicts will typically occur, e.g. with data acquisition. Most of the time, signal processing code will be available from the instruction cache so the external bus will be free.

If data buffers are large and do not fit in internal memory, the situation gets more complicated. You should consider to move data into internal RAM by DMA for piece-wise processing. Another possible setup is to locate code and coefficients in internal memory and leave the external memory for data. It may be necessary to divide the code into different sections: internal for signal processing and external for framework tasks.

If interrupts occur frequently, or your program is divided into small functions, make sure that the stack and the program memory are located in different, simultaneously accessible memory areas. Typically, interrupt code is not available in the instruction cache and must be fetched from memory. If stack and code accesses cause conflicts, context save and restore as well as function entry code, may require twice as much cycles. The stack allocation is extremely important for functions using many local variables too. The C compiler will allocate memory on the system stack, the frame segment, for local variables.

Global variables are mostly stored in the const and bss sections (names may differ according to the compiler). Typical C programs frequently access these segments which therefore should not conflict with op-code fetches.

Other parts of C programs, like initialisation data, is accessed only during run-time initialisation and does not need special attention.

As you can see, no general rule-of-thumb can be given, memory allocation is dictated by your algorithms and the DSP's architecture. Profiling is the tool to verify the success and identify program parts which do not perform as expected. Most debuggers offer profiling options, if not, you may include profiling code, e.g. starting and stopping internal timers. If the timers are already in use, toggling I/O ports will allow to measure the time spent in a code fragment with external counters or storage oscilloscopes.

### **6.3 Real-Time Operating Systems**

Several Real-Time Operating Systems (RTOS) are available for popular DSP families. A RTOS is useful if many individual tasks have to be managed, or if access to shared resources has to be controlled. RTOS are especially useful on multi-processor systems, where they handle task distribution, communication and synchronisation. A typical RTOS job - handling data I/O - gets more and more obsolete as DMA controllers can be used for this.

## 6.4 Debugging

Most DSPs feature in-circuit debuggers, usually JTAG based. This allows to debug your program directly on the target hardware. Often this is called 'real-time debugging' , but caution: JTAG is a serial interface with a 10.25 MHz clock and lots of protocol overhead. This limits typical data transfer rate to a few 10 kBytes/sec. The debugger / emulator will halt the DSP for several milli-seconds to display the register file and some memory contents. This may have catastrophic effects on control loops, so use this 'real-time' definition with care!

The emulator may have other impacts on system behaviour: often serial ports or DMA are no longer serviced during emulator halts.

The emulator however is the preferred tool to debug hardware-related problems, e.g. interrupt processing and verifying data acquisition. It's speed makes it superior to software simulation if algorithm debugging is concerned, but be aware that some problems might be masked by real-word data. Errors in a filter implementation might cause limit cycle oscillations or noise which is easily blamed on the A/D converter. You should verify your algorithms with specially selected test data sets too.

Debuggers with data visualisation capabilities may save you a lot of time to detect problems like ringing, limit cycle oscillations, clipping etc. Most of these debuggers can display data in time and frequency domain, some even in FFT waterfalls, eye diagrams or polar plots.

## **7 Conclusion**

Although a lot of text has been written so far, DSP design is straightforward if a few basic estimations are performed in advance:

- external bus bandwidth and possible collisions caused by memory access, peripherals and data acquisition
- runtime of interrupt services, control and framework tasks
- does the DSP architecture and instruction set support your application's requirements ?
- are you willing to spend some time in getting acquainted to the DSP's architecture and features to achieve optimum performance, or do you prefer to use a hardware-independent programming technique requiring a faster system and more memory resources?

We hope this paper has provided some help in avoiding common pitfalls. We greatly appreciate any feedback, comments and suggestions and we will be pleased to include them into this document.

**D. SignT**

Digital Signalprocessing Technology

Norbert Nölker & Adolf Klemenz GbR

Gelderner Straße 36

D - 47647 Kerken

phone +49 (0) 2833 / 570 977

fax +49 (0) 2833 / 33 28

email [info@dsigt.de](mailto:info@dsigt.de)

www <http://www.dsigt.de>