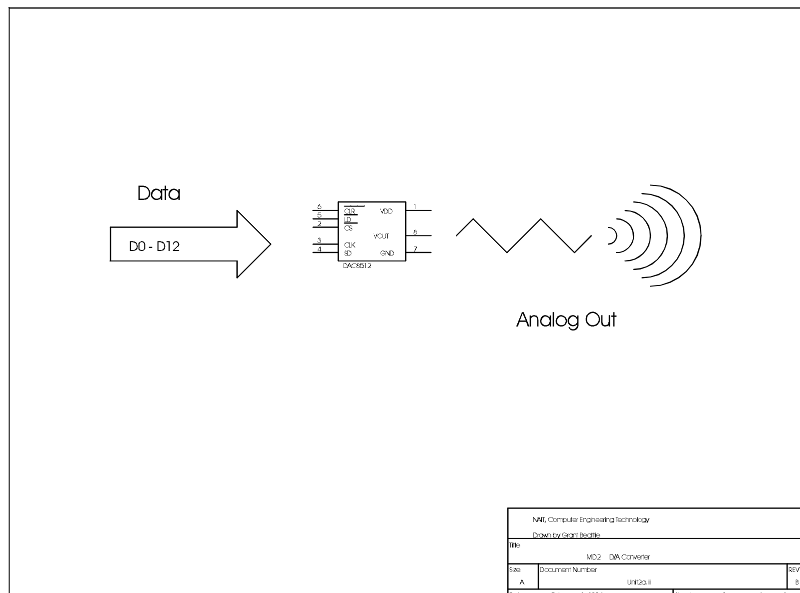
Micro Design 2

Lab #2



Digital to Analog Converters

Introduction

The use of a microprocessor in a given application depends on the translation of real life quantities into a form that may be understood by the binary world of the microprocessor. Many applications for computer based equipment involve analog quantities. An analog quantity is one that varies in a continuous fashion; such as pressure, temperature or velocity.

A designer of embedded microprocessor systems is faced with three major hardware challenges. First, appropriate sensors and signal conditioning circuitry must be chosen which can translate analog quantities into either current or voltage. The second task is translating the current or voltage into the digital domain. This is typically taken care of by the Analog to Digital Converter (ADC or A/D). Third, it is necessary to design a suitable circuit to connect the A/D to the CPU in a manner which is cost effective.

The purpose of the A/D is to encode information sent to it from a sensor (temperature, position, etc.) into a sampled binary format (often called Pulse Code Modulation). This format allows a wide variety of analog quantities to be sent to the microprocessor for evaluation or manipulation.

The A/D's counterpart is the Digital to Analog Converter (DAC or D/A). D/A's reverse the process and convert binary information back into analog voltage or current. These devices are commonly used in various control applications where the microprocessor is to intervene or affect the world around it.

In this lab you will design an interface circuit for a D/A converter. This interface circuit will be vastly different from the interface to RAM or EPROM because it will be implemented using the 68HC11's Serial Peripheral Interface (SPI). Thus, an investigation is required to understand the operation and purpose of the SPI. Further classes focusing on the actual D/A and A/D processes should give you a general understanding of the mechanisms within these devices.

LAB OUTLINE

PRE-LAB: D/A Circuit and SPI Interface Design

You will prepare the schematic for your D/A circuit in order to correctly interface it using the HC11's Serial Peripheral Interface.

SECTION 1: User Interface in Assembler

When creating embedded systems it is important not to forsake one programming language for another. You are expected to be completely bilingual. In the last lab you created a cgets( ) routine and some data conversion routines for your SCI_LIB.C library. It's time now to go back and catch up on our assembly language skills. You must create a similar user interface using the AS6811 assembler.

SECTION 2: Wiring and Testing the D/A Circuit

Section 2 will consist of wiring of your circuit and verifying it's operation. You must write a short calibration program in assembler which will be used to test the D/A over its complete operating range.

SECTION 3: D/A Programming in Assembler

Some simple waveform generation code will be written in AS6811 assembler to exercise your D/A and test its capabilities.

SECTION 4: D/A Programming in C

Some simple waveform generation code will be written in ICC11 C to exercise your D/A and test its capabilities.

Pre-Lab

D/A Circuit and SPI Interface

(   )   Step 1: D/A Schematic and SPI Interface Design

Your first step in adding any new hardware to the HC11 board is always to prepare a schematic diagram. This interface may be designed in many ways, but there are a couple of criteria for this lab:

i. Minimum connections to CPU. We would like to save as many CPU pins as Possible for future use.

ii. Safety. It is possible to design the D/A circuit so that the D/A automatically outputs zero volts whenever the CPU's $\overline{\text{RESET}}$ is asserted. Remember, $\overline{\text{RESET}}$ can be asserted by the Maxim supervisory chip or by the CPU itself! Why do we care? Because the output of the D/A might just be connected to something our life depends on.

With the above in mind, consider the following points when drafting your schematic:

- Connect the $\overline{\text{CLR}}$ input to $\overline{\text{RESET}}$
- **OR**, Connect the $\overline{\text{CLR}}$ input to a buffered $\overline{\text{RESET}}$ output (use pin 7 on the 16v8 as a RESET* input and pin 13 as a RESET* output).
- Connect the appropriate CPU SPI pins to CLK and SDI.
- Connect $\overline{\text{LD}}$ to the slave select output from the CPU **OR** use an unused 16v8 (chip select) output (pin 17 or 12).
- Permanently enable the DAC by tying $\overline{\text{CS}}$ low.
- The D/A should be tied to ANALOG ground (as in your class notes).
- See Figure 24 from the data sheets for proper power supply bypassing.
- The DAC output will go to the pad marked “A” next to the power supply input. Locate the DAC in that immediate area.

*** Have your schematic (pre-lab) checked off prior to construction!!! ***

Procedure

SECTION 1: User Interface Design in Assembler

( ) Step 1: Library Routines

Make sure that your assembler libraries contain all of the required routines from Micro 1. A list of the routines and their parameter requirements is listed in the appendix. Please review this information and make sure that your libraries are complete.

( ) Step 2: The RxString Routine

You are to write a library routine which operates in the same manner as cgets( ). The following sample call and subroutine header indicate the basic parameter requirements:

```
Main:       :                           ;(Prompt the User for 20 characters)
            :
            ldx     #Buffer             ;X points to buffered string input structure.
            ldaa    #21                 ;Space for 20 characters + trailing NULL.
            staa    0,X                 ;Place the string length in the first element of the
                                        ;structure.
            jsr     RxString            ;Get the string from the user.
            ldaa    1,X                 ;Check the string length...
            :
            :


Buffer:     .ds     23                  ;Reserve 23 bytes for string input structure



;RxString:  Using a RAM buffer, gets a string from the terminal keyboard.
;Requires:  X points to the RAM buffer.  0,X and 1,X are parameters  within the buffer.
;           [0] - First element tells RxString the number of bytes the user may enter (plus 1).
;Returns:   [1] - Second element tells the caller how many chars were actually entered.
            [2 ]. . .[ n+3] The body of the null terminated string.
```

( ) Step 3: Test Program

Write a short program which fulfills the following specification:

i. Initialize the SCI and PIA (all outputs).

ii. Prompt the user for three bytes (to be used as a 12 bit DAC value).

iii. Using RxString, get the input limiting the user to three characters max. After returning to the calling code, examine the input data and if three characters were not entered make the user try again (step ii).

iv. Once three chars have been accepted, convert them to a 16 bit value (the MSN will always be 0) and display them on the PIA test board. Then return to step ii. Your program should accept the characters 0-9, a-f and A-F. It is suggested that you Create two other short subroutines, AscToHex and UpCase (or StrUpr).

When your program is functional have your work checked off. Make sure you have a documented list file available for your instructor. You will not be handing this list file in.

SECTION 2: Wiring and Testing the D/A Circuit

( ) Step 1: DAC8512 Placement and Power Wiring

Wire the D/A in the location specified by your instructor. Since several other components must be added to the 68HC11 board it is important to place the D/A in its proper location.

A key factor in the interface of analog and digital circuitry is preventing noise from the digital circuits from interfering with the analog signals. Note the following suggestions:

i. Maintain as much physical separation between the analog and digital circuitry as possible.

ii. All capacitors should be located as close to the D/A chip as possible. You may find it convenient to actually solder the caps directly to the D/A socket. If you choose this route, complete all soldering before any wirewrapping is started!

Complete the wiring of your D/A according to the color conventions given in MD1. Double check all wiring for accuracy.

( ) Step 2: External DAC Wiring

The pad marked “A:” will be used to route the DAC signal off-board. Prepare a length of stranded wire as before and connect it to the “A” pad. Then add a vector pin and wire-wrap connection to the DAC output.

( ) Step 3: Augmented D/A Test Program

To test the operation of the D/A modify your previous test program to do the following:

i. Initialize the required hardware (SCI and SPI).

ii. Prompt the user to enter a 3 digit hex number to be sent to the D/A.

iii. Collect the digits using RxString/UpCase and convert the value into hex and send it to the D/A.

iv. Return to step ii or iii as required to allow the program to run in a continuous loop.

SPI INITIALIZATION NOTES:

1. See the example code from the data sheet. Although this code is utterly unacceptable, some details can be gleaned from it.

2. Make sure you set the correct pins as outputs on Port D. Initialize SCK= $\overline{SS}$ =MOSI = high.

3. The SPI should be initialized for the **highest** possible data transfer rate.

DAC_OUT SUBROUTINE NOTES:

1. Create a DAC_LIB.ASM or SPI_LIB.ASM (or whatever) that contains the DAC_OUT subroutine, the SPI equates and the usual title information.

2. Create a subroutine which performs the SPI data transfer. The data transfer must Be as fast as possible. Don't push or pull any registers! Write the subroutine without any loops! This means you will have to count cycles to determine how long to wait between each 8 bit data transfer. See the data sheet for sample code, but your code must NOT use the BPL WAIT scheme because the subroutine time must be constant and predictable (and as short as possible).

3. The subroutine header might look like:

```
;DAC_OUT:        Transmits the 16 bit value in acc D to the DAC via the SPI.  Only the 12 LSB's
;                are actually used.  The MSB is sent first, then the LSB is sent (which pushes out
;                the top four bits).  Finally, the LD line is pulsed.
;
;REQUIRES:       D contains the word to send to the DAC.
;RETURNS:        Nothing
;REGS AFFECTED:  ??? (You tell me!)
```

When the program is functional, begin by writing 000 to the D/A. Use a voltmeter to verify that the output voltage is 0.000 volts. Now write the value 0xFFF to the D/A. Use a voltmeter to verify that the output voltage is 4.095 volts. Try various values in between to verify the DAC's linearity and step size.

*** Have your D/A circuit checked off at this point! ***
You must hand in this documented list file for evaluation.

SECTION 3: D/A Programming in Assembler

Once the D/A is in working order we can proceed to put it through its paces. The following program will provide an interesting insight into the design of a programmable function generator.

( ) Step 1: Sinewave

Creating a sinewave with a D/A "on-the-fly" would require relatively intense and time consuming computations with square roots and pi. An alternative method is to employ a look-up table which holds a list of pre-calculated values.

Write a program to produce a sinewave based on a 32 point (every 11.25 degrees) look-up table. We will not be attempting to create a sinewave of a specific frequency, but your code must be efficient enough to produce a sinewave of **greater than 450Hz**.

The sinewave can be calculated as follows. Note that the sinewave must be offset or "lifted up" by approximately 2 volts since our D/A has been hooked up to be uni-polar.

Examples:

11.25 degrees: 2048 x sin(11.25) + 2047 = 2446.545
22.50 degrees: 2048 x sin(22.50) + 2047 = 2830.736
:
360.0 degrees: 2048 x sin(360.0) + 2047 = 2047.000

Therefore, the look-up table looks like (32 entries total):

```
SineWave:     .dw       2447, 2831, . . .
                        :
                        :
              .dw                                  . . . , 2047
EndOfTable:
```

Assemble and test the sinewave on your board. Your code must be efficient enough to produce a sinewave of greater than 450Hz. [HINT: Write your DAC_Out code in-line instead of calling the subroutine. This will save the time spent in the JSR and RTS.]

( ) Step 3: Reconstruction Filter

Using a spare 3k3 resistor (or similar) and 0u1 capacitor, temporarily construct a crude lowpass filter (LPF). Attach the LPF to the D/A output and compare the waveform at the DAC output to the waveform at the filter output. Answer the question below.

Q1. What has happened to the sinewave as a result of the LPF?

*** Have your program checked off at this point! ***

Provide cycle counts in the comment field of your file! Also, you must show how the cycle counts add up to establish the particular waveform and the particular frequency!.

SECTION 4: D/A Programming in C

( ) Step 1: Sinewave Function

Creating a sinewave in C requires a slightly different approach. Using the following specifications/ suggestions, create a 32 point sinewave that has a **minimum frequency of 225Hz**:

i. You must create a DacOut( ) function that fits the prototype below. You cannot Place the DacOut( ) code “in-line”. The DacOut( ) code and your SPI equates must Be placed in a DAC_LIB.C (or SPI_LIB.C or whatever).

```
void DacOut(unsigned int dacval);
```

ii. The body of the DacOut( ) code may be created in pure C or in-line assembly. Again, avoid loops polling on SPIF.

iii. The sine table may be created as a table local to main, as a global table or as a function containing a table built using inline assembly (other ideas??).

*** Have your program checked off at this point! ***

LAB #2: LABCHECK SUMMARY

NAME:________________________________

SECTION: _____________________________

PRE-LAB: Schematic

Date: ____________________ Instructor: _____________________ /5

LABCHECK #1: User Interface in Assembler

Milestone Checkoff (do not hand in this list file).

Date: ____________________ Instructor: _____________________

LABCHECK #2: Wiring and Testing the D/A

Verify program and D/A operation	/10
Wiring	/5
Program Documentation	/10

Date: ____________________ Instructor: _____________________

LABCHECK #3: Sinewave in Assembler

Sinewave Operation	/5
Frequency: ______________	
Program Documentation (including calculations)	/5

Date: ____________________ Instructor: _____________________

LABCHECK #4: Sinewave in C

Sinewave Operation	/5
Frequency: ______________	
Program Documentation	/5

Date: ____________________ Instructor: _____________________

TOTAL: /50

APPENDIX

Required Library Subroutines

Analog Devices DAC-8512
Data Sheets
(Separate PDF File)

Required Library Subroutines

Your library subroutines must use only LOCAL variables (the stack) or GLOBAL variables whose addresses are passed from main (pass-by-reference). Static named variables (eg. TEMPBYTE) must never be used in your library because your library must be re-entrant. The reasons for this will be covered in Lab #7.

Your PIA_LIB.ASM file (or equivalent) must contain the following:

```
TITLE BLOCK - A descriptive title block and the .title directive.

PIA EQUATES - Hardware equates for the PIA.

;DELAY           Provides a 0.01 second delay for each count in the
;                X index register.
;REQUIRES -      X contains the desired delay count (eg. in hundredths).
;RETURNS -       Nothing is returned to the calling program.
;REGS AFFECTED - None
```

Your SCI_LIB.ASM file (or equivalent) must contain the following:

```
TITLE BLOCK - A descriptive title block and the .title directive.

SCI EQUATES - Hardware equates for the SCI.

;ASCToHex        Converts the two ASCII bytes in accumulator D into one hex byte
;                in accumulator A.
;REQUIRES -      A holds the ASCII MSB.
;                B holds the ASCII LSB.
;RETURNS -       A holds the hex result
;REGS AFFECTED - ?? Does your version trash B?


;HexToASC        Converts the hex byte in accumulator A into two ASCII bytes
;                in accumulator D.
;REQUIRES -      A contains the byte to convert.
;RETURNS -       A holds the ASCII MSB
;                B holds the ASCII LSB
;REGS AFFECTED - No other registers affected


;KbHit           Checks the status of the SCI to see if a new character has
;                arrived.
;REQUIRES -      Nothing
;RETURNS -       If a new byte is in, C flag = 1
;                If no byte is waiting, C flag = 0
;REGS AFFECTED - None
```

```
;RxByte           Receives a byte from the terminal via the SCI.  Waits for a
;                 keypress, but no echo.
;REQUIRES -       Nothing
;RETURNS -        Received character in accumulator A
;REGS AFFECTED - No other registers affected


;TxByte           Transmits a character to the terminal via the SCI.
;
;REQUIRES -       Character to transmit must be in accumulator A.
;RETURNS -        Nothing
;REGS AFFECTED - None


;TxString         Transmits a NULL terminated string via the SCI.
;
;REQUIRES -       X must point to the start of the string.
;RETURNS -        Nothing
;REGS AFFECTED - None.
```

You may add initialization routines to your library if you wish.