

Mechatronics Laboratory Assignment 4

Parallel Communication

Glue Logic, Hardware Interrupts, Analog to Digital Conversions, and Board Fab

Recommended Due Date: By your lab time the week of March 2nd

Possible Points: If checked off before your lab time the week of Mar. 9th ... 14 points

If checked off after your lab time the week of Mar. 9th and before your lab time the week of Mar. 16th ... 11 points

If checked off after your lab time the week of Mar. 16th and before 4:00PM Mar. 20th ... 9 points

If work not finished before 4:00PM Mar. 20th ... 0 points

Goals for this Lab Assignment:

1. Gain a better understanding of glue-logic and how the DSP interfaces (or communicates) with I/O hardware.
2. Introduce the hardware interrupt (HWI) DSP/BIOS object.
3. Learn about how to use an analog-to-digital converter
4. Learn how to design a new board for the DSP

DSP/BIOS Objects Used:

HWI, PRD

Library Functions Used:

Init_DAC7724, writeDAC7724, ADC7864_Start, ADC7864_Read

Matlab Functions Used:

FIR1

Prelab: Review and know how to use the FIR1 function in Matlab to design both low- and high-pass filters. Understand how to take the coefficients given by FIR1 to create difference equations that are coded in C to implement a filter. The help for “filter” in Matlab gives the order of coefficients for filters created in MATLAB. You will have to design a FIR filter for the lab, so know how to code one in C.

The following sections are presented to help you with the written portion of your prelab. Please complete a rough draft of this prelab by the date assigned in lab; the final version will be due with the remaining parts of the report. Your T.A. will post a due date.

The goal of this prelab is to introduce you to writing software drivers for hardware I/O boards. We will first review basic electronics terminology to help with reading the manuals. An overview is then presented on how DSP’s generally communicate with attached hardware. Next, we describe how a general write cycle occurs to an external hardware device, and for an example we provide a step-by-step explanation of the software driver for the DAC portion of the I/O daughter card. You will then be asked to do the same for the ADC portion. There are a number of data sheets at the GE420 lab web site that are specifically referenced in this prelab, so please read them when referred to do so to get a better understanding of what we are discussing.

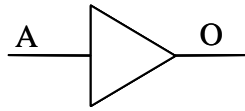
Prelab: Electronics Preliminaries

The following section on notation will be helpful in understanding many of the figures in this lab. First, you will have to learn how to convert between Hexadecimal, Decimal, and Binary. There is no method of representing binary in C, the default method is decimal, and a hex representation is always begun with a ‘0x’ prefix (i.e. 0xF). The following table should be helpful in basic conversions, but the Windows Accessories calculator is probably the most useful for big

numbers. Open up the standard Windows calculator, change the mode to Scientific, enter the decimal number in binary and use the HEX, DEC, and BIN button to convert between binary/decimal/hex for you.

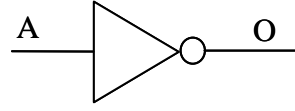
Binary Decimal Hex

0000	0	0x0
0001	1	0x1
0010	2	0x2
0011	3	0x3
0100	4	0x4
0101	5	0x5
0110	6	0x6
0111	7	0x7
1000	8	0x8
1001	9	0x9
1010	10	0xA
1011	11	0xB
1100	12	0xC
1101	13	0xD
1110	14	0xE
1111	15	0xF



Buffer

A	O
1	1
0	0

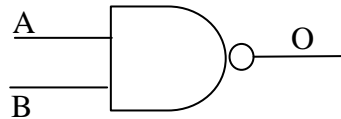


Inverter (74F04 chip)

A	O
1	0
0	1

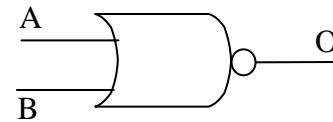
1 = 5 / 3.3V

0 = GND



NAND (74F00 chip)

A	B	O
0	0	1
0	1	1
1	0	1
1	1	0

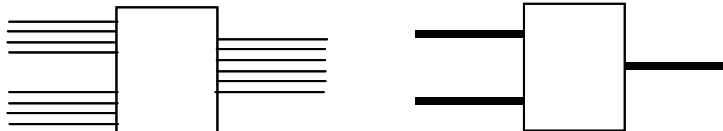


NOR (74F02 chip)

A	B	O
0	0	1
0	1	0
1	0	0
1	1	0

Most circuit diagrams use logic elements to implement glue logic. Since these may not yet be familiar to you, the most common ones are listed above, alongside their truth tables and chipset used to implement each element.

- The buffer circuit (aka. follower) is denoted by an op-amp symbol because this is how it was originally implemented in telephone systems. Buffers are used to ensure sufficient signal strength (impedance) for later circuits, and as a crude method to isolate static charges, etc., that may otherwise pass into the DSP.
- The inverter changes logic 1 (5V or 3.3V) signals to logic 0 (GND), and vice versa.
- The NOR and NAND circuits are used to compare two signals and output a third signal based on the comparison.
- The small circle in the Inverter, NOR, and NAND represents an inversion operation. Thus, a NAND block that is missing the inversion is simply an AND circuit (the choice of logic elements is generally the choice of the circuit designer... we use NAND and NOR throughout the daughter card).
- A thick line going into an element is used to represent multiple elements. For instance, the following diagrams are equivalent:



- A bar above the name of a signal line indicates that the line is active when the signal is low. For instance, the chip-enable signal \overline{CE} means that the chip is enabled only when the line is 'pulled low', or digital 0.

General Communication to Peripherals

First, you should browse the “External Memory Interface” section of the “TMS320C6000 Peripherals Reference Guide”. If you installed the full version of CCS on your home computer, this manual is in the Help Files under “User Guides” on the main page (the document number is SPRU 190). If you did not install CCS, this specific guide can be found at: coeecs1.ece.uiuc.edu/ge423/datasheets/spru190d.pdf. Within the “External Memory Interface” section, browse the “Overview” and “Asynchronous Interface” subsections. While this may be overwhelming for a first-time reader, the figures and diagrams in this prelab are based on these subsections, and it will be helpful for you to have seen them in their original form and perhaps print them, particularly Figures 10-59 and 10-60.

Before we discuss the DAC, we first must discuss how the TMS320C6713 DSP communicates with peripherals (or chips) soldered to a daughter card external the DSP. This section of the prelab explains how the External Memory InterFace (EMIF) section of the DSP controls external pins on the 6713 DSP. If you were designing a DSP daughter board, you would connect these DSP pins to an I/O (Input/Output) chip to allow the DSP to send or receive data from external chips. Many times there is some additional hardware logic that needs to be placed between the DSP and the I/O chip to facilitate correct communication. We call this logic the “glue logic” for the chip interface.

The 6713 DSP has quite a number of pins that are controlled by the EMIF. Many of these pins are designed specifically for communicating with specific memory types: Synchronous Dynamic RAM (SDRAM), synchronous burst RAM, FLASH, etc. For the I/O chips we will be dealing with, we will be using an asynchronous memory interface that requires only the following pins: $\overline{CE2}$, \overline{AWE} , \overline{ARE} , EA2-21, and ED0-31. For an asynchronous device such as our daughter board, the communication is not driven by a generated clock signal as with a synchronous device. Instead a DSP communicates by matching specific read-and write-cycle timing specifications that are discussed in detail below.

General Write Cycle

A write cycle is the action of sending one word or 32 bits of data from the DSP to the daughter board or another external device. It is initiated in code by a command that writes to a memory location, such as:

```
*(int *) (0xA01F1DC4) = 1562334;
```

The data word that will be sent is on the right, the address it will be sent to is listed as the hex number starting with 0x. The entire hex number is in parenthesis and is typecast as a pointer to an integer by the (int *) specification. The leftmost star tells C to reference the integer value stored at the memory location 0xA01F1DC4. For instance, if `variable` is declared as a pointer to a 32 bit integer, then `*variable` refers to the 32 bit integer value pointed to by `variable`.

This single line of code, because it writes data to an external address location, automatically initiates a sequence of pin transitions that eventually indicate to the external device that the DSP is sending 32 bits of data on pins ED0-31. Look at Figure 10-60 in the “TMS320C6000 Peripherals Reference Guide” for a picture of the pin transitions. A general diagram of the write cycle is shown below in Figure 1, which is a simplification of Figure 10-60 in the user’s guide. If you compare our figure to the one in the reference guide, you will notice that we are ignoring some data signals that are not important to this discussion:

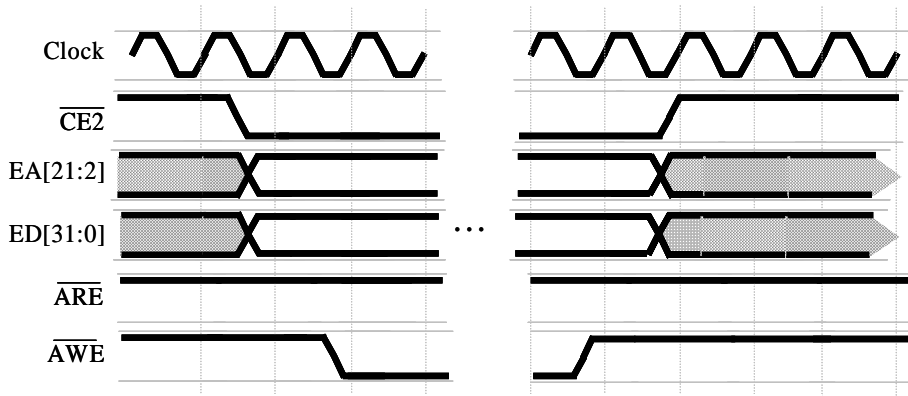


Figure 1: Timing Diagram for a Write Cycle

When an instruction in the DSP source code writes a value to an external address (See Table 3, “TMS320C6713 Datasheet” coecls.ece.uiuc.edu/ge423/datasheets/tms320c6713.pdf for the memory map of the DSP) the following pin transitions occur.

1. To setup for the write cycle:
 - a. Pin $\overline{\text{CE2}}$ is pulled low.
 - b. Pins EA2-EA21 are simultaneously set to match the address written. (Note: the C6713 DSP has 32 bits of address space but externally you can only use 19 bits of asynchronous addressing.). The board designer will use the $\overline{\text{CE2}}$ pin along with the EA2-EA21 pins and glue logic to generate the chip select ($\overline{\text{CS}}$) signal for their device.
 - c. Pins ED0-31 are simultaneously set to match the bits of the value written.
2. The write cycle occurs when the pin $\overline{\text{AWE}}$ (Write Enable) pulses low. The duration of this event is called the strobe length and is specified in the DSP EMIF settings to match the external device. A device’s data sheet lists the minimum strobe length for its write-enable line. For the chips on the daughter card in lab, the strobe length is 100 nanoseconds or less.
3. If there are no more writes scheduled, the $\overline{\text{CE2}}$ line is pulled back high.

Example

So take for example this line of C code:

```

*(int *) (0xA01F1DC4) = 1562334;
      address           data
  
```

What happens to the external pins?

1. a. $\overline{\text{CE2}}$ is pulled low, because the address 0xA01F1DC4 is in $\overline{\text{CE2}}$ space (address range 0xA0000000-0xAFFFFFFF).
- b. Pins EA2-EA21 are set to the **address** values in binary, i.e.:

Address bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	bits	1	0	1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	1	1	1	0	0	0	1	0	0	
	word	A				0				1				F				1				D				C				4			

Note that the processor uses (reserves) bits 0 and 1 and bits 22 and higher. We don't have access to them from the daughter board (there are no physical pins), so we differentiate these bits in gray.

c. Pins ED0-31 are set to the **data** values in binary, i.e. 0x17D6DE in HEX:

Data bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bits	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0	1	1	1	0
word	0				0				1				7				D				6				D				E			

- The $\overline{\text{AWE}}$ pin is pulsed low for the strobe length. During this time the device accepts the data on the ED0-31 pins.
- $\overline{\text{CE2}}$ is pulled back high.

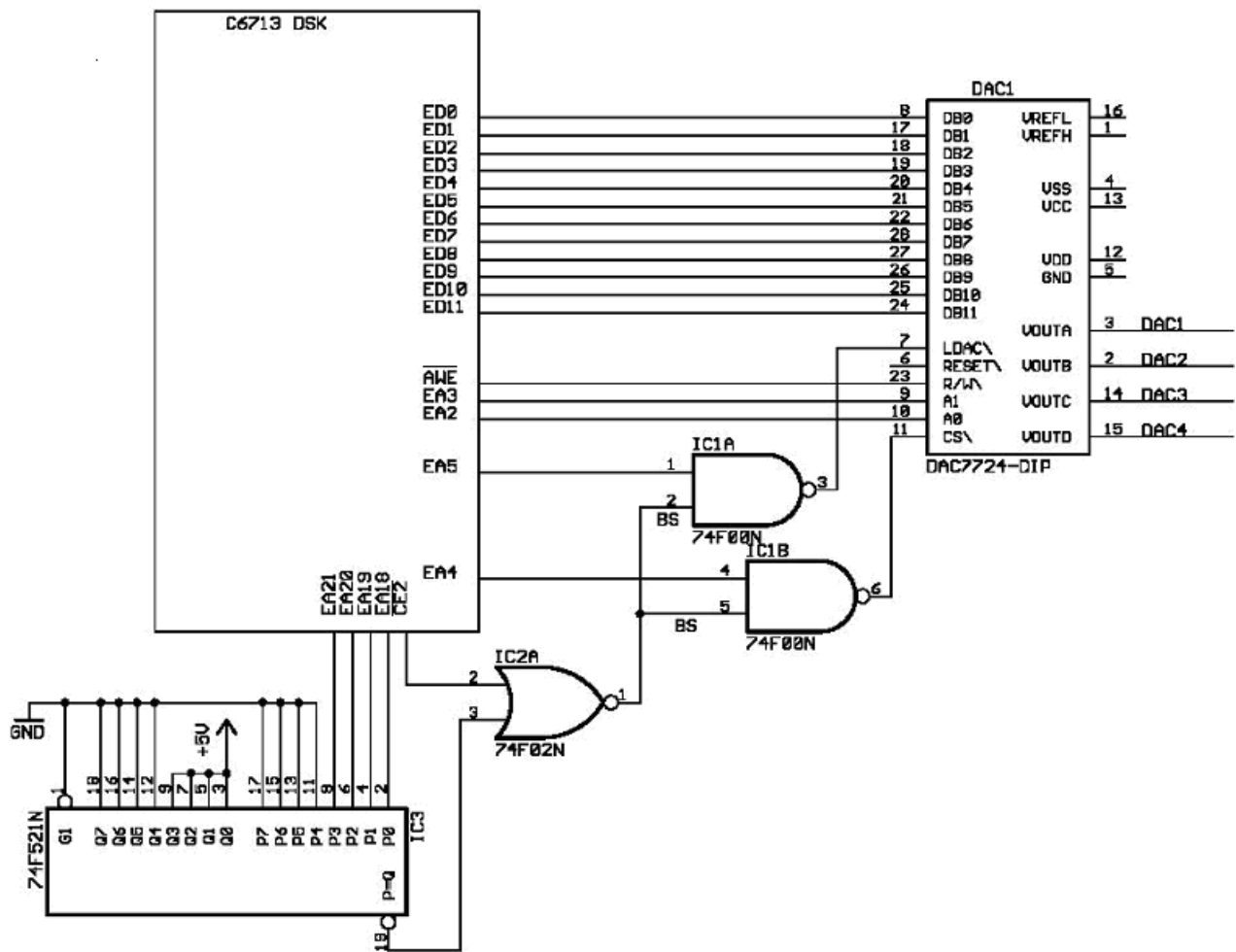


Figure 2: DAC portion of the Daughter Card

Specific I/O Example: DAC7724 on the Daughter Card

Now let's take a look at the source code for the function **writeDAC7724** in the file **dac7724.c**. Along with the source code, you will want to study the schematic given in Figure 2 above and also look at the DAC7724's datasheet (coecsl.ece.uiuc.edu/ge423/datasheets/dac7724.pdf). The abbreviation, DAC, stands for Digital-to-Analog Converter.

There are two functions that deal with the DAC: **Init_DAC7724** and **writeDAC7724**. The function **Init_DAC7724** initializes the DAC7724 by zeroing the 4 channels and resetting the RESET pin. The function we are going to focus on is **writeDAC7724**, which is the nuts and bolts function that communicates to the DAC7724 chip. Looking at Figure 2, you will see that only 12 of the 32 data lines (ED0-11) are connected between the DSK and the DAC chip. This is because the DAC7724 is only a 12-bit DAC. So to write a value to one of the DAC channels you perform a write cycle that sets pins A0 and A1 to the desired DAC channel, and the desired DAC value on pins D0-D11. After writing the new output value, the DSP needs to instruct the DAC7724 to latch this new value to the analog output stage. By sending a pulse to the DAC7724's $\overline{\text{LDAC}}$ pin, the DAC is instructed to latch. In the daughter board design, we choose to use a write instruction with EA5 high to produce this pulse.

We now explain the driver code for the DAC7724. If you are not familiar with the DAC7724 and the other chips mentioned you will need to reference their data sheets. You can find these data sheets at the GE423 web site

Source File: **dac7724.c**

```
void writeDAC7724(float dac1,float dac2,float dac3,float dac4) {
```

There are 4 DACs on the DAC7724, therefore the **writeDAC7724** function requires all four output voltage values to be passed to it.

```
int rawdac1,rawdac2,rawdac3,rawdac4;
```

These four integer variables will be used to store the actual "raw" value that will be written to the DAC. Remember that the DAC7724 is a 12 bit DAC. That means it can accept an integer number between 0 and 4095 ($2^{12} - 1$). For this DAC chips 0 is -10 Volts, 2048 is zero Volts and 4095 is 10 Volts.

```
rawdac1=(int)(dac1*204.8+2048.0+0.5); // 0.5 is used to round to the nearest integer  
rawdac2=(int)(dac2*204.8+2048.0+0.5); // 0.5 is used to round to the nearest integer  
rawdac3=(int)(dac3*204.8+2048.0+0.5); // 0.5 is used to round to the nearest integer  
rawdac4=(int)(dac4*204.8+2048.0+0.5); // 0.5 is used to round to the nearest integer
```

The next 8 lines of code guard against sending an invalid number to a DAC channel. It is possible that a user of the **writeDAC7724** function could pass a number greater than 10 for example. In this case these checks would catch that error and only send 4095 to the DAC.

```
if (rawdac1 < 0) rawdac1 = 0;  
if (rawdac1 > 4095) rawdac1 = 4095;  
if (rawdac2 < 0) rawdac2 = 0;  
if (rawdac2 > 4095) rawdac2 = 4095;  
if (rawdac3 < 0) rawdac3 = 0;  
if (rawdac3 > 4095) rawdac3 = 4095;  
if (rawdac4 < 0) rawdac4 = 0;  
if (rawdac4 > 4095) rawdac4 = 4095;
```

These final four lines of code are where the DSP actually communicates with the DAC chip. All pin transitions will be described.

```
*(volatile int *) (0xA03C0010) = rawdac1;
```

The write command to an address space starting with hex 'A' automatically causes the EMIF to initiate a write cycle to $\overline{\text{CE2}}$ memory space. We now begin the three steps of writing as mentioned earlier. (Step 1.a.) The EMIF will pull $\overline{\text{CE2}}$ low because we are writing to an address in the 0xA0000000-0xAFFFFFFF range. (Step 1.b.) The EMIF will then set EA2-21 to:

Address bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Value	Bits	1	0	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
	Word	A				0				3				C				0				0				1				0				

It is in this step that the DAC chip needs to know that it is selected (will be used). If you refer to the circuit diagram of Figure 2, we see that when EA18-EA21 are all high, the 74F521 comparator chip's output "O" is driven low because the jumper settings are set to compare to 1111. With $\overline{\text{CE2}}$ low the NOR gate drives its output high. This signal is named the "Board Select" (BS) because this signal is used for every chip on the daughter card to indicate when the DSP is writing to the daughter card. Now with BS high and EA4 high the NAND gate's output is driven low. This selects the DAC7724, which has an active low chip select ($\overline{\text{CS}}$) input. The lines EA2 (connected to A0) and EA3 (connected to A1) are both low selecting the DAC1 input register. (Step 1.c.) Data lines ED0-11 are set with the value in *rawdac1*. (Step 2) Finally $\overline{\text{AWE}}$ (connected to $\overline{\text{WR}}$) is pulsed low. This causes the data on ED0-11 to be transferred to the DAC1 register in the DAC7724. (Step 3) The $\overline{\text{CE2}}$ is automatically pulled high by EMIF. We are now ready to write to the second DAC channel

```
*(volatile int *) (0xA03C0014) = rawdac2;
```

This causes the EMIF to again initiate a new write cycle. (Step 1.a.) It will pull $\overline{\text{CE2}}$ low because we are writing to an address in the 0xA0000000-0xAFFFFFFF range. (Step 1.b.) It will set EA2-21 to:

Address bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Value	bits	1	0	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
	word	A				0				3				C				0				0				1				4				

Again, the pins EA18-EA21 are all high causing the 74F521 comparator chip's output "O" to be driven low. With $\overline{\text{CE2}}$ low the NOR gate drives its output high. With BS high and EA4 high the NAND gate's output is driven low. This selects the DAC7724. The lines EA2=1 and EA3=0 select the DAC2 input register. (Step 1.c.) Data lines ED0-11 are then set with the value in *rawdac2*. (Step 2) Finally $\overline{\text{AWE}}$ is pulsed low then high causing the data on ED0-11 to be transferred to the DAC2 register in the DAC7724. (Step 3). The $\overline{\text{CE2}}$ pin is pulled high by EMIF.

```
// DAC 3 is 1C due to a mistake in the board schematic
*(volatile int *) (0xA03C001C) = rawdac3;
```

For DAC 3 we have very similar pin transitions. (Step 1.a.) The EMIF will pull $\overline{CE2}$ low because we are writing to an address in the 0xA0000000-0xAFFFFFFF range. (Step 1.b.) It will set EA2-21 to:

Address bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	bits	1	0	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
	word	A				0				3				C				0				0				1				C			

EA18-EA21 are all high causing the 74F521 comparator chip’s output “O” to be driven low. With $\overline{CE2}$ low the NOR gate drives its output high. With BS high and EA4 high the NAND gate’s output is driven low. This selects the DAC7724. EA2=1 and EA3=1 selects the DAC3 LSB register. (Step 1.c.) Data lines ED0-11 are set with the value in *rawdac3*. (Step 2) Finally \overline{AWE} is pulsed low then high causing the data on ED0-11 to be transferred to the DAC3 input register in the DAC7724. (Step 3). The $\overline{CE2}$ line is pulled high.

```
// DAC 4 is 18 due to a mistake in the board schematic
*(volatile int *) (0xA03C0018) = rawdac4;
```

And again in the same fashion DAC4 is written to. (Step 1.a.) EMIF will pull $\overline{CE2}$ low because we are writing to an address in the 0xA0000000-0xAFFFFFFF range. (Step 1.b.) EMIF will set EA2-21 to:

Address bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	bits	1	0	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
	word	A				0				3				C				0				0				1				8			

EA18-EA21 are all high causing the 74F521 comparator chip’s output “O” to be driven low. With $\overline{CE2}$ low the NOR gate drives its output high. With BS high and EA4 high the NAND gate’s output is driven low. This selects the DAC7724. EA2=0 and EA3=1 selects the DAC4 input register. (Step 1.c.) Data lines ED0-11 are set with the value in *rawdac4*. (Step 2.) Finally \overline{AWE} is pulsed low then high causing the data on ED0-11 to be transferred to the DAC4 register in the DAC7724. (Step 3). The $\overline{CE2}$ pin is again pulled high.

The DAC channels now have the data we desire to output, but haven’t been told to output this new voltage value yet. This last line of code performs this “latch” instruction causing the voltage values to change. All the purpose of this instruction is to pulse the \overline{LDAC} pin low that then back high. The pin transition description will explain this.

```
// dummy write to Latch all four DACs simultaneously
*(volatile int *) (0xA03C0020) = 0;
```

This instruction causes the EMIF to pull $\overline{CE2}$ low and set EA2-21 to:

Address bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	bits	1	0	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	word	A				0				3				C				0				0				2				0			

EA18-EA21 are all high causing the 74F521 comparator chip’s output “O” to be driven low. With $\overline{CE2}$ low the NOR gate drives its output high. With BS high and EA5 high the NAND gate’s output is driven low, driving \overline{LDAC} low. For the

latch enable command the state of EA2 and EA3 *does not matter* so I set them both low. The $\overline{CE2}$ pin is again pulled high causing \overline{LDAC} to be pulled high.

General Read Cycle

The read cycle is now explained in detail. A read cycle is the process of sending data from an external device to the DSP. It is also initiated by the DSP via a sequence of pin transitions that indicate to the external device that the DSP is requesting a read of 32 bits of data on pins ED0-31. Look at Figure 10-59 in the “TMS320C6000 Peripherals Reference Guide” for a picture of the pin transitions. But for your convenience we provide a transition diagram below:

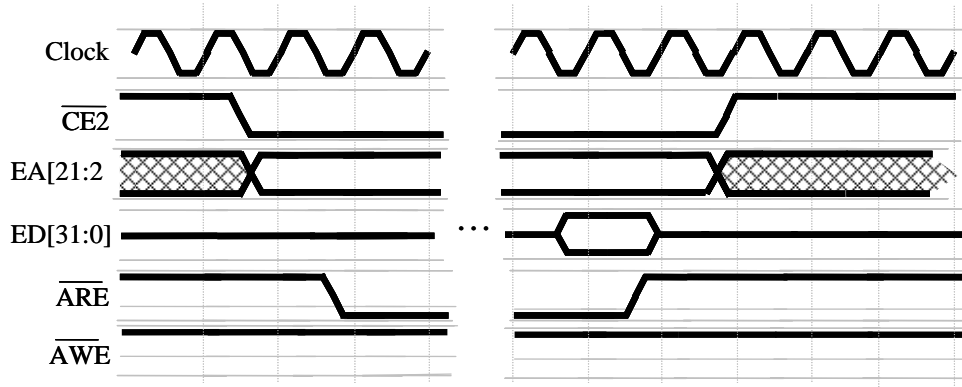


Figure 3: Timing Diagram for a Read Cycle

When an instruction in the DSP source code reads a value from an external address, the EMIF causes the following pin transitions to occur:

1. Setup:
 - a. Addresses Pin $\overline{CE\#}$ is pulled low (we always use $\overline{CE2}$).
 - b. Pins EA2-EA21 are set to match the address to read. Again, the board designer will use the $\overline{CE2}$ pin along with the EA2-EA21 pins and glue logic to generate the chip select (\overline{CS}) signal for the device.
2. Pin \overline{ARE} (Read Enable) pulses low for the strobe length of the read cycle. This strobe length is the main timing value that has to be matched with the device. The device’s data sheet lists the minimum strobe length for its read enable line. For the chips on the daughter card in lab the strobe length is 100 nanoseconds.
3. Towards the end of the strobe length the DSP reads the data lines ED0-31 to receive the data from the external device. This is represented by the open block of data that appears at the end of the strobe cycle of the \overline{ARE} line. The \overline{ARE} line is pulled high after the data is read.
4. The $\overline{CE2}$ line is pulled back high.

So take for example this line of C code:

```
mynewdata = *(int *) (0xA03A5678);
```

What happens to the external pins?

1. Setup:
 - a. $\overline{CE2}$ is pulled low, because the address 0xA03A5678 is in $\overline{CE2}$ space (address range 0xA0000000-0xAFFFFFFF).
 - b. Pins EA2-EA21 are set to:

Address bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	bits	1	0	1	0	0	0	0	0	0	0	1	1	1	0	1	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0
	word	A				0				3				A				5				6				7				8			

- \overline{ARE} pulses low for the strobe length
- During the strobe length the DSP reads the 32bit value on ED0-31. For this example let us assume that the state of the 32 pins were:

Data bits		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	bits	0	0	1	0	1	1	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	1	0	0	1	1	0	1	0	1	0	0
	word	2				F				9				5				4				4				D				4			

So now the variable **mynewdata** is assigned the value 0x2F9544D6 (Decimal 798311638). The \overline{ARE} is pulled back high after the strobe length.

- $\overline{CE2}$ is pulled back high.

NOTE: These types of read and write cycles are called “parallel” transfer cycles as compared to a “serial” transfer. The data is written/read 32 bits at a time instead of 1 bit at a time as would occur with a serial interface as in the last lab.

Format for Prelab:

1. When I ask for detailed comments on signal pin transitions I am asking for explanations with the same detail that I used to explain the DAC interface in the above reading.

2.

```
void ADC7864_Start() {  
    // Dummy write to pulse the convert start pin  
    *(volatile int *) (0xA03C0080) = 0;  
}
```

Here add detailed comments about what external signals are being driven to tell the ADC chip to start a conversion.

```
}
```

3. Now that the conversion has started explain how the DSP will know that the conversion has completed. What pin transition occurs?

4.

```
void ADC7864_Read(float *adc1, float *adc2, float *adc3, float *adc4) {  
    int rawadc1, rawadc2, rawadc3, rawadc4;
```

```
    rawadc1 = *(volatile int *) (0xA03C0100) & 0xFFFF;
```

Add detailed comments explaining how DSP pins read ADC1's value

```
    rawadc2 = *(volatile int *) (0xA03C0100) & 0xFFFF;
```

Add detailed comments explaining how DSP pins read ADC2's value

```
    rawadc3 = *(volatile int *) (0xA03C0100) & 0xFFFF;
```

Add detailed comments explaining how DSP pins read ADC3's value

```
    rawadc4 = *(volatile int *) (0xA03C0100) & 0xFFFF;
```

Add detailed comments explaining how DSP pins read ADC4's value

What am I doing in these next 8 lines of code?

```
    if (rawadc1 >= 2048)  
        rawadc1 = rawadc1 - 4096;  
    if (rawadc2 >= 2048)  
        rawadc2 = rawadc2 - 4096;  
    if (rawadc3 >= 2048)  
        rawadc3 = rawadc3 - 4096;  
    if (rawadc4 >= 2048)  
        rawadc4 = rawadc4 - 4096;
```

What am I doing in these last 4 lines of code?

```
    *adc1 = (rawadc1 * 10.0) / 2047.0;  
    *adc2 = (rawadc2 * 10.0) / 2047.0;  
    *adc3 = (rawadc3 * 10.0) / 2047.0;  
    *adc4 = (rawadc4 * 10.0) / 2047.0;
```

```
}
```

Laboratory Exercise

Exercise 1: Using the ADC

To demonstrate the hardware interrupt (HWI) section in DSP/BIOS, we will take advantage of the fact that the AD7864 generates a pulse signal on pin “BUSY” at the end of its conversion. The schematic in Figure 4 shows that this “BUSY” signal is brought into the DSP through the external interrupt pin EXT_INT7. The TMS320C6713 has four external interrupt pins EXT_INT4-EXT_INT7. When BUSY transitions high to low, indicating that the ADC conversion is complete, the DSP automatically stops the code it is currently processing and jumps to the interrupt service routine (ISR) specified for the EXT_INT7 pin. On completion of the ISR code, the DSP/BIOS kernel will resume processing the code interrupted.

Your first task will be to build an application that samples all four of the ADC channels and echoes their values to the DAC channels. Your application should:

1. Include a PRD function that is called every 1 ms. This function should start the ADC conversion each millisecond.
2. Set up HWI_INT7 to call a function that reads the ADC values and then writes them back out to the DAC channels. When setting up HWI_INT7 in the Configuration Tool, set the Function item to the function you wrote and make sure to select “Use Dispatcher” in the HWI’s properties. The remaining HWI property settings should remain at their default values. Also check that HWI_INT7 signal polarity is set correctly. To check signal polarity, right-click over the **HWI – Hardware Interrupt Service Routine Manager** and select the Properties option. You will then be able to select a low-to-high or high-to-low transition to initiate the interrupt for INT7; your prelab work should help you decide which one to pick.
3. Enable the interrupt bit for the ADC. DSP/BIOS does enable the global interrupt bit (this allows any interrupt to occur) during its initialization routine, but it does not enable individual interrupts. You must do that yourself in the main() function of your code. There are two registers that we will be concerned with, the Interrupt Enable Register (IER) and the Interrupt Clear Register (ICR). Read about the IER and ICR in Chapter 8 of “TMS320C6000 CPU and Instruction Set Reference Guide”, which can be found at the link below: (coecsl.ece.uiuc.edu/ge423/datasheets/spru189f.pdf). Carefully read the paragraph above Fig. 8-6 and the paragraph below Fig. 8-7 (in addition to both figures). Before we enable any hardware interrupts, we must make sure to clear any possible pending interrupts by writing to the interrupt clear register (ICR). Note, when making changes to the IER or ICR, we do not want to change the state of other enabled interrupts. The datasheet lists assembly code on how to do this using INT9, so we will now show you this same example in C. First, we write to the ICR to make sure no interrupts are pending on INT9. We do this by writing the bits to ICR that would ‘clear’ only this interrupt, i.e. hex 0x0200:

IER Register

Register Bits	IE15	IE14	IE13	IE12	IE11	IE10	IE9	IE8	IE7	IE6	IE5	IE4	Rsv	Rsv	NMIE	1
Value	bits	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	word	0			2			0			0					

ICR Register

Register Bits		IC15	IC14	IC13	IC12	IC11	IC10	IC9	IC8	IC7	IC6	IC5	IC4	Rsv	Rsv	Rsv	Rsv
Value	bits	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	word	0				2				0				0			

To enable INT9 without affecting the other interrupts, we use the “or” (|) operator to mask off the same bits, i.e. 0x200. The following lines of code would then appear in main to enable INT9.

```
ICR = 0x0200;           // clear pending interrupts
IER |= 0x0200;         // enable interrupt on line 9
```

This last line is an assignment operator shorthand notation (like the ‘+=’ operator), and could be equivalently written as:

```
IER = IER | 0x0200;     // enable interrupt on line 9
```

You must do the same process for INT7. (Note, if we wanted to halt all interrupts on INT9, we would use the ‘and’ operator and a bitwise mask to toggle bit 9 only off, i.e.: `IER &= 0xFDFE;` We don’t use this in this lab, but it is useful to know.).

Build, Load and Run you program.

With this small echo program you can show the effects of signal aliasing. Drive the ADC input with the function generator at your bench. Input a sine wave to the DSP and watch the DAC output on the oscilloscope. Vary the frequency of the input sine wave and notice at what frequency the output begins aliasing.

Exercise 2: Filter Design and implementation

Your lab instructor and TA will detail what is required for this exercise in during your lab time. In short, you will be asked to demonstrate 2 filters of your choosing.

Exercise 3: Board Layout

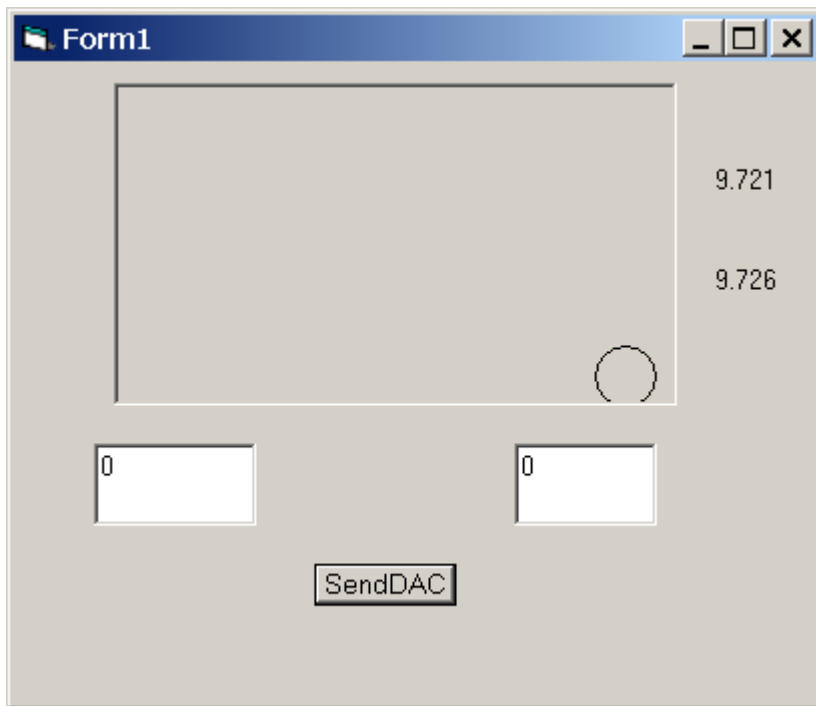
In lab, your TA will demonstrate how to lay out a simple circuit. You will follow his or her demonstration to build a circuit from scratch on your own computer. The lab will then conclude with a discussion on how to send the board off for fabrication and some design tips on making your own boards in the future. For reasons that will be explained in lab an older version of Eagle CAD is installed on the Lab computers. So if you would like to install Eagle CAD on your home computer use the install file found at http://coecsl.ece.uiuc.edu/ge423/datasheets/eagle/install_exe/eagle-4.11e.exe.

Exercise 4: VB Application Starter Help

The goal of this assignment is to get you up to speed faster in using VB for a host interface to your robot. This quick example shows you how to both download data from the PC to the DSP and upload data from the DSP to the PC. Also it shows you how to move an object inside a VB picture box.

When you are finished with both the VB code and the DSP code you will be able to type values into the text boxes and click SendDAC and the values you entered will be output on the DSP’s DACs 1 and 2. Also every 250ms the DSP will send your VB application the values of ADC1 and ADC2. The values will be displayed in two labels and a circle

inside the picture box will move to the x, y coordinates of (adc1,adc2). If the input into ADC1 and ADC2 varies, you will see the circle move on the screen.



1. Open up VB, create a new project, and immediately save it to a local directory. Our suggestion is `c:\yourloginname\lab4\VB_Robot_interface\`.
2. Open up the form view.
3. Add 2 text boxes and name them txtDAC1 and txtDAC2.
4. Add 2 labels and name them lblADC1 and lblADC2.
5. Add a MSComm (Microsoft Comm Control 6.0) object
6. To add this object to your project go to the menu Project->Components.
7. Under the Controls Tab, scroll down and check the Microsoft Comm Control 6.0 item and then Apply.
8. Now the MSComm object is added to your toolbar and you can add it to your project by clicking on it and drawing a square in an open space on the form.
9. Add a button, name it cmdSendDAC and give it the caption SendDAC. Also change the button's property "Default" to True. When the "Default" setting of a button is set to true, the button's code is run when you press the "Enter" key of the keyboard.
10. Add a Picture Box and name it pbxArena. Change the property AutoRedraw to True. Change ScaleHeight to 30, ScaleWidth to 30, ScaleLeft to -15 and ScaleTop to -15.

11. Single-click on the modem object in the form view. Leave the modem settings to the defaults, except for the following:

Name: SerialCom1
Comm Port: 1
Settings: 57600,n,8,1
RThreshold: 1

12. Double Click on the SendDAC button and insert the following code: This sends a space delimited string to the DSP

```
SerialCom1.Output = Chr(253) & txtDAC1.Text & " " & txtDAC2.Text & Chr(255)
```

13. Double click on empty space in your form window. Add the following code to the auto-generated function

Form_Load:

```
SerialCom1.PortOpen = True 'Enable the serial port when form is loaded
```

14. Arrow to the very top of your form code above all the other functions. Add the following global variables:

```
Option Explicit  
Dim strNextString As String  
Dim Adc1value As Single  
Dim Adc2value As Single
```

15. Go back to the form view. Double click on the MSComm object. Add the following code (cut and paste from the lab):

```
Private Sub SerialCom1_OnComm()  
  
    Dim intStart As Integer 'the index of StartChar  
    Dim intEnd As Integer 'the index of EndChar  
    Dim splitstrings As Variant  
    Dim Chars_received As String  
  
    If SerialCom1.CommEvent = 2 Then ' Event messages... only care about event 2 (receive)  
        strNextString = strNextString & SerialCom1.Input ' Append new com data  
        intStart = InStr(strNextString, Chr(253)) 'Find the index of StartChar  
        If (intStart > 0) Then  
            strNextString = Mid(strNextString, intStart)  
            intEnd = InStr(strNextString, Chr(255)) 'Find the index of EndChar  
            If (intEnd > 1) Then 'If Start&End exist and Start comes before End  
                Chars_received = Mid(strNextString,2, intEnd - 2)  
                splitstrings = Split(Chars_received, " ")  
                If Ubound(splitstrings) = 1 Then  
                    Adc1value = Val(splitstrings(0))  
                    Adc2value = Val(splitstrings(1))  
                    lblADC1.Caption = CStr(Adc1value)  
                    lblADC2.Caption = CStr(Adc2value)  
                    pbxArena.Circle (Adc1value,Adc2value) , 1  
                End If  
                strNextString = Mid(strNextString, intEnd + 1)  
            End If  
        End If  
    End If  
End Sub
```

Update DAC value to output from VB.

On the DSP side add this code in user_UARTFuncs.c. This task will wait for a full message and then use sscanf to read in the two values sent from VB. In your main C-file extern the variables newDAC1, newDAC2 and newDACdata. Each time the ADC interrupt function is called check to see if newDACdata is equal to 1. If it is, copy the values from newDAC1 and newDAC2 into variables that are passed as parameters to writeDAC7724.

```
// UART 1 GLOBAL VARIABLES
int    UART1sensordatatimeouterror = 0;    // Initialize timeout error count
int    UART1transmissionerror = 0;        // Initialize transmission error count
char  UART1receivechar;
int    UART1beginnewdata = 0;
extern far SEM_Obj SEM_UART1MessageReady;
int    UART1datacollect = 0;
char  UART1MessageArray[101];
float  newDAC1 = 0;
float  newDAC2 = 0;
int    newDACdata = 0;

void UART1ReceiveCharTask(void) {
    UART1sensordatatimeouterror = 0;    // Initialize timeout error count
    UART1transmissionerror = 0;        // Initialize transmission error count
    while(1) {                          // Loop forever
        // Wait forever for new character
        if ( checkfornewcharUART1(&UART1receivechar, SYS_FOREVER) ) {
            if (!UART1beginnewdata) { // Only true if have not yet begun a message
                if ((unsigned char)UART1receivechar == 253) { // Check for start
                    char
                        UART1datacollect = 0;          // amount of data collected
                    in message set to 0
                        UART1beginnewdata = 1;        // flag to indicate we are
                    collecting a message
                }
            } else { // Filling data
                // Dont go too large... limit message to 100 chars
                if ((UART1datacollect < 100) && ((unsigned char)UART1receivechar
                    != 255)) {
                    UART1MessageArray[UART1datacollect] = (char)
                    UART1receivechar;
                    UART1datacollect++;
                } else { // too much data or 255 char received
                    if ((unsigned char)UART1receivechar != 255) { // check if
                        end character recvd
                            // Not received
                            UART1transmissionerror++;
                    }
                    // Whether or not message terminated correctly, send data
                    to other tasks
                        UART1MessageArray[UART1datacollect] = '\0'; // Null
                    terminate the string
                        UART1beginnewdata = 0; // Reset the flag
                        UART1datacollect = 0; // Reset the number of chars
                    collected
                        sscanf(UART1MessageArray, "%f%f", &newDAC1, &newDAC2);
                        newDACdata = 1;
                }
            }
        }
    }
}
```

