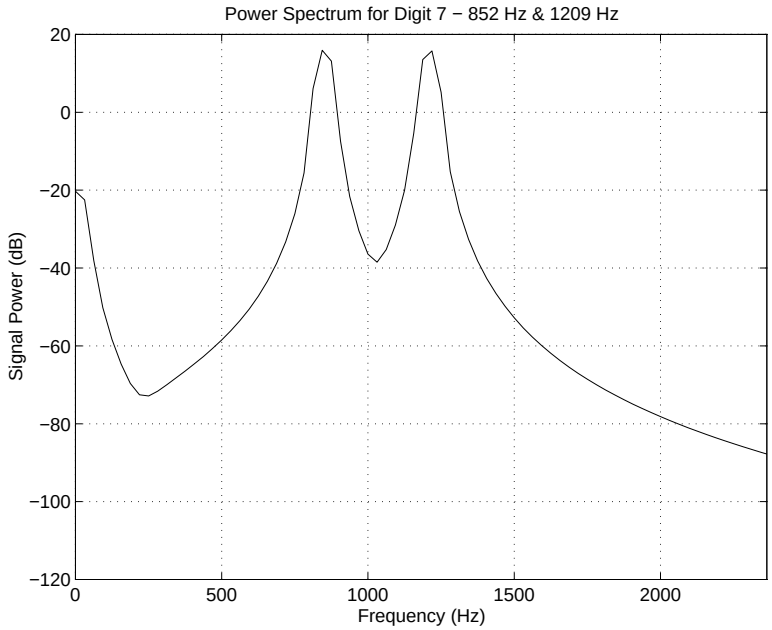
DTMF Encoding & Decoding

An application of The Goertzel Algorithm



Electronics IV (Honours) Project
1994

by Steven J. Merrifield
Supervised by Dr. Chris Dick

This thesis was awarded the *Nokia Telecommunications Electronic Engineering Award for the Best Final Year Thesis in the Faculty of Science and Technology* for the year 1994.

School of Electronic Engineering
La Trobe University
Bundoora 3083
Victoria
AUSTRALIA

Contents

List of Figures

Acknowledgements

I wish to thank my supervisor, Dr. Chris Dick, for his invaluable assistance throughout the year. His advice and encouragement was a great asset, and without it, this project would never have come to fruition. Chris' generosity with his time, and provision of data sheets and manuals was greatly appreciated. Mr. Geoff Liersch also deserves a big thank-you, providing a great deal of assistance with both hardware and software debugging. His help with configuring the Tektronix logic analyser was invaluable, and his knowledge of microprocessor systems is incredible. Thanks Lurch! I would also like to thank Steve Lutrov, of *The Software Parlour BBS*, for providing the low-level C routines used in the PC front-end software.

Chapter 1

Introduction

This project involved the implementation of a fixed point DSP processor to send and receive DTMF tones. The TMS320C25 by Texas Instruments was used to create a general purpose development system, utilising ROM, RAM, and a universal serial interface. This system could then be used for a number of digital signal processing applications, but DTMF encoding/decoding was chosen in this case. The advantages of implementing a DTMF coding routine with a DSP processor are its speed and its flexibility. A dedicated DTMF chip is hard-wired to send and receive only a certain number of fixed tones. The project presented here, on the other hand, can send and receive any number of tones simply by altering the keypad lookup table. This results in a much wider application range, and offers increased security for data-sensitive applications.

Chapter 2

DTMF

DTMF, or Dual Tone Multi Frequency, is a method of sending and receiving control information over a communications channel. The reader is probably most familiar with DTMF tones as heard on a modern push-button telephone. Each digit on the keypad is encoded as a DTMF tone, which is then transmitted over a medium, and decoded at the receiving end. A keypad as shown in Figure 2.1, is usually used to generate the required DTMF tone. Each key has associated with it a row frequency, and a column frequency. When a key is pressed, the encoding circuitry mixes together these two frequencies, and transmits the result. The receiver then decodes the tone back into its two respective frequencies, and then the processing circuit will act accordingly.

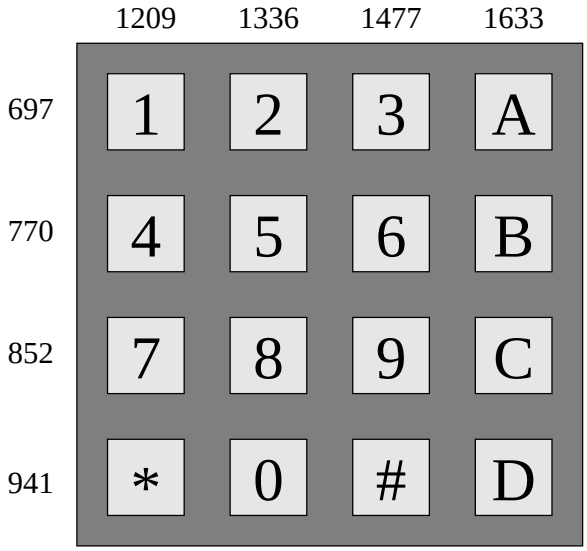


Figure 2.1: DTMF Keypad

This project reads digits entered on an IBM PC keyboard, encodes the digit into its relevant tones, and transmits it over a cable. The receiver then decodes the tones and sends it back to the IBM PC where the digit is displayed on a monitor.

2.1 Applications

Typically, DTMF coding is used by the telecommunications industry for control applications, such as exchange signaling, and remote process control. DTMF coding systems are also used widely in other scientific areas. Applications such as remote data acquisition from a mountain top weather station, or electronic banking, where a customer sends information using a telephone keypad. The applications of DTMF coding are many and varied, but all require a transmitter which encodes the required tones, and a receiver which decodes the tones into relevant information.

2.2 Encoding

The DTMF encoding program implemented on the TMS320C25 used two look up tables. One to determine which key was pressed on the keyboard, and hence, which tones to generate, and secondly, a look up table of sine values required to synthesize the necessary frequencies. The sine wave generation routine was based on that detailed in [3], but was modified slightly for the TMS320C25. It was also necessary to replicate the generation section of code in order to produce two sine waves. These two waves were then added together, and sent to the analog interface circuit ready for transmission.

2.3 Decoding

Decoding DTMF tones involves the detection of two specific frequencies. As the DTMF encoding scheme uses a 4x4 frequency matrix, the detector need only search for these eight particular tones.

The correct detection of a valid DTMF digit must ensure that there is a minimum energy value at both of the required frequencies. If for example, the detector only finds an energy peak at one of the required frequencies, the tone received was not a valid DTMF digit. The detection of a single frequency could be caused by a multitude of occurrences, ranging from human speech through to random noise. In this particular application, the Goertzel algorithm was used for detection.

2.3.1 The Goertzel Algorithm

The Goertzel algorithm is a special case of the Discrete Fourier Transform (DFT), where the DFT defining equation is given by:

$$X(k) = \sum_{i=0}^{N-1} x(i) W_N^{ik} \qquad k = 0, \ldots, N-1 \tag{2.1a}$$

where

$$W_N = e^{\frac{-j2\pi}{N}} \tag{2.1b}$$

The Goertzel algorithm makes uses of the fact that the phase factors, W_N^k, are periodic, and thus the DFT equation can be expressed as a linear filtering operation.

The transfer function for a single pole filter is defined as:

$$H_k(z) = \frac{1}{1 - W_N^{-k} z^{-1}} = \frac{N(z)}{D(z)} \tag{2.2}$$

This filter has a pole on the unit circle, at the frequency $\omega_k = \frac{2\pi k}{N}$. Thus the entire DFT can be computed by applying the input data to a bank of single pole parallel filters, each having a pole at the corresponding DFT frequency. The filter recurrence relation can be determined by taking the inverse z-transform of (2.2) If we let $X(z)$ be the z-transform of the filter input sequence, and $Y_k(z)$ be the z-transform of the filter output, then

$$H_k(z) = \frac{Y_k(z)}{X(z)} = \frac{1}{1 - W_N^{-k} z^{-1}} \tag{2.3}$$

where the k subscript denotes the k^{th} DFT coefficient.
By re-arranging (2.3) and taking the inverse z-transform, we arrive at

$$y_k(i) = x(i) + y_k(i-1) W_N^{-k} \tag{2.4}$$

This is the defining equation for a single pole resonator, with output $y_k(i)$. An N point DFT could thus be implemented by using a parallel arrangement of such filters, where each filter calculates a single DFT coefficient. The signal flow diagram for such a filter is shown in Figure 2.2.

The filter shown in Figure 2.2 must calculate a complex multiplication for each recursive pass. This is inefficient, and can be eliminated by transforming the single pole filter into a double pole resonator. The transfer function for such an implementation is given by

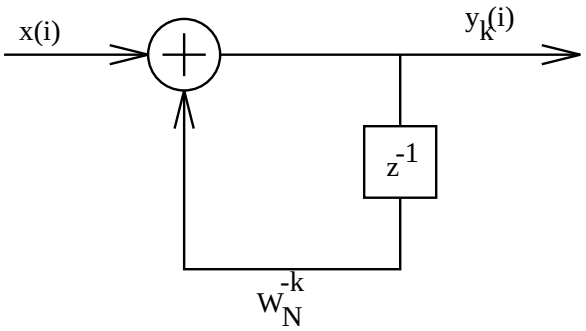


Figure 2.2: Signal flow graph for first order recursive computation of the DFT

$$H_k(z) = \frac{1 - W_N^k z^{-1}}{1 - 2\cos(\frac{2\pi k}{N})z^{-1} + z^{-2}} \tag{2.5}$$

The direct form II realisation of this filter can be represented by the difference equation

$$v_k(i) = 2\cos(\frac{2\pi k}{N})v_k(i-1) - v_k(i-2) + x(i) \tag{2.6}$$

$$y_k(i) = v_k(i) - W_N^k v_k(i-1) \tag{2.7}$$

and is shown in Figure 2.3.

The recursive relation, $v_k(i)$ is calculated for $i = 0, 1, \ldots, N-1$, but the final equation is only calculated once, when $i = N$.

The Goertzel algorithm is more efficient when only a small number of points need to be calculated. In this case, a parallel arrangement of Goertzel second order filters, as shown in Figure 2.4, is usually implemented. For DTMF detection, it is only necessary to implement the filters which correspond to the required eight frequencies. By doing this the Goertzel algorithm makes a huge time saving over a more conventional DFT decoder. If a DFT decoding scheme was implemented, using a transform length, of say 256, then all 256 points would need to be calculated in order to determine the required eight outputs. The values which are not required are then discarded - a huge waste of processing time. The Goertzel algorithm, on the other hand, only calculates the required coefficients, resulting in a much more efficient process.

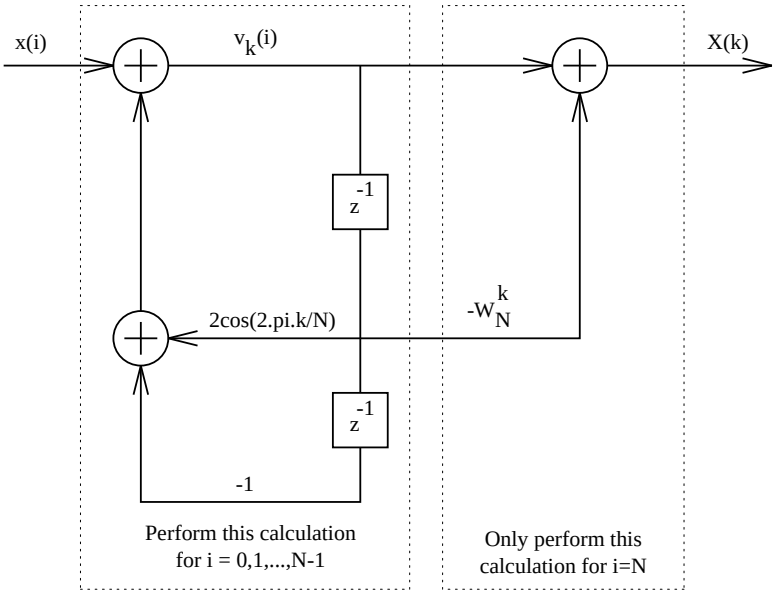


Figure 2.3: Signal flow graph for second order recursive computation of X(k)

2.3.2 Software Description of the Goertzel algorithm

The Goertzel algorithm takes the form of a series of second-order infinite impulse response filters. As can be seen in Figure 2.3, the signal flow graph is divided into two separate sections. The left hand part which includes the two delay elements, and the right hand side where there is no feedback. For DTMF decoding, it is really only the last iteration $(N-1)$ of the algorithm which is required. As a result, there is no need to execute the right hand side until the last iteration. What is not obvious though, is the fact that the multiplier of the left hand side, $2\cos(\frac{2\pi k}{N})$, is the same as the right hand side constant, W_N^k, when the absolute magnitudes are taken. W_N^k is a complex number, and the left hand side multiplier is a real number. However, the software calculates the magnitude squared of the output, hence the Goertzel algorithm adapted to DTMF decoding executes more quickly, and occupies less memory space since there is a reduction in the number of variables required.

This algorithm is compact and requires only one real coefficient for each frequency to determine its magnitude. In order to extract both magnitude and phase, complex coefficients are required, and hence more in-depth programming, but fortunately, DTMF tones may be decoded simply by extracting the magnitude of the two respective frequency components, and ignoring their phase. In addition to this, the program processes each sample as it

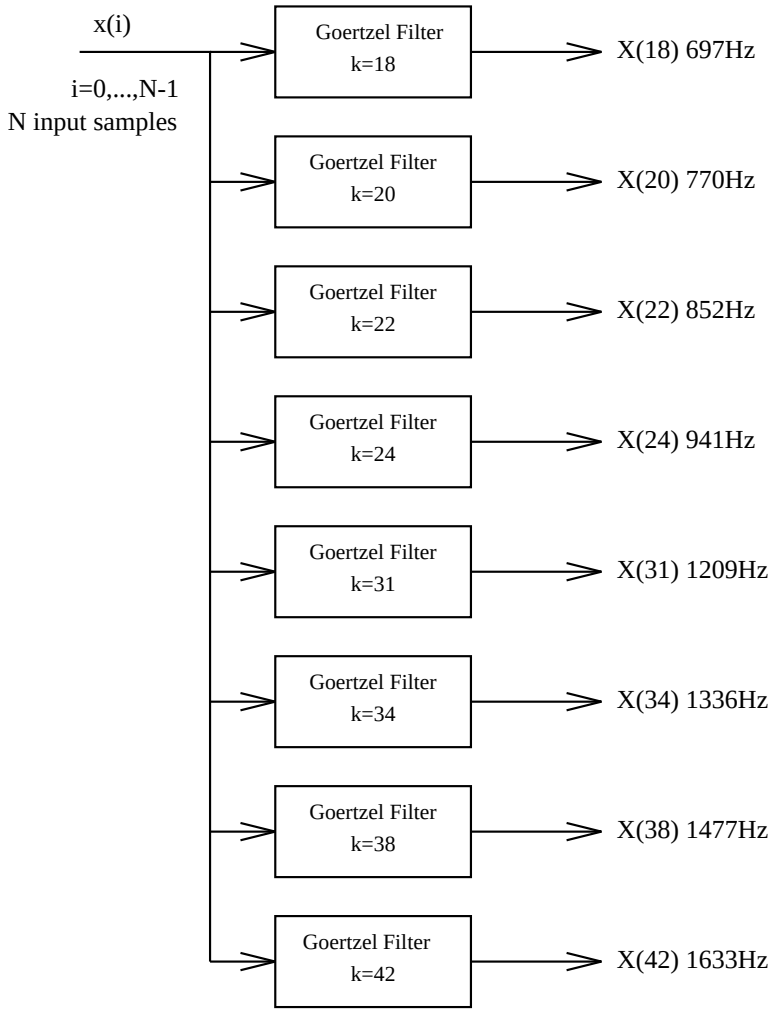


Figure 2.4: Parallel second order filter bank for selective DFT computation

arrives, instead of waiting for a complete set of samples.

The procedure used to implement the Goertzel algorithm is shown in Figure 2.5. After the main DFT loop has processed 205 samples and calculated the energy at each of the eight keypad frequencies, it then performs a series of tests. These tests are designed to discriminate between true DTMF tones, and other signals which may have similar spectras. Since the decoder processes its input data continuously, it does not know if a digit is valid until after it has processed all the data, and performed these tests. The first test checks to see if the decoded digit has changed since the last pass. If it has changed, then it moves the last digit into the second last position, and the current digit into the last position and repeats the DFT. If the current digit

was the same as the last digit, it then compares it with the second last digit. If these are also the same, it concludes the digit has not changed, and so branches back to the top of the algorithm. If however, the current digit is not the same as the second last digit, the program concludes that the digit is a new tone, and sends it back to the PC to be displayed. It then moves the last position into the second last position, and the current digit into the last position, and repeats the DFT.

It was also necessary to check the signal strength of the decoded tone to ensure that random interference or white noise had not been decoded. This was simply a matter of testing the relevant column and row energies to determine if they exceeded a pre-determined value. If the energy was less than this level, then the program branched back to the start of the algorithm.

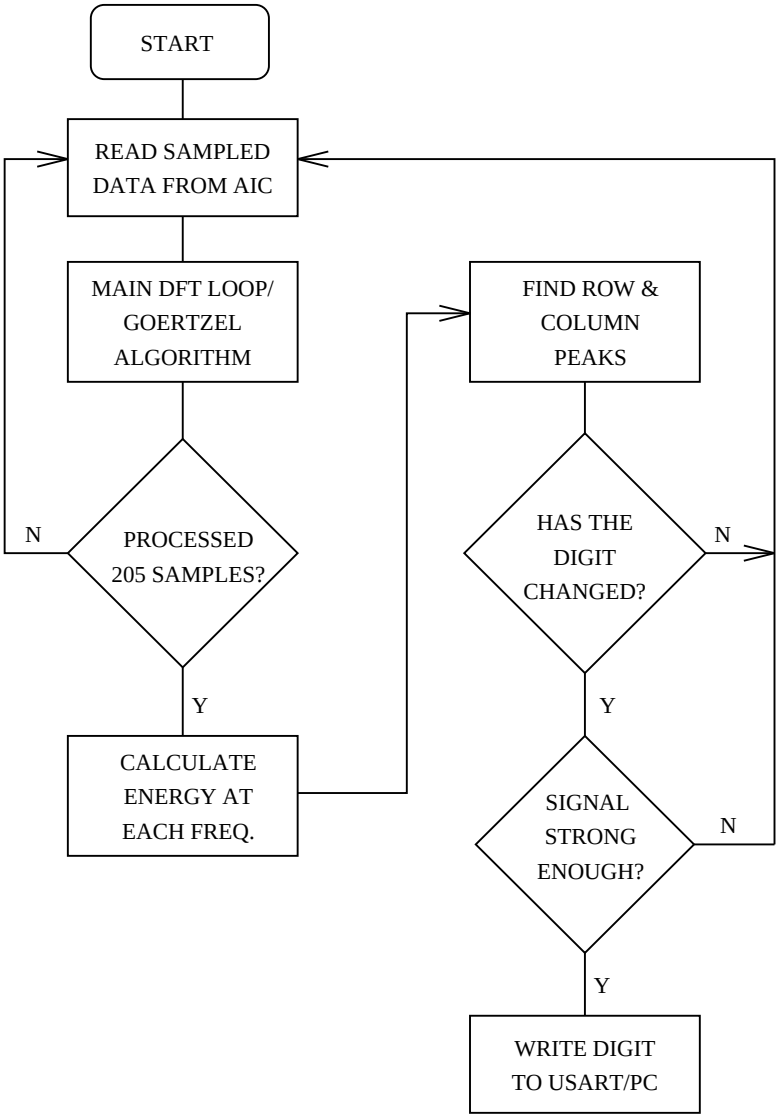


Figure 2.5: Flowchart for DTMF decoder

Chapter 3

Hardware Development

The DTMF decoder prototype was constructed using a wire-wrap technique. This construction method has a number of advantages over a PCB implementation. It allows an infinite number of changes and alterations to be made, it is much faster to build, and easier to debug since a color coding scheme can be implemented. In this case, the address bus and data bus were wired to correspond to the standard resistor color codes, ie, A0 was black, A1 brown, and so on.

Extensive use was made of bypass capacitors on the supply rails, and close to each IC. This was done to alleviate any possibility of ground-bounce, or switching transients, affecting the correct circuit operation. Without such bypassing, glitches would have been likely to occur.

3.1 System Overview

The DSP development system was based on Texas Instrument's TMS320C25 fixed point processor. This processor was designed specifically for digital signal processing applications, and has specialised internal hardware for specific DSP operations. This approach has the advantage of being much faster than a software based approach and makes programming DSP specific applications much simpler.

To make the development system fully versatile, a full complement of program RAM, data RAM and ROM was provided on board. A bank of diagnostic LEDs was also provided as a visual indication of a program's status.

A development system is not very useful if an EPROM must be programmed and erased every time a new program is required, so a serial interface to an IBM PC was provided to allow programs to be down loaded from

the PC directly to the TMS320C25's external program RAM.

In order to transmit and receive DTMF tones, an Analog Interface Circuit, or AIC, was connected to the peripheral serial port of the TMS320C25. This converted the digital sequence of data from the processor into analog signals for transmission, as well as receiving an analog signal, and converting it back into the digital domain for processing.

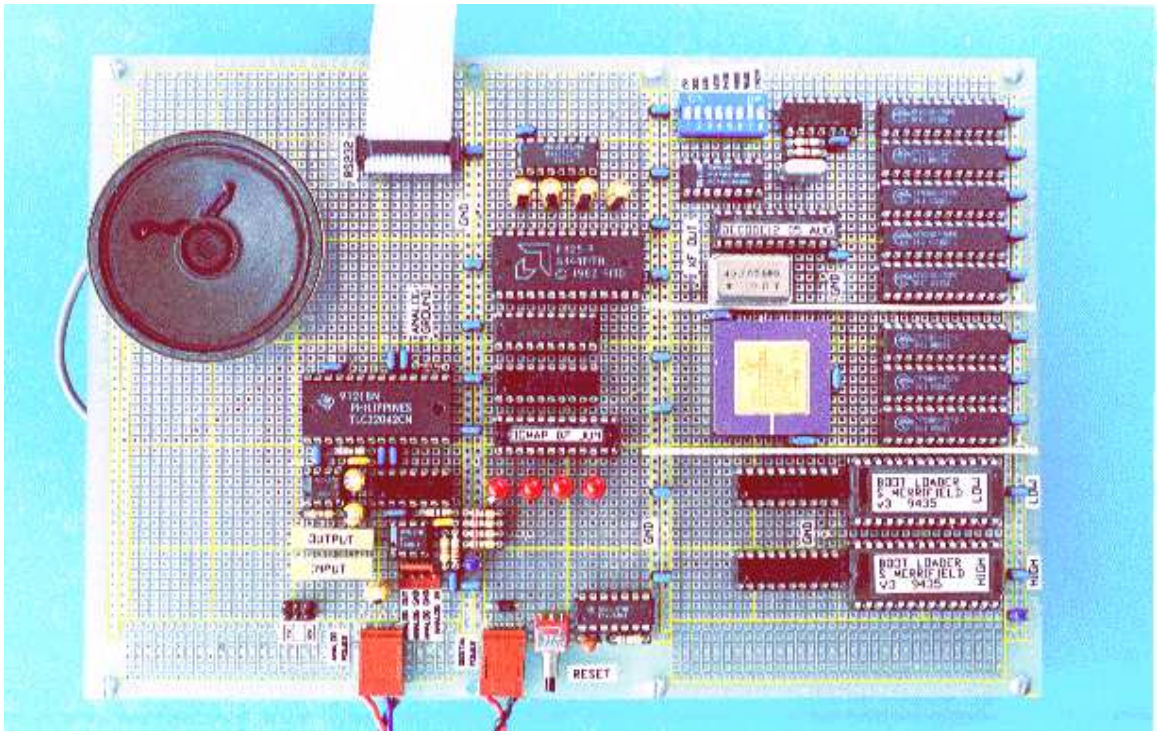


Figure 3.1: Photograph of the TMS320C25 DSP system

3.1.1 TMS320C25

The Texas Instrument's TMS320C25 is a second generation digital signal processor, with a specialised DSP instruction set, and a Harvard-type architecture. This style of architecture has separate program and data address spaces, providing an enormous speed and flexibility advantage over other general purpose processors.

Some of the key features of the TMS320C25 are listed below.

- 100ns instruction cycle time
- 544 word on-chip data RAM

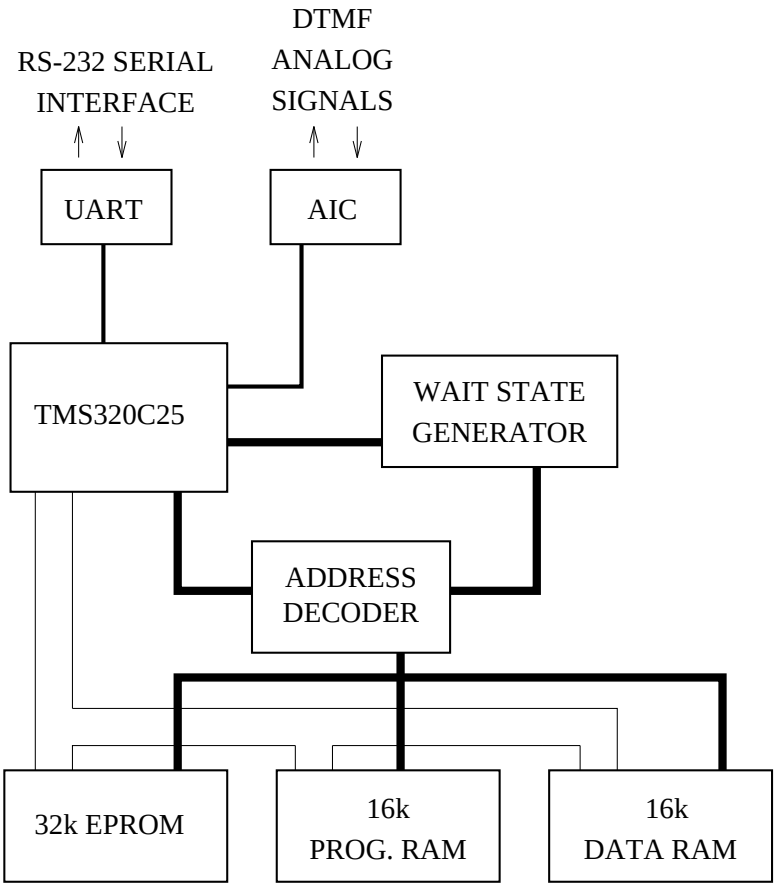


Figure 3.2: System Block Diagram

- 32 bit accumulator
- Block moves for easy data/program space transfer
- Eight level on-chip hardware stack
- Automatic provision for one wait state generation
- Serial port for direct codec/AIC interface
- Specific DSP instructions for bit-reversal, adaptive filtering etc.

These features, along with many others make the TMS320C25 particularly attractive for signal processing applications. At the same time though, general purpose applications are greatly enhanced by the large address spaces, multiple interrupt structure, serial port, provision for external wait states, and the capability for multi processor interfacing and direct memory access.

3.1.2 EPROM

The TMS320C25 required that the ROM be mapped into the bottom section of program memory, since the boot vectors and interrupt table occupy addresses `0x0000` through to `0x0020`. The 27C256-12 is a 32k x 8 EPROM with an access time of 120ns. Two of these devices were necessary to construct 32k of EPROM space, since the TMS320C25 has a 16 bit data bus. With these particular EPROMs, the data output turn off time was too slow, and so bus clashes would have resulted. This potential problem was overcome by the addition of 74F244 buffers which disabled the EPROM data bus when it was not selected.

3.1.3 SRAM

Static RAM was chosen for the TMS320C25 development system because of it's fast access time, and ease of use. Dynamic RAM, although less expensive, is more difficult to use since it requires refreshing at regular intervals. SRAMs, on the other hand, are virtually foolproof. The CY7C166-25, by Cypress Semiconductors, is a 16k x 4 bit SRAM, with an access time of 25ns. With a 16 bit bus, four of these IC's were required to make a full 16k of addressable memory space.

The memory configuration of the TMS320C25 is such that program space and data space are mapped into different areas. As this is the case, 4 x CY7C166's were used for data memory, and 4 x CY7C166's for program memory.

3.1.4 USART

The Intel 8251A Universal Synchronous / Asynchronous Receiver and Transmitter was chosen to interface the TMS320C25 development system with a standard IBM PC serial port. This USART operates at asynchronous baud rates from 150 baud to 19200 baud, depending on the clock input. The clock circuit chosen was a simple crystal oscillator, running at 2.4576MHz, then divided by 2,4,8,16, etc. using a 74HC4046 12-bit binary counter. The actual baud rate was adjusted using a DIP switch to set the appropriate clock speed.

A 74LS373 8 bit latch and a 74LS245 buffer were necessary to isolate the main system data bus from the USART data bus. The USART required that the data be held for a minimum of $t_{WD} = 20$ns after $\overline{WR}$ had gone high. The latch was used to hold the data on the bus until the next write cycle, thus satisfying the USART requirements. See Figure 3.3. Here the R/W line from

the TMS320C25 goes low, followed soon after by the IO select line. UARTW was generated using a PAL22V10, and the latch enable input (LE) to the 74LS373 is simply an inversion of the UARTW line. Hence, when writing to the USART, the 74LS373 will allow data to flow through, latching the data on the falling edge of LE.

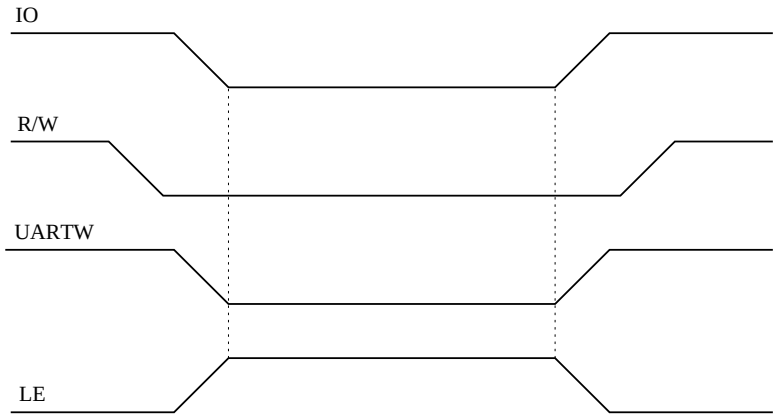


Figure 3.3: USART Write Cycle

During a read cycle, the latch outputs must be driven to a high impedance state, and the 74LS245 buffer then transfers data from the USART data bus to the main system data bus. The timing diagram for this operation is shown in Figure 3.4. The output enable input (OE) to the 74LS373 was tied directly to the R/W line from the TMS320C25. Thus, when the processor is executing a read from the USART, the buffer outputs will be tri-stated, and data can be read from the USART bus, without the possibility of conflictions. The UARTR line was generated in the same PAL as the UARTW line.

3.1.5 PALs

PALs were used in the place of discrete logic devices for several reasons.

- The design can be altered very easily by changing one IC, rather than reconnecting a large number of discrete ICs.
- A PAL occupies much less board space than a collection of discrete ICs.
- The propagation delay through a PAL is very much smaller than that due to a chain of discrete ICs, resulting in faster operation.
- Construction is simpler.

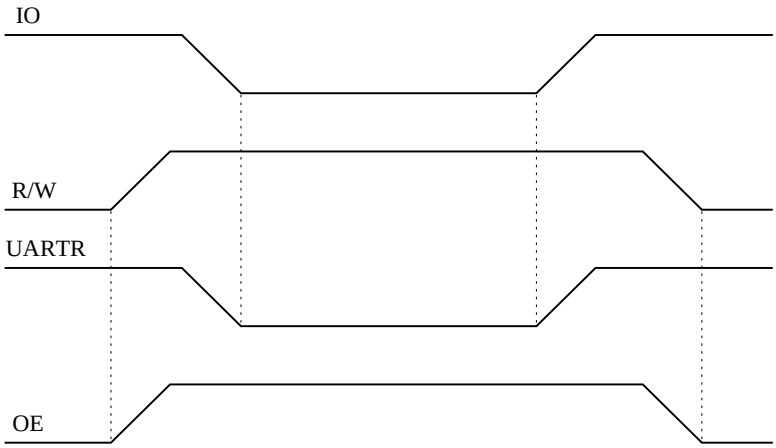


Figure 3.4: USART Read Cycle

Initially, a PAL16L8 was used for address decoding and READY generation, and a PAL16R4 for IO addressing. Both these ICs are not very versatile, in that they are not reprogrammable. It was inevitable that changes needed to be made, and this necessitated a new PAL. After several changes, it was decided to remove both PALs, and replace them with a more modern, erasable PAL22V10.

One PAL22V10 was used for generating the READY signal and handled all PRAM, DRAM, and EPROM control lines. A second PAL22V10 was used for implementing the USART control lines, as well as providing a latched output for the diagnostic LEDs.

The software package PALASM, was used for creating the JEDEC files to be down loaded to a PAL programmer. A fairly significant amount of time was spent learning how to use this program, which can be used to simulate the output of a PAL by configuring the appropriate inputs. A design can therefore be fully debugged before programming the PAL.

3.1.6 AIC

The TLC32042 Analog Interface Circuit includes both analog to digital, and digital to analog converters in the one package. This device incorporates a bandpass switched capacitor antialiasing filter, a 14 bit conversion process for both the ADC and DAC, and a lowpass switched capacitor output reconstruction filter. In addition, the AIC also provides a direct serial interface to the TMS320C25.

Use of this AIC greatly simplified the hardware necessary to provide an analog interface to the DSP development system.

AIC Initialisation

In order to set the sampling frequency to 8kHz, the internal registers of the AIC must be programmed. The programming sequence differs vastly from normal data transmission. In order to access the internal registers, the bottom two bits of a primary data transfer must be set, ie, the program must send `03h`. The AIC recognises that the bottom two bits are set, and then initialises secondary communication. It is during secondary communication that the actual initialisation data must be sent from the TMS320C25 to the TLC32042.

This process proved to be quite a stumbling block during the development of the DTMF decoding system. Normal primary communications could be initialised, and data transferred to the DAC registers without any problems, but the internal registers could not be accessed using secondary communications.

By sending `03h` as the primary data, secondary communication mode was then entered, but the initialisation data was not being sent. This could be seen very easily by triggering the logic analyser at the commencement of primary transmission. It was observed that the same data, ie, `03h`, was also being sent as the initialisation data.

This meant that the code which loads the secondary data was not executing fast enough, since the next interrupt occurred before the data was ready. The specifications for the TLC32042 required that the secondary data be sent 4 shift clock cycles after the conclusion of the primary transmission. With a 40MHz clock this allows a maximum of 16 CPU clock cycles in which to branch to the interrupt service routine, and prepare to transmit the data.

An interrupt occurs as the last bit is transferred from the TMS320C25 to the AIC. When an interrupt occurs, the processor branches to the interrupt vector table in ROM, which then points to a replica table in data RAM, and this points to the actual service routine. The first instructions in the ISR must save the status registers and accumulator, then transmit the data. This whole procedure must take place within the allowed 16 clock cycles.

The ROM in the development system operates with one wait state, further increasing the time taken to process an interrupt. The total time required when all these considerations were taken into account was 17 clock cycles, which is only slightly greater than that allowed. This explains why the secondary data was not being written consistently. The initialisation data could be sent on rare occasions, since the setup time was very close to the maximum allowed, although such a system is not very reliable.

To overcome this problem, it was necessary to divide the AIC master clock frequency using a flip-flop circuit, which would increase the amount

of time available for processing the interrupt. By doing this, we effectively now have 32 clock cycles in which to prepare for data transmission, and the program is able to service the interrupt within this time period.

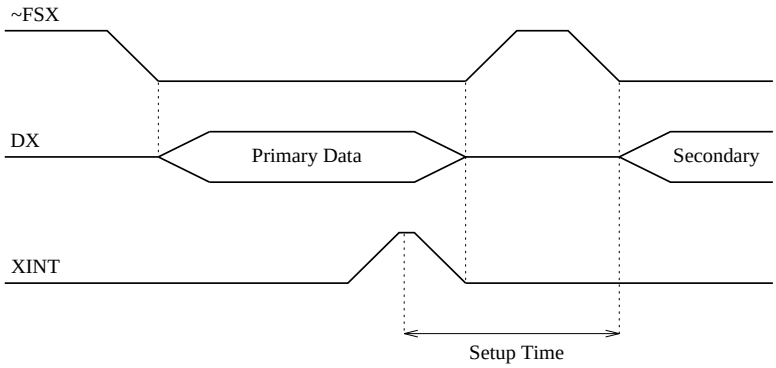


Figure 3.5: AIC Initialisation Timing

The interrupt code used initially was adapted from [10], and would not work at all. It was eventually concluded that the one wait state in the ROM was the problem, as the example code must have used zero wait states on all memory accesses. The extra delays in the branches from the one wait state ROM to the RAM were causing the program to exceed the maximum allowable delay of 16 CPU clock cycles. The time taken by the interrupt service routine in [10] was exactly 16 clock cycles, although this was not documented, leading future users to believe that their code was adaptable to third party applications. In actual fact, the frequency of the master clock input will probably need to be reduced.

3.2 System Memory Maps

The TMS320C25 can address a total of 64k of program space and 64k of data space through the use of separate $\overline{\text{PS}}$ and $\overline{\text{DS}}$ control lines. In this system, there is only 16k of program RAM and 16k of data RAM installed, although both these areas are mapped into 32k segments. This results in images, and so anything addressed in the upper 16k will be mapped to the lower half of the segment.

3.2.1 IO Space Memory Map

The IO space has a very simple address map, since there are only four LEDs and the USART in this space. The LEDs were mapped into the lower four

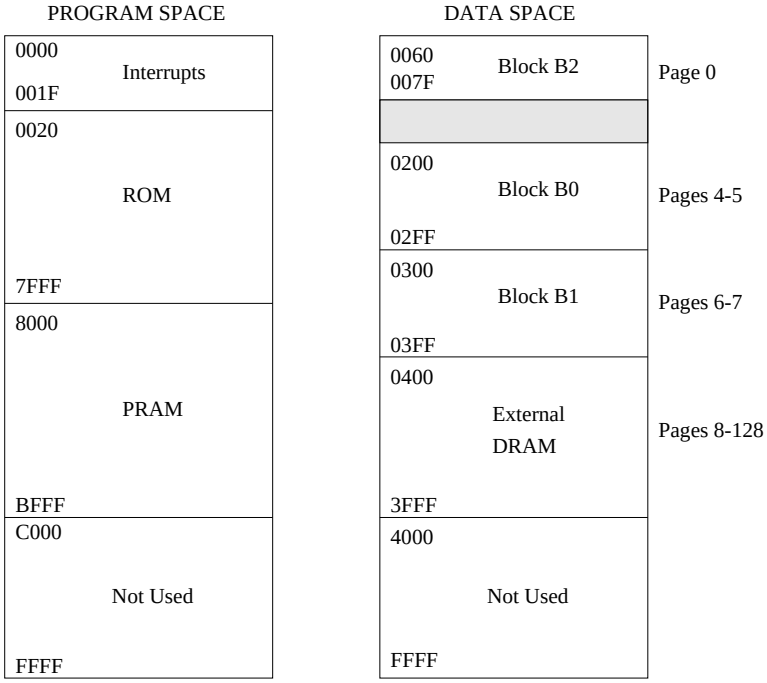


Figure 3.6: System Memory Maps

addresses (0-3), with the USART data register at address 4, and the control/status register at address 5. Again, images will result since incomplete address decoding was not implemented, although this is not significant.

3.3 Wait States

Wait states were necessary when interfacing the relatively fast TMS320C25 with slower peripheral ICs. In this case, wait states were required for accessing the EPROMs, and the USART. Both these peripheral chips have slow access times (relative to the processor), and hence the processor must be told to wait until the peripheral chip has finished doing its job, either reading a program from ROM, or transferring data via the PC serial interface.

The number of wait states depends on the total access time required. This time must include not only the relevant IC access times, but also any propagation delays in address decoding logic, as well as the logic necessary to generate the READY signal.

The number of wait states, N, can be found using the equation :

$$[100(N-1)+40]ns < t_a \leq (100N+40)ns \tag{3.1}$$

A3	A2	A1	A0	Address	Device
X	X	0	0	0	LED 0
X	X	0	1	1	LED 1
X	X	1	0	2	LED 2
X	X	1	1	3	LED 3
X	1	0	0	4	USART data in/out register
X	1	0	1	5	USART control/status register

Table 3.1: IO Address Map

where t_a is the total access time as outlined above.

3.3.1 EPROM Wait States

The worst case, total access time required by the EPROM circuitry was calculated as being 140ns, allowing the use of one wait state. The TMS320C25 provides a *microstate complete* output which, when gated with the relevant control lines, provides the automatic generation of one wait state. This allowed for a much simpler design, and reduced the amount of hardware required.

3.3.2 USART Wait States

The USART has a delay of 200ns from when the read line goes low, to when the data appears on the bus. This delay time, along with the decoding and latch/buffer logic delays meant that the total access time was 252ns. Hence, three wait states were required.

A number of difficulties were encountered in implementing three wait states. A PAL22V10 was chosen for this design, but this IC only contains D type flip-flops, whereas the example wait state generator shown in [5] used JK flip-flops. A significant amount of time was spent trying to adapt the example to the required circuit, but with no success. It was eventually decided that the example was proving too difficult to adapt to the PAL22V10, so another tact was necessary.

The final PAL equation was chosen by drawing the necessary waveforms, and using Karnaugh maps to design the required circuitry. The inputs were then setup, and the design was simulated using PALASM.

The waveforms associated with the wait state generator are shown in Figure 3.7. The Q1, Q2, and Q3 waveforms are internal to the PAL, and

the UART waveform incorporates the relevant chip select lines, address lines and strobe.

3.3.3 Ready Generation

The READY input to the TMS320C25 must go high to end the current cycle. As shown in Figure 3.7, READY is normally low, but $2\frac{1}{2}$ clock cycles after the USART is accessed, READY goes high. This timing delay is due to the wait state generator. The TMS320C25 then polls the READY input, detects a high, and ends the current cycle on the next positive edge of CLKOUT2. Hence, the USART line changes state $3\frac{1}{2}$ clock cycles after it began.

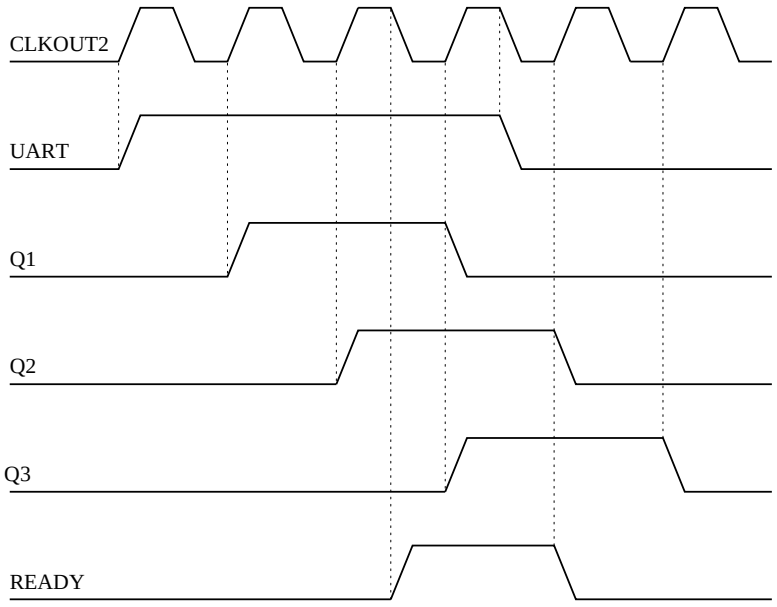


Figure 3.7: Ready Generation

Chapter 4

Software Development

The development of software for the TMS320C25 was aided by a collection of PC software tools, which provided a convenient platform for assembling, linking, and simulating programs. Initially there were various problems with these tools. Unfortunately, Texas Instruments have released several versions of these tools, and they appear to be incompatible. The initial development of the TMS320C25 assembly language programs was stunted due to mixing several versions of the tools together. It is important that a complete revision of tools and printed manuals be undertaken when upgrading to a more recent version.

4.1 EPROM Software Development

The initial software development cycle involved writing test programs in assembly language, linking to an object file, then converting this file format into a form suitable for down loading to an EPROM. This cycle is a time consuming one, since an EPROM must be erased for approximately 30 minutes before it can be reprogrammed. In order to avoid errors which could lengthen this development cycle, an assembly language simulator was used to verify that the source code was indeed correct before programming the EPROMs.

The simulator was found extremely useful, since it allowed single-stepping through instructions, and displayed the current contents of all registers, the accumulator, and the data memory contents. By tracing through a program, any errors could be quickly located, and corrected. Use of such a simulator is highly recommended, although it should not be taken as gospel.

There were several occasions when a program worked in the simulator, but would not work when implemented in hardware. Eventually it was traced to

a software error, which was not showing up in the simulator. The simulator initialised unused data bits to zero, so when for example, a read from the USART is performed (an 8 bit bus), the simulator would set the top 8 bits to zero. In actual fact, the hardware sets these bits to `FFh`, so when ORing the read data into the accumulator, the top 8 bits would get overwritten.

This type of occurrence should be taken into account if something appears to work when simulated, but not when actually implemented.

4.1.1 Creating EPROM files

The creation of files suitable for down loading to an EPROM requires a number of steps. The following commands assume the assembly language source code has been written in an ASCII text editor, and is called `EXAMPLE.ASM`

```
C:\>  dspa -lc EXAMPLE
```

This takes `EXAMPLE.ASM` and produces `EXAMPLE.OBJ`. The `-lc` parameters tell the assembler to produce a listing file, and to ignore case.

```
C:\>  dsplnk EXAMPLE.OBJ EXAMPLE.LNK -o EXAMPLE.OUT
              -m EXAMPLE.MAP
```

This links `EXAMPLE.OBJ` with `EXAMPLE.LNK` to produce `EXAMPLE.OUT`, and the memory map file, `EXAMPLE.MAP`. The linker command file defines the memory locations of the program, data and io-spaces, and depends on the physical memory map of the hardware system.

```
C:\>  dsprom -i EXAMPLE.OUT
```

This takes `EXAMPLE.OUT` and produces `EXAMPLE.HI` and `EXAMPLE.LO` files using the Intel hex file format.

```
C:\>  hexobj02
```

This has to be run twice, and takes `EXAMPLE.LO`, and produces `EXAMPLE.L` using Intel hex file format. Likewise for `EXAMPLE.HI` The newly created files `EXAMPLE.L` and `EXAMPLE.H` are then in a suitable format for programming into an EPROM.

4.2 PRAM Down Loader

Since the EPROM software development cycle is very tedious, a down loader program was written which allowed new programs to be sent via the serial port and run from program RAM. The obvious advantage of this technique is it's speed. There is no longer any delay time while programming and erasing EPROMs.

This technique involved the creation of two sets of software - the assembly language software programmed into the EPROM, and a suite of Turbo C programs running on the PC.

The software development cycle is similar to that of the EPROM cycle, in that the code is assembled, linked and converted to EPROM format, but then the file is modified ready for down loading.

4.2.1 Down Loader Protocol

The protocol used for sending the program to the DSP board was the same as that used by Geoff Liersch for his TMS320C50 board, and was used here for consistency amongst the University's DSP systems.

An EPROM-ready file consisting of 16 bit words is modified by adding the destination address of the program, and the program length to the beginning of the file.

The first 16 bit word received by the DSP board indicates the destination address in program RAM, high byte followed by low byte. The next 16 bit word specifies the length of code to be loaded.

This length N is defined as, $N = \frac{S}{2} - 1$. Where S is the number of bytes to be sent. Note that N should not include the first four bytes specifying the destination address or length of code to be transferred.

At the completion of the serial transfer, the TMS320C25 branches to the destination address and begins executing the program.

The loader program running in the EPROM, initialises the USART, then waits for a byte to be received. It takes the first two bytes and creates the destination address, and uses the following two bytes to determine the length of code. It then reads the code from the USART, and stores it at the specified address in program RAM, incrementing the address after each 16 bit word.

4.2.2 Echo Testing

The loader program also incorporates a form of error-detection, by echoing every received byte back to the PC. The down loader program on the PC then compares the transmitted byte with that received, and if they are the

same, it sends the next byte. Otherwise it terminates, informing the user that an echo-test error occurred.

4.2.3 PRAM Down Loader Software Development

The development cycle for generating code to be down-loaded direct to the DSP board is similar to that of the EPROM development cycle.
The steps to be followed are:

```
C:\>  dspa -lc EXAMPLE
```

This is the same as for an EPROM development cycle.

```
C:\>  dsplnk EXAMPLE.OBJ EXAMPLE.LNK -o EXAMPLE.OUT
             -m EXAMPLE.MAP
```

This varies from an EPROM development cycle, in that the linker command file now has a different memory map. For an EPROM program, the ROM lies in the lower half of the memory map, whereas for a RAM program, the upper half of the memory map is used.

```
C:\>  dsprom -w EXAMPLE.OUT
```

This command is also different. The `-w` parameter specifies Intel word format, since we want to download a single file, rather than create two EPROM files. This command produces `EXAMPLE.HEX`

```
C:\>  hexobj02
```

This is similar to the EPROM cycle, but it reads `EXAMPLE.HEX`, and produces `EXAMPLE.BIN` using Intel format. This `.BIN` file is the actual program code, but now it needs the address and length words added to it.

```
C:\>  bin2load
```

This program reads `EXAMPLE.BIN` and creates `EXAMPLE.LOD` which is ready to download via the serial interface. The address parameter must be the same as that specified in the linker command file.

```
C:\>  send 2 9600 EXAMPLE.LOD
```

This sends `EXAMPLE.LOD` to COM2, using a baud rate of 9600, and assumes the following communications parameters: Parity = None, Data bits = 8, Stop bits = 1.

4.3 Programming the USART

A number of problems were encountered in programming the USART. The 8251A provides two operation modes, asynchronous and synchronous. Asynchronous mode was chosen for this development system, since it allows an easy interface to an IBM PC. Hence, the initialisation software was written to place the USART into asynchronous mode immediately after the TMS320C25 was reset. This resulted in intermittent operation, so a closer look at the initialisation sequence was necessary.

The data sheet on the 8251A states that in order to ensure the USART is placed in a pre-determined state before attempting to initialise any registers, the mode must first be setup. This is accomplished by choosing synchronous mode, and sending two dummy sync characters, before powering down the USART into *idle* mode. A software reset can then be issued, followed by the command to place the USART into asynchronous mode, then programming the internal registers. This sequence of instructions resulted in a more robust reset sequence but it was still not completely reliable. It was discovered that the USART could not recover quickly enough after a write instruction during the initialisation sequence. The data sheet specified the write recovery time for asynchronous mode as being $8t_{CY}$. With the USART clock running at 2.4576MHz, this equates to $3.26\mu s$. In order to ensure a reliable initialisation, the TMS320C25 must wait for at least $3.26\mu s$ after every OUT instruction. The cycle time for the processor running at 40MHz is 100ns, hence the program must wait for 33 cycles before issuing the next USART instruction.

A delay of 33 cycles was accomplished by implementing the following loop :

```
                LALK 07h        ; Requires 2 clock cycles
LOOP            NOP             ; One clock cycle
                SUBK 01h        ; One clock cycle
                BNZ LOOP        ; Three clock cycles
```

Using a loop counter of `06h` gives a delay of 31 clock cycles, so in order to ensure reliability, `07h` was chosen, providing a delay of 37 clock cycles. After both these corrective changes were made, the USART was found to function reliably, and as expected, with no sign of erroneous operation.

Chapter 5

Testing and Verification

5.1 TMS320C25 and EPROM

In order to verify that the TMS320C25 was operating correctly, a small program was written and burnt into the EPROMs which simply toggled the XF pin. This pin was then monitored using an oscilloscope, and it was observed that the output was indeed switching state. This therefore verified that the processor was working, the EPROMs had been wired correctly, and that the one-wait state generator was correct.

5.2 IO Ports

Four LEDs were assigned as output ports to be used as test indicators for software development. Another program was written and programmed into the EPROMs to verify that these LEDs were functioning correctly. This program repeatedly flashed the LEDs in a cyclic sequence, thus verifying the PAL used for this task was correct. Since the LEDs were to operate with zero wait states, this test was also used to verify that the wait state circuitry could generate both zero and one wait states.

5.3 USART Clock

The USART clock was a simple crystal oscillator using a counter to divide the main crystal frequency to the required baud rate frequency. This was verified to function correctly, although the piano style DIP switch caused a few intermittent contact problems. Changing it to a more conventional slider DIP switch fixed this problem.

5.4 USART

Testing of the USART was accomplished using two separate programs. Initially, a simple program to repeatedly send `AAh` followed by `55h` was written, and burnt into the EPROMs. A second program, written in Turbo C was developed to run on the PC in order to receive the characters sent by the DSP system. A number of difficulties were encountered here, all of which have been documented previously. When these problems had been ironed out, the PC was able to reliably receive the correct characters. This proved that the USART could transmit satisfactorily.

In order to verify the receive capabilities of the USART, another program was written and burnt into the EPROMs which simply read a character from the keyboard, added one to it, and echoed it back to the screen. This was done to show that the character was actually being processed, and not simply being turned around somewhere. Since the program was now expecting to read a character, the Turbo C program was modified to act as a general purpose terminal program which could both send data entered from the keyboard and display data received from the serial port.

Having shown that the USART was functioning correctly, the PRAM down loader was then developed which allowed programs to be sent from the PC instead of going through the repetitive EPROM program/erase cycle for developing new software.

5.5 DTMF Encoder Testing

The encoding software was tested using a number of steps. Initially, only a single frequency sine wave was generated, and verified to be of the correct frequency. This was accomplished by measuring the output with an oscilloscope to verify that the shape of the wave was clean, and without distortion. The frequency of oscillation was measured using a digital frequency counter, and quite surprisingly, the generated wave was exactly the correct frequency. This procedure was repeated for all eight frequencies of the DTMF keypad, and all eight were proved to be correct, accurate to within 0.5Hz, or approximately 0.05% depending on frequency.

The next test added two sine waves together to produce a DTMF signal. This signal was examined on an oscilloscope, and the characteristic modulation effects obtained by mixing two signals together was observed. In order to verify that the signals were in fact standard DTMF tones, the signal was applied to a small speaker. This speaker was then coupled to the mouthpiece of an electronic telephone, and by pressing digits on the PC keyboard, it was

possible to dial remote telephone numbers, and receive a ringing tone back from the exchange. This test proved that the DTMF encoder was functioning correctly.

5.6 DTMF Decoder Testing

In order to test the operation of the DTMF decoder, it was essential that the DTMF encoder functioned properly. Without a calibrated source of DTMF tones, the development of the decoder would have been very difficult.

In order to verify the operation of the decoder, the encoder was used to generate a series of DTMF tones which were recorded using a standard magnetic cassette recorder. These tones could then be played back and coupled into the AIC interface on the DSP board. The AIC digitised these analog signals, and the decoder was able to correctly determine which tone had been recorded. The decoded tone was then sent to the PC, and displayed on a monitor.

A number of different DTMF recordings were made. These varied in tone length, amplitude, playback level and transmission speed. It was found that the decoder required a minimum tone length of approximately 50ms, as specified in [1], but the maximum length was not important. Likewise, a minimum amplitude was required in order to raise the received signal above the ambient noise level. This was determined experimentally to be approximately 1Vpp at the input to the AIC. This same situation also applied to the playback level. The transmission speed did not affect the rate at which the tones were generated, since the keyboard strokes were buffered by the Turbo C interface program.

Simply transmitting, recording, playing back, and decoding a signal was not sufficient to prove that the system was actually generating correct DTMF tones though. All this proved was that the decoder could decipher the tone generated by the local encoder. In order to fully verify the decoding software, a commercial DTMF generator was used for testing. This type of device is commonly used for remote-control applications over a telephone system. Using this commercial encoder, the decoder was able to correctly decipher all possible tones, thereby proving its operation.

Chapter 6

Conclusion and Future Development

A general purpose digital signal processing system has been presented here. The basic hardware consisting of a TMS320C25 processor, external ROM, RAM and PC interface will allow this system to be adapted to a large variety of applications.

A typical signal processing application involving the implementation of the Goertzel algorithm for DTMF detection has been included. This, along with a selection of test programs showed that the hardware did function as expected, and gave an indication of the suitability of this system for DSP applications.

A DTMF encoder was implemented, and tested by interfacing to the public telephone network. The encoder was successfully able to dial any given number, including long distance codes.

The DTMF decoder was verified to function correctly by reading an analog data signal from a magnetic tape, processing the data, and displaying the decoded tone on a PC monitor. An independent, hand-held DTMF generator was also used to verify the correct operation of the decoder.

As it stands at the moment, the DSP system could be used for a large variety of applications with virtually no modifications. The analog interface may need altering under certain circumstances, since this section was customised for DTMF encoding and detection. The digital hardware should not require any modifications.

Typical applications for this development system could include real-time spectral analysis, speech recognition, image processing, function generation and so on. The possibilities are virtually endless.

Bibliography

[1] Mock, P. *Add DTMF generation and decoding to DSP-uP designs*, Digital Signal Processing Applications with the TMS320 Family, Theory, Algorithms and Implementations, Vol. 1, 1989, Reprinted from Electronic Design News, October 1985.

[2] *General-Purpose Tone Decoding and DTMF Detection*, Digital Signal Processing Applications with the TMS320 Family, Theory, Algorithms and Implementations, Vol. 2, 1990, Texas Instruments

[3] *Precision Digital Sine-Wave Generation with the TMS32010*, Digital Signal Processing Application Report, 1984, Texas Instruments

[4] *Dual-Tone Multi-Frequency Coding*, ADSP-2100 Family Applications Handbook, Vol. 2, 1988, Analog Devices.

[5] *Second Generation TMS320 User's Guide*, 1987, Texas Instruments.

[6] *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide*, 1991, Texas Instruments.

[7] *TMS320C2x C Source Debugger User's Guide*, 1991, Texas Instruments.

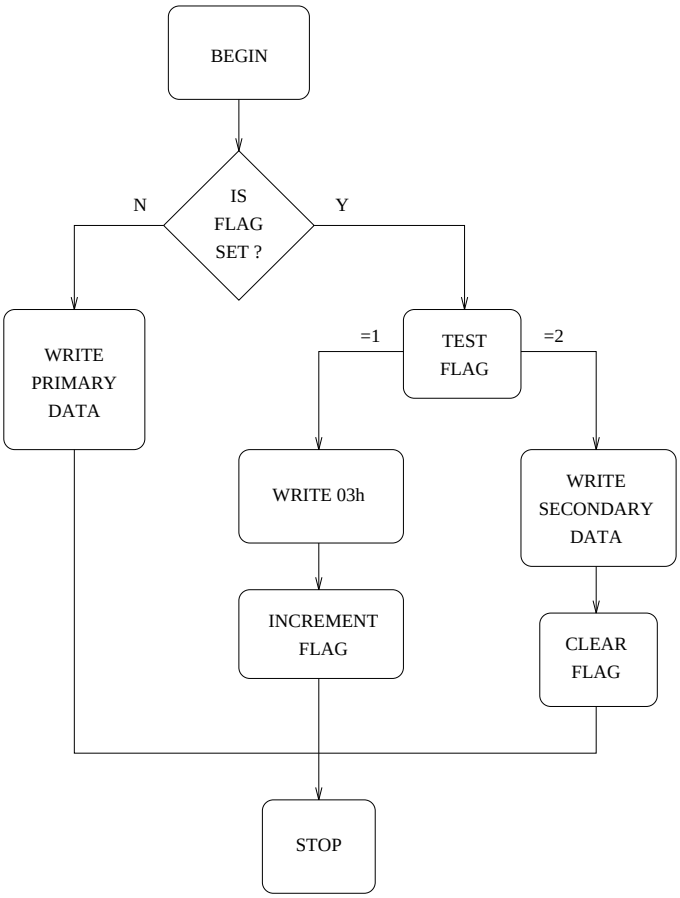
[8] *TMS320 Family Simulator User's Guide*, 1987, Texas Instruments.

[9] *TMS320C2x/C5x Optimizing C Compiler User's Guide*, 1991, Texas Instruments.

[10] *TLC32040 Interface to the TMS32020*, Digital Signal Processing Applications with the TMS320 Family, Theory, Algorithms and Implementations, Vol. 2, 1990, Texas Instruments

Appendix A

AIC Transmit Interrupt Service Routine



Appendix B

Linker command files

B.1 EPROM development

```
/*
FileName = EPROM.LNK
Interrupt vector table exists from 0 -> 001Fh
ROM exists from 0020h -> 7FFFh
DRAM exists from 0200h -> 3FFFh   ie Data Pages  4 -> 128
IO ports exists from 0 -> 6
*/

MEMORY
{
PAGE 0 : INTVEC : origin = 0x0000,length = 0x0020
EXE : origin = 0x0020,length = 0x6FDF
PAGE 1 : DRAM : origin = 0x0200,length = 0x3DFF
PAGE 2 : IO : origin = 0x0000,length = 0x0006
}

SECTIONS
{
TRAP : {} > INTVEC
.text : {} > EXE
.data : {} > EXE
.bss : {} > DRAM
}
```

B.2 PRAM down loader development

```
/*
FileName = DOWN.LNK
Duplicate vector table exists from 8000h -> 801Fh
PRAM exists from 8020h -> BFFFh
DRAM exists from 0200h -> 3FFFh   ie Data Pages  4 -> 128
IO ports exists from 0 -> 6
*/

MEMORY
{
PAGE 0: VECT : origin = 0x8000,length = 0x0020
EXE : origin = 0x8020,length = 0x3FDF
PAGE 1: DRAM  : origin = 0x0200,length = 0x3DFF
PAGE 2: IO : origin = 0x0000,length = 0x0006
}

SECTIONS
{
VECTORS : {} > VECT
.text : {} > EXE
.data : {} > EXE
.bss : {} > DRAM
}
```

Appendix C

Simulator command file

```
; FileName = SIMINIT.CMD
; This file defines the system memory map as used by the
; simulator.

ma 0,0,0x7000,ram ; bottom section of program ram
ma 0x7000,0,0x3000,ram ; remaining section of pram
ma 0,1,6,ram ; dram reserved registers
ma 0x60,1,0x20,ram ; dram on chip block B0
ma 0x0200,1,0x7000,ram ; dram - on chip and external

ma 0,2,1,oport ; LED0
mc 0,2,LED0,write
ma 1,2,1,oport ; LED1
mc 1,2,LED1,write
ma 2,2,1,oport ; LED2
mc 2,2,LED2,write
ma 3,2,1,oport ; LED3
mc 3,2,LED3,write

ma 4,2,1,ioport ; Uart data register
mc 4,2,u_datar,read
mc 4,2,u_dataw,write

ma 5,2,1,ioport ; Uart control/status register
mc 5,2,u_ctrlr,read
mc 5,2,u_ctrlw,write

ma 6,2,1,iport ; DTMF input to be decoded
```

```
mc 6,2,decin.dat,read

ma 7,2,1,oport ; Decoded DTMF output
mc 7,2,decout.dat,write
```

Appendix D

PAL Equations

D.1 Wait State Generation

```
;PALASM Design Description

;------------------------ Declaration Segment ------------
TITLE Memory decoding and wait state generator
PATTERN
REVISION 12
AUTHOR Steven J. Merrifield  VK3ESM
COMPANY La Trobe University
DATE 05 Aug 94

CHIP  _DECODE  PAL22V10

;------------------------ PIN Declarations ---------------
PIN  1 CLK ; INPUT
PIN  2 /PS ; INPUT
PIN  3 /DS ; INPUT
PIN  4 /IS ; INPUT
PIN  5 RW ; INPUT
PIN  6 /STRB ; INPUT
PIN  7 A2 ; INPUT
PIN  8 A15 ; INPUT
PIN  9 /MSC ; INPUT
PIN  14 /ONEWT ; OUTPUT
PIN  15 /UART ; OUTPUT
PIN  16 Q3 REGISTERED ; OUTPUT
```

```
PIN  17 Q2 REGISTERED ; OUTPUT
PIN  18 Q1 REGISTERED ; OUTPUT
PIN  19 /ROMREAD ; OUTPUT
PIN  20 /DRAMCS ; OUTPUT
PIN  21 /PRAMCS ; OUTPUT
PIN  22 /ROMCS ; OUTPUT
PIN  23 READY ; OUTPUT

;------------------------ Boolean Equation Segment ------


EQUATIONS

ROMCS = PS * STRB * /A15
PRAMCS = PS * STRB * A15
DRAMCS = DS * STRB * /A15
ROMREAD = RW * PS * STRB * /A15
ONEWT = PS * MSC * /A15
UART = IS * A2 * STRB
Q1 = UART * /Q2
Q2 = Q1
Q3 = Q2
READY = (ROMCS * /ONEWT) ; EPROM (1 ws)
+ (PS * A15) ; PRAM (0 ws)
+ (DS * /A15) ; DRAM (0 ws)
+ (IS * /A2) ; LEDS (0 ws)
+ ((Q3 * UART) + (Q2 * /CLK)) ; UART (3 ws)
```

D.2 IO Decoding

```
;PALASM Design Description

;------------------------ Declaration Segment ------------
TITLE IO map decoding
PATTERN
REVISION 1.0
AUTHOR Steven J. Merrifield  VK3ESM
COMPANY La Trobe University
DATE  07 JUN 94
```

```
CHIP  _IOMAP  PAL22V10

;------------------------ PIN Declarations ---------------
PIN  1 CLK ; INPUT
PIN  2 /STRB ; INPUT
PIN  3 /IS ; INPUT
PIN  4 RW ; INPUT
PIN  5 D0 ; INPUT
PIN  6 A0 ; INPUT
PIN  7 A1 ; INPUT
PIN  8 A2 ; INPUT
PIN  1 /UARTCS ; OUTPUT
PIN  1 /UARTR ; OUTPUT
PIN  18 /IOPORT3 REGISTERED ; OUTPUT
PIN  19 /IOPORT2 REGISTERED ; OUTPUT
PIN  20 /IOPORT1 REGISTERED ; OUTPUT
PIN  21 /IOPORT0 REGISTERED ; OUTPUT
PIN  22 /UARTW ; OUTPUT
PIN  23 /LE ; OUTPUT

;------------------------ Boolean Equation Segment ------
EQUATIONS

UARTCS = A2 * STRB * IS
UARTR  = A2 * STRB * IS * RW
UARTW  = A2 * STRB * IS * /RW
LE = /UARTW

IOPORT0 = ((/A2 * /A1 * /A0 * STRB * IS * /RW) * D0) +
(/(/A2 * /A1 * /A0 * STRB * IS * /RW) * IOPORT0)

IOPORT1 = ((/A2 * /A1 * A0 * STRB * IS * /RW) * D0) +
(/(/A2 * /A1 * A0 * STRB * IS * /RW) * IOPORT1)

IOPORT2 = ((/A2 * A1 * /A0 * STRB * IS * /RW) * D0) +
(/(/A2 * A1 * /A0 * STRB * IS * /RW) * IOPORT2)

IOPORT3 = (/A2 * A1 * A0 * STRB * IS * /RW * D0) +
(/(/A2 * A1 * A0 * STRB * IS * /RW) * IOPORT3)
```

Appendix E

Source Code

This appendix contains both the TMS320C25 assembly language source code, and the Turbo C code necessary for interfacing to an IBM PC serial port.

Appendix F

Schematic Diagrams

```
; ***************************************************************
; This program toggles the XF (external flag) pin high and low, and
; was the first ever written for the TMS320C25 DSP system. It verified
; that the processor, EPROM and wait state generator were working
; correctly. Note that it was assembled and linked using an old version
; of the tools, hence the different assembler directives.
; ***************************************************************
;
            TITL 'TOGGLE TEST'

            AORG >0000
RESET       B INIT

            AORG >0020
INIT        ldpk 0          ; set DP reg. to point to data page 0
LOOP        sxf             ; set external flag pin high
            rxf             ; reset external flag pin low
            b LOOP

            END
```

```
; ***************************************************************
; LED chaser program.
; Display pattern is as follows : (0 1 2 3 2 1) 0 1 2 3 2 1 0 1 2 ...
; We need to repeat the marked sequence with a delay after each flash
; so we can see the LED changing state. This program was initially
; burnt into ROM to verify the IO port addressing PAL.
; ***************************************************************
;
            .text

LED0        .set 0                  ; address of LED 0
LED1        .set 1                  ; address of LED 1
LED2        .set 2                  ; address of LED 2
LED3        .set 3                  ; address of LED 3
LP_DEL      .set 0FFh               ; loop delay
ON          .set 1                  ; offset from page pointer
OFF         .set 0                  ; offset from page pointer

            ldpk 4                  ; store data on page 4 (0200h)
            lalk    1               ; data to turn LED on
            sacl    ON              ; store at 0201h - offset 1
            zac                     ; data to turn LED off (ACC <- 0)
            sacl    OFF             ; store at 0200h - offset 0

TOP         out OFF,LED0
            call DELAY
            out ON,LED0
            call DELAY

            out OFF,LED1
            call DELAY
            out ON,LED1
            call DELAY

            out OFF,LED2
            call DELAY
            out ON,LED2
            call DELAY

            out OFF,LED3
            call DELAY
            out ON,LED3
            call DELAY

            out OFF,LED2
            call DELAY
            out ON,LED2
            call DELAY

            out OFF,LED1
            call DELAY
            out ON,LED1
            call DELAY

            b TOP                   ; repeat the entire sequence again

DELAY       lalk LP_DEL,8           ; left shift to make delay longer
LOOP        subk 1                  ; decrement counter
            bnz LOOP                ; until counter is zero
            ret                     ; then return to caller

            .end
```

```
; ***************************************************************
; Basic loader program - Reads .LOD files from the serial port into
; data ram then copies from data ram into program ram. When the whole
; file has been copied into program ram, it branches to the start address
; and starts running the downloaded program. It also echos any received
; data back to the PC for error checking.

; 08 Aug 94 - Initial release
; 23 Aug 94 - Removed delays after every uart instruction that was not
;             part of the init. sequence (now loads more quickly)
; 02 Sep 94 - Added interrupt vector table

; LED 0 turns on after initialising the UART.
; LED 1 turns on after reading the start address.
; LED 2 turns on after reading the length of code to be sent.
; LED 3 turns on after loading the program.
; ***************************************************************
;
vect        .set 8000h          ; start of interrupt vector table in DRAM

            .sect "VECTORS"
            b INIT              ; external reset
            b vect+2            ; int 0
            b vect+4            ; int 1
            b vect+6            ; int 2
            b vect+8            ; reserved
            b vect+10           ; reserved
            b vect+12           ; reserved
            b vect+14           ; reserved
            b vect+16           ; reserved
            b vect+18           ; reserved
            b vect+20           ; reserved
            b vect+22           ; reserved
            b vect+24           ; internal timer
            b vect+26           ; serial port rx
            b vect+28           ; serial port tx
            b vect+30           ; trap instruction address

            .text

temp0       .set 0
temp1       .set 1
temp2       .set 2
boot_addr   .set 3      ; destination addr. of boot code
byte_cnt    .set 4      ; length of code to be sent
save_acc    .set 5      ; temp for intermediate acc. access
ON          .set 6      ; data to turn LED on
OFF         .set 7      ; data to turn LED off

; IO ports
LED0        .set 0
LED1        .set 1
LED2        .set 2
LED3        .set 3
u_data      .set 4      ; UART data register
u_ctrl      .set 5      ; UART control/status register

; ***************************************************************
;           Execution starts here
; ***************************************************************
INIT        dint     ; disable interrupts
            rovm     ; disable overflow
            ssxm     ; allow extended signed no's.
            cnfd     ; configure block B0 as data memory
            ldpk 4   ; start data memory at 0200h
```



```
; Setup LED data
            zac
            sacl OFF
            lalk 01
            sacl ON

; Reset all LEDs
            out OFF,LED0
            out OFF,LED1
            out OFF,LED2
            out OFF,LED3

; ***************************************************************
;           Assume worst-case UART initialisation
; ***************************************************************
            zac
            sacl temp0
            out temp0,5     ; set sync mode operation
            call U_DELAY

            out temp0,5     ; load 1st dummy sync char
            call U_DELAY

            out temp0,5     ; load 2nd dummy sync char
            call U_DELAY

            lack 40h        ; internal reset
            sacl temp0
            out temp0,5
            call U_DELAY

; UART is now idling and waiting for configuration data

            lack 04Eh   ; N,8,1 x16
            sacl temp0
            out temp0,5
            call U_DELAY

            lack 05     ; enable Tx & Rx
            sacl temp0
            out temp0,5
            call U_DELAY

; Light LED0 after UART initialisation
            out ON,LED0

; ***************************************************************
; Get destination addr. of boot code & store it in dma(boot_addr)
; ***************************************************************
;
label1      in temp0,u_ctrl   ;
            bit temp0,14      ;  wait until we rx a char
            bbz label1        ;

            in temp0,u_data     ; read high byte of dest. addr.
            call SENDBACK
            lac temp0,8         ; shl 8
            sacl save_acc

label2      in temp0,u_ctrl  ;
            bit temp0,14     ;  wait for a char
            bbz label2       ;

            in temp0,u_data     ; read low byte of dest. addr.
            call SENDBACK
            lac temp0
            andk 0FFh           ; mask out top 8 bits
```

```
            sacl temp0
            lac save_acc
            or temp0
            sacl boot_addr

; Light LED1 after setting up boot adddress
            out ON,LED1

; ****************************************************************
;           Get length of code & store it in dma(byte_cnt)
; ****************************************************************
label3      in temp0,u_ctrl ;
            bit temp0,14    ; wait for a char
            bbz label3      ;

            in temp0,u_data     ; high byte of count
            call SENDBACK
            lac temp0,8
            sacl save_acc

label4      in temp0,u_ctrl
            bit temp0,14        ; wait for a char
            bbz label4

            in temp0,u_data     ; low byte of count
            call SENDBACK
            lac temp0
            andk 0FFh ; mask out top 8 bits
            sacl temp0
            lac save_acc
            or temp0
            addk 01h ; add 1 so byte_cnt agrees with Lurch's protocol
            sacl byte_cnt

; Light LED2 after setting up byte count
            out ON,LED2

; ****************************************************************
; Get code and store it in a temp memory location in data ram then transfer
; from that temp location to program ram and decrement byte_cnt. Check if
; byte_cnt = 0, if not then get next piece of code.
; ****************************************************************
            lac boot_addr   ; store boot_addr in temp mem loc so it can
            sacl temp1      ; be incremented for tblw

loop1       in temp0,u_ctrl
            bit temp0,14        ; wait for a char
            bbz loop1

            in temp0,u_data     ; high byte of data
            call SENDBACK
            lac temp0,8
            sacl save_acc

label5      in temp0,u_ctrl
            bit temp0,14        ; wait for a char
            bbz label5

            in temp0,u_data     ; low byte of data
            call SENDBACK
            lac temp0
            andk 0FFh           ; mask out top 8 bits
            sacl temp0
            lac save_acc
            or temp0
            sacl temp0  ; write data to temp mem. addr. for tlbw
```



```
            lac temp1       ; temp1 contains addr to write to in pm
            tblw temp0      ; transfer from dma(temp0) to pma(ACC)
            addk 1          ; increment ACC for next access by tblw
            sacl temp1      ; save new index for pma
            lac byte_cnt
            subk 1
            sacl byte_cnt
            bnz loop1

; Light LED3 after loading code into pm
            out ON,LED3

; jump to dest. addr and start running program
            lac boot_addr
            bacc

; ****************************************************************
; Echo the received byte back to the PC for error checking. The PC end
; compares the sent byte with the echoed byte, and if they are not the
; same it terminates with an "echo test error".
; ****************************************************************
SENDBACK    sacl save_acc
CHECK       in temp2,u_ctrl     ; wait until TxRDY
            bit temp2,15
            bbz CHECK

            out temp0,u_data    ; echo data back to PC
            lac save_acc
            ret

; ****************************************************************
; We need a delay of at least 33 CPU clock cycles (at 40MHz) after each
; UART access during initialisation to allow for the recovery time.
; ****************************************************************
U_DELAY     sacl save_acc       ; PUSH accumulator
            lalk 7
wait        nop                 ; 1 clock cycle
            subk 1              ; 1 clock cycle
            bnz wait            ; 3 clock cycles
            lac save_acc        ; POP accumulator
            ret

            .end
```

```
; ****************************************************************
; This program reads a character from the PC serial interface, and
; uses a lookup table to decide which tone to generate. It then
; synthesises a DTMF tone for a fixed period of time, then zeros the
; ouput of the DAC.

; Note that with a 40MHz CPU clock, the actual sampling frequency could
; not be set to exactly 8kHz. It was defined to be 7936.5Hz, and the
; values in the key_table reflect this alteration.

; Note that flags are used to determine when to process new data.
; If the flag is set to 00FFh then an interrupt has occured, and the
; program branches to the relevant service routine.

; When a transmit interrupt occurs the program branches to the transmit
; interrupt service routine and sets the tx_flag. It then gets data
; from data ram and writes it to the tx serial port register.
; ****************************************************************
;
            .sect "VECTORS"
            b start         ; 0 - External reset
            b INT0          ; 2 - User int 0
            b INT1          ; 4 - User int 1
            b INT2          ; 6 - User int 2
            b d_int         ; 8 - Reserved
            b d_int         ; 10 - Reserved
            b d_int         ; 12 - Reserved
            b d_int         ; 14 - Reserved
            b d_int         ; 16 - Reserved
            b d_int         ; 18 - Reserved
            b d_int         ; 20 - Reserved
            b d_int         ; 22 - Reserved
            b tim_int       ; 24 - Internal timer
            b rx_int        ; 26 - Serial port rx
            b tx_int        ; 28 - Serial port tx
            b d_int         ; 30 - Trap instruction addr.

            .text

sine        .word   0000h, 0324h, 0646h, 0964h, 0c7ch, 0f8dh, 1294h
            .word   1590h, 187eh, 1b5dh, 1e2bh, 20e7h, 238eh, 2620h
            .word   289ah, 2afbh, 2d41h, 2f6ch, 3179h, 3368h, 3537h
            .word   36e5h, 3871h, 39dbh, 3b21h, 3c42h, 3d3fh, 3e15h
            .word   3ec5h, 3f4fh, 3fb1h, 3fech, 4000h, 3fech, 3fb1h
            .word   3f4fh, 3ec5h, 3e15h, 3d3fh, 3c42h, 3b21h, 39dbh
            .word   3871h, 36e5h, 3537h, 3368h, 3179h, 2f6ch, 2d41h
            .word   2afbh, 289ah, 2620h, 238eh, 20e7h, 1e2bh, 1b5dh
            .word   187eh, 1590h, 1294h, 0f8dh, 0c7ch, 0964h, 0646h
            .word   0324h, 0000h
            .word   0fcdch, 0f9bah, 0f9bah, 0f69ch, 0f384h, 0f073h
            .word   0ed6ch, 0ea70h, 0e782h, 0e4a3h, 0e1d5h, 0df19h
            .word   0dc72h, 0d9e0h, 0d766h, 0d505h, 0d2bfh, 0d094h
            .word   0ce87h, 0cc98h, 0cac9h, 0c91bh, 0c78fh, 0c625h
            .word   0c4dfh, 0c3beh, 0c2c1h, 0c1ebh, 0c13bh, 0c0b1h
            .word   0c04fh, 0c014h, 0c000h, 0c014h, 0c04fh, 0c0b1h
            .word   0c13bh, 0c1ebh, 0c2c1h, 0c3beh, 0c4dfh, 0c625h
            .word   0c78fh, 0c91bh, 0cac9h, 0cc98h, 0ce87h, 0d094h
            .word   0d2bfh, 0d505h, 0d766h, 0d9e0h, 0dc72h, 0df19h
            .word   0e1d5h, 0e4a3h, 0e782h, 0ea70h, 0ed6ch, 0f073h
            .word   0f384h, 0f69ch, 0f9bah, 0fcdch

m1          .word 07fffh

key_table   .word 0f2dh, 158ch    ; 0
            .word 0b3dh, 137fh    ; 1
            .word 0b3dh, 158ch    ; 2
```



```
            .word 0b3dh, 17d2h    ; 3
            .word 0c6bh, 137fh    ; 4
            .word 0c6bh, 158ch    ; 5
            .word 0c6bh, 17d2h    ; 6
            .word 0dbdh, 137fh    ; 7
            .word 0dbdh, 158ch    ; 8
            .word 0dbdh, 17d2h    ; 9
            .word 0b3dh, 1a56h    ; A
            .word 0c6bh, 1a56h    ; B
            .word 0dbdh, 1a56h    ; C
            .word 0f2dh, 1a56h    ; D
            .word 0f2dh, 137fh    ; *
            .word 0f2dh, 17d2h    ; #

; IO ports
u_data      .set 4
u_ctrl      .set 5


; Page 0 variables (0000h)
dxr         .set 1              ; data tx reg. address
imr         .set 4              ; interrupt mask reg.
tx_flag     .set 96             ; data has been sent flag
stat_st     .set 97             ; temp for saving status reg
accl_st     .set 98             ; temp for low half of accumulator
acch_st     .set 99             ; temp for high half of accumulator
tx_data     .set 100            ; data to be tx'ed must be stored here
init_data   .set 101            ; data for init. must be stored here
init_flag   .set 102            ; flag for secondary communications

delta1      .set 107            ; increment for first sine wave
alpha1      .set 108
sin1        .set 109            ; actual sine wave value
temp        .set 110
mask        .set 111
sin_offset  .set 112            ; pointer into sine lookup table
key_offset  .set 113            ; pointer into keypad lookup table
temp2       .set 114
temp3       .set 115
tone_len    .set 116            ; address for time one tone is sent
delta2      .set 117            ; increment for second sine wave
alpha2      .set 118
sin2        .set 119            ; actual sine wave value
last        .set 120
sec_last    .set 121

; ****************************************************************
;           Initialization
; ****************************************************************
start       ldpk 0              ; point to data page zero
            fort 0              ; set serial port to be 16 bits wide
            rtxm                ; external sync
            sfsm                ; sync required for each transfer
            cnfd                ; configure block B0 as data

            zac                 ; reset tx and init flags
            sacl tx_flag
            sacl init_flag

            larp ar1            ; counter for time one tone is sent

            eint
            lalk 020h           ; enable only tx interrupt
            sacl imr

; ****************************************************************
```

```
;           Main program
; ****************************************************************
;
            lalk 1223h          ; setup TA and RA (divide by 9)
            sacl init_data
            call tx_ready
            lalk 1
            sacl init_flag
wait_2nd_a  lac init_flag       ; wait until secondary comms is finished
            bnz wait_2nd_a

            lalk 468eh          ; setup TB and RB (divide by 35)
            sacl init_data
            call tx_ready
            lalk 1
            sacl init_flag
wait_2nd_b  lac init_flag       ; wait until secondary comms is finished
            bnz wait_2nd_b

            lalk 1000            ; time one time is transmitted (50ms)
            sacl tone_len

            lalk m1
            tblr mask
            lalk sine           ; save start addr. of sinewave lookup table
            sacl sin_offset

            zac
            sacl alpha1         ; start at zero in sine LUT
            sacl alpha2
            sacl delta1
            sacl delta2

            lalk key_table      ; prepare keypad lookup table
            sacl key_offset

loop        call check_key      ; see if a key has been pressed
            lac alpha1,8
            sach temp
            lac temp
            add sin_offset
            tblr sin1           ; calculate first sine wave sample
            lac alpha1
            add delta1
            and mask
            sacl alpha1

; 2nd sine wave
            lac alpha2,8
            sach temp
            lac temp
            add sin_offset
            tblr sin2           ; calculate second sine wave sample
            lac alpha2
            add delta2
            and mask
            sacl alpha2

            lac sin1            ; add the first and second together
            add sin2

            sacl tx_data        ; write the combined sum to DAC
            call tx_ready



            banz loop           ; banz has got a built in decrement
            zac                 ; when loop has run down, zero DAC output
```



```
            sacl alpha1
            sacl alpha2
            sacl delta1
            sacl delta2
            b loop

; ****************************************************************
;           Check if a key was pressed
; ****************************************************************
check_key   in temp2,u_ctrl      ; test if key pressed
            bit temp2,14
            bbz end_check

            in temp2,u_data     ; read data from uart into temp addr.
            lac temp2           ; remove top half since IN only reads
            andk 00FFh          ; an 8 bit number, so the top 8 bits
            sacl temp2          ; will be garbage!

            lac temp2,1         ; mult kbhit by 2 since LUT uses row-col
            add key_offset      ; move to correct position in LUT
            tblr delta1         ; read row value from LUT
            addk 1
            tblr delta2         ; read column value from LUT
            lar ar1, tone_len   ; reset tone length after every keypress

end_check   ret

; ****************************************************************
;           Test if transmit flag is set
; ****************************************************************
tx_ready    lac tx_flag         ; when flag is set, tx_flag = 0ffh
            andk 00ffh
            subk 0ffh
            bnz tx_ready        ; wait until flag is set
            sacl tx_flag        ; if flag is set, then reset it
            ret

; ****************************************************************
;           Transmit interrupt service routine
; ****************************************************************
tx_int      sst stat_st         ; push status reg
            sacl accl_st        ; push accumulator
            sach acch_st

            bit init_flag,14    ; are we sending init. data?
            bbz test_2nd
send_2nd    lac init_data       ; send actual init. data (ie divide no's)
            sacl dxr
            zac
            sacl init_flag      ; reset init flag
            b exit_tx_int

test_2nd    bit init_flag,15
            bbnz send_1st

primary     lalk 0ffh           ; set transmit data flag
            sacl tx_flag
            lac tx_data         ; get data from memory
            andk 0fffch         ; mask out bottom 2 bits
            sacl dxr            ; write data to AIC
            b exit_tx_int

send_1st    lac init_flag       ; increment init_flag
            addk 1
            sacl init_flag
            lalk 03             ; start secondary communications
```

```
            sacl dxr

exit_tx_int zals accl_st        ; pop accumulator
            addh acch_st
            lst stat_st         ; pop status reg

            eint
            ret

; ***************************************************************
;           Interrupts we're not interested in
; ***************************************************************
d_int       ret
rx_int      ret
tim_int     ret
INT0        ret
INT1        ret
INT2        ret

            .end
```

```
; ****************************************************************
; DTMF decoder implemented using the Goertzel algorithm.
; Electronics IV (Honours) Project 1994
; by Steven J. Merrifield
;
; This program incorporates changes made after the original thesis
; was submitted. Where there are discrepancies between this, and the
; original code, the code presented here should take precedence.
; ****************************************************************
;

            .sect "VECTORS"
            b start          ; 0 - External reset
            b INT0           ; 2 - User int 0
            b INT1           ; 4 - User int 1
            b INT2           ; 6 - User int 2
            b d_int          ; 8 - Reserved
            b d_int          ; 10 - Reserved
            b d_int          ; 12 - Reserved
            b d_int          ; 14 - Reserved
            b d_int          ; 16 - Reserved
            b d_int          ; 18 - Reserved
            b d_int          ; 20 - Reserved
            b d_int          ; 22 - Reserved
            b tim_int        ; 24 - Internal timer
            b rx_int         ; 26 - Serial port rx
            b tx_int         ; 28 - Serial port tx
            b d_int          ; 30 - Trap instruction addr.

            .text

; IO ports
LED0        .set 0
LED1        .set 1
LED2        .set 2
LED3        .set 3
u_data      .set 4
u_ctrl      .set 5

; Page 0 variables (0000h)
drr         .set 0      ; data rx reg. address
dxr         .set 1      ; data tx reg. address
imr         .set 4      ; interrupt mask reg.
tx_flag     .set 96     ; data has been sent flag
rx_flag     .set 97     ; data has been received flag
stat_st     .set 98     ; temp for saving status reg. during subroutine calls
accl_st     .set 99     ; temp for low half of accumulator
acch_st     .set 100    ; temp for high half of accumulator
rx_data     .set 101    ; received data is stored here
tx_data     .set 102    ; data to be transmitted must be stored here
init_data   .set 105    ; data for initialisation muse be stored here
init_flag   .set 106    ; flag for secondary communications (initialisation)
OFF         .set 107    ; data to turn LED off
ON          .set 108    ; data to turn LED on
stat_1      .set 109    ; save status register 1

nfilt       .set 8          ; No. of filters (one for each row/col freq)
u_data      .set 4
u_ctrl      .set 5

dram        .set 0200h      ; DRAM starts at addr. 0200h  (ie DP = 4)
cs1         .set dram+00
cs2         .set dram+01
cs3         .set dram+02
cs4         .set dram+03
cs5         .set dram+04
cs6         .set dram+05
```



```
cs7         .set dram+06
cs8         .set dram+07

negmax      .set dram+08

rowmx       .set dram+11
colmx       .set dram+12
rowmax      .set dram+13
colmax      .set dram+14
count       .set dram+15
rowcol      .set dram+16
last        .set dram+19
sec_last    .set dram+20

dat11       .set dram+28
dat23       .set dram+33
dat14       .set dram+34
dat15       .set dram+36
dat17       .set dram+40
dat27       .set dram+41
dat18       .set dram+42
dat28       .set dram+43
dat29       .set dram+45
dat213      .set dram+53
dat216      .set dram+59
datin       .set dram+60
temp        .set dram+61
temp2       .set dram+62
temp3       .set dram+63
save_acc    .set dram+64
prnt        .set dram+65
test        .set dram+66

; Filter co-efficients for each row/col. frequency
; Fundamental - Real coeff N=205

tblstrt     .word 27906         ; 697 Hz
            .word 26802         ; 770 Hz
            .word 25597         ; 851 Hz
            .word 24295         ; 941 Hz
            .word 19057         ; 1209 Hz
            .word 15654         ; 1336 Hz
            .word 12945         ; 1477 Hz
            .word 09166         ; 1633 Hz
tblend      .word 08000h        ; NegMax - mask for data out

; ****************************************************************
;           Initialization
; ****************************************************************
start       ldpk 0          ; Point to data page zero
            fort 0          ; Set serial port to be 16 bits wide
            rtxm            ; External sync
            sfsm            ; Sync required for each transfer

            cnfd            ; Set block B0 to be data memory
            sovm            ; set overflow mode
            ssxm            ; set sign extention mode

            zac             ; Reset tx and rx flags
            sacl tx_flag
            sacl rx_flag
            sacl init_flag
            sacl OFF
            lalk 1
            sacl ON
```

```
            eint
            lalk 020h       ; Enable only tx interrupt to init AIC
            sacl imr

            lalk 1223h      ; setup TA and RA
            sacl init_data
            call tx_ready
            lalk 1
            sacl init_flag
wait_2nd_a  lac init_flag       ; wait until secondary comms is finished
            bnz wait_2nd_a

            lalk 468eh          ; setup TB and RB
            sacl init_data
            call tx_ready
            lalk 1
            sacl init_flag
wait_2nd_b  lac init_flag       ; wait until secondary comms is finished
            bnz wait_2nd_b

            lalk 010h           ; enable only rx int for receiveing data
            sacl imr

; *****************************************
;       Execution starts here
; *****************************************
            ldpk 4              ; Point to data page 4 (0200h)

            larp    ar0
            lrlk    ar0,cs1             ; Pointer to the start of ram to be
                                        ; initialised.
            lalk    tblstrt             ; Pointer to the start of init table.
            lrlk    ar1,tblend-tblstrt  ; Count of data to be moved.

next        tblr    *+,ar1
            addk    1
            banz    next,*-,ar0

            lalk 0ffh                   ; set last and second last decoded
            sacl last                   ; digits to be "invalid"
            sacl sec_last

again       zac                         ; Zero DFT loop variables
            lrlk 0,15
            lrlk 1,dat11
zero        larp 1
            sacl *+,0,0
            banz zero

; *****************************************
;       Take data and calculate DFT loop
; *****************************************
            lalk 205                    ; Set DFT loop variable
loop        sacl count
            lrlk ar0,cs8                ; Set up pointer to co-efficients.
            lrlk ar1,dat28              ; Set up pointer to delayed outputs.
            lrlk ar2,nfilt-1            ; Number of filters.

; read from AIC under interrupts
            ldpk 0
            call rx_ready
            lac rx_data
            ldpk 4

            sfr         ; Stop accumulator from overflowing by shifting
            sfr         ; data to the right - this effectively takes the
```



```
;           sfr         ; 16 bit value read from the serial port and converts
;           sfr         ; it to a 14 bit value as generated by the AIC.
                        ; My hardware has an amplifier which limits the gain
                        ; of the input, so only two shifts are required. When
                        ; run in the simulator, all four shifts are needed.

            sacl datin

; *****************************************
;       Begin DFT loops
; *****************************************
frpt        larp ar0
            lt *,ar1            ; load cos(8*C) ready for multiply
            lac datin,12        ; X(n)
            subh *-             ; X(n) - Y(n-2)
            mpy *               ; cos(8*C) * Y(n-1)
            ltd *               ; Y(n-1) -> Y(n-2)
            apac
            apac
            apac                ; X(n) + 2cos(8*C) * Y(n-1) - Y(n-2)
            sach *-,0,ar0       ;    --> Y(n-1)
            bv overflow
            mar *-,ar2          ; Decrement the co-efficient pointer.
            banz frpt,*-,ar0    ; Decrement the filter number.

            lac count           ; Repeat for length of transform
            subk 1
            bnz loop
            b check

overflow    out OFF,LED0        ; Show overflow status on LED0

; *****************************************
;       Calculate energy at each frequency
; *****************************************
check       lrlk 0,cs8
            lrlk 1,dat28
            lrlk 2,nfilt-1

maglp       call energy
            sach *-,1,ar0
            mar  *-,ar2
            banz maglp,*-,ar0

; *****************************************
;       Compare energies and determine decode value
; *****************************************
            lalk 3
            sacl rowmx
            sacl colmx

; *****************************************
;       Find row peak
; *****************************************
rows        lrlk 1,2
            lrlk 0,dat23
            lac dat14
            sacl rowmax
rowl        larp 0
            mar *-
            lac rowmax
            sub *
            bgez rowbr

            sar 1,rowmx
            lac *
```

```
            sacl rowmax
rowbr       mar *-,1
            banz rowl

; *********************************************
;       Find column peak
; *********************************************
column      lrlk 1,2
            lrlk 0,dat27
            lac dat18
            sacl colmax
coll        larp 0
            mar *-
            lac colmax
            sub *
            bgez colbr

            sar 1,colmx
            lac *
            sacl colmax
colbr       mar *-,1
            banz coll

; *********************************************
;       Merge row / column together
; *********************************************
            lac rowmx,4
            or  colmx
            sacl rowcol

; *********************************************
;       Check for valid signal strength
; *********************************************
sig_str     lac colmax
            subk 4
            blz invalid_dig

            lac rowmax
            subk 4
            blz invalid_dig
            b test_new

invalid_dig lalk 0ffh
            sacl rowcol

; *********************************************
;       Test if the decoded digit is new
; *********************************************
test_new    lac rowcol
            sub last
            bz samelast

            lac last
            sacl sec_last
            lac rowcol
            sacl last
            b again

samelast    lac rowcol
            sub sec_last
            bz again

            lac last
            sacl sec_last
            lac rowcol
            sacl last
```



```
; check if decoded digit == 0 (my TC prog can't read an ASCII null)
            lac rowcol
            subk 0
            bnz check_u
            addk 55h
            sacl rowcol

check_u     in temp,u_ctrl          ; wait until TxRdy
            bit temp,15
            bbz check_u
            out rowcol,u_data       ; send decoded digit to PC

            b again                 ; get next sample

; *********************************************
; Energy calculation subroutine
; *********************************************
energy      lac negmax,15       ; NegMax = 8000h
            add *,15,1
            sach count
            lt *-               ; -1/2 + CSn/2
            mpy count
            pac
            sach count,1        ; D2(CSn-1)/2
            lt *+
            mpy count
            pac
            sach count,1        ; D1 * D2(CSn-1)/2
            lac *-,15
            sub *,15
            abs
            sach *              ; abs(D2-D1)/2
            lt *
            mpy *
            pac                 ; ((D2-D1)/2)^2
            sub count,15        ; ((D2-D1)^2)/4 - D1*D2(CSn-1)/2
            ret

; *****************************************************************
;           Test if receive flag is set
; *****************************************************************
rx_ready    ldpk 0
            lac rx_flag
            andk 00ffh
            subk 0ffh
            bnz rx_ready
            sacl rx_flag
            ret

; *****************************************************************
;           Test if transmit flag is set
; *****************************************************************
tx_ready    ldpk 0
            lac tx_flag
            andk 00ffh
            subk 0ffh
            bnz tx_ready
            sacl tx_flag
            ret

; *****************************************************************
;           Receive interrupt service routine
; *****************************************************************
rx_int      sst stat_st         ; Push status register
            ldpk 0
```

```
            sst1 stat_1
            sacl accl_st        ; Push accumulator
            sach acch_st
            lalk 0ffh           ; Set received data flag
            sacl rx_flag
            lac drr             ; Get data from AIC
            sacl rx_data        ; Save it to memory
            zals accl_st        ; Pop accumulator
            addh acch_st
            lst1 stat_1
            lst stat_st         ; Pop status register
            eint
            ret

; ****************************************************************
;           Transmit interrupt service routine
; ****************************************************************
tx_int      sst stat_st     ; Push status reg
            ldpk 0
            sst1 stat_1
            sacl accl_st    ; Push accumulator
            sach acch_st

            bit init_flag,14
            bbz test_2nd
send_2nd    lac init_data
            sacl dxr
            zac
            sacl init_flag
            b exit_tx_int

test_2nd    bit init_flag,15
            bbnz send_1st

primary     lalk 0ffh       ; Set transmit data flag
            sacl tx_flag
            lac tx_data     ; Get data from memory
            andk 0fffch     ; Mask out bottom 2 bits
            sacl dxr        ; Write data to AIC
            b exit_tx_int

send_1st    rxf
            lac init_flag
            addk 1
            sacl init_flag
            lalk 03
            sacl dxr

exit_tx_int zals accl_st    ; Pop accumulator
            addh acch_st
            lst1 stat_1
            lst stat_st     ; Pop status reg

            eint
            ret

; ****************************************************************
;           Interrupts we're not interested in
; ****************************************************************
d_int       ret
tim_int     ret
int0        ret
int1        ret
int2        ret

            .end
```

```
#include <stdio.h>

main()
 {
  char infile[80];
  char outfile[80];
  FILE *in;
  FILE *out;
  unsigned int i;
  unsigned char tmp;
  unsigned int count=0;
  unsigned int bytecnt;
  unsigned char data;
  unsigned int address;

  printf("Welcome to bin2load Version 3.1415\n\n\n");
  printf("Enter bin file name:");
  scanf("%s",infile);
  in = fopen(infile,"rb");
  if (!in)
   {
    printf("Error: cannot find file %s\n",infile);
    return(-1);
   }
  printf("Enter load file name:");
  scanf("%s",outfile);
  out = fopen(outfile,"wb");
  if (!out)
   {
    printf("Error: cannot open file %s for writing\n",outfile);
    return(-1);
   }
  printf("Enter start address (in hex):");
  scanf("%x",&address);
  while (!feof(in))
   {
    fread(&data,1,1,in);
    count++;
   }
  count--; /* Remove one for the EOF character */
  bytecnt = count;
  count = (count / 2) - 1;
  printf("Address = 0x%04x %5u\n",address,address);
  printf("Byte Count = 0x%04x %5u\n",bytecnt,bytecnt);
  printf("Count = 0x%04x %5u\n",count,count);
  fseek(in,0,0);
  tmp = (unsigned char)(address >> 8);
  fwrite(&tmp,1,1,out);
  tmp = (unsigned char)(address & 0xff);
  fwrite(&tmp,1,1,out);
  tmp = (unsigned char)(count >> 8);
  fwrite(&tmp,1,1,out);
  tmp = (unsigned char)(count & 0xff);
  fwrite(&tmp,1,1,out);
  for (i=0;i<bytecnt;i++)
   {
    fread(&data,1,1,in);
    fwrite(&data,1,1,out);
   }
  fclose(in);
  fclose(out);
  return(0);
 }
```

```
/*************************************************************************

          DSP RAM downloader by Steven J. Merrifield

      Based on the serial routines originally written by Peter Ibbotoson
      of Borland Intl. c1987 and downloaded as SERIAL.ARJ from
          "The Software Parlour BBS +(613) 338 3794."

          Revision history :

      1.0   940807 - First release.
      1.1   940823 - Reduced delay for overrun errors & altered printf
                     statement to speed up transfer. Changed the way the
                                    header was formatted.
          1.2   940903 - Commented out echo testing for Darin's 6809 board
          1.3   940922 - Cleaned up opening screen - added <ESC> to abort
                                    during download

*************************************************************************/

#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "serial.h"      /* communication routines */


#define FALSE 0
#define TRUE !FALSE

#define NOERROR 0
#define BUFOVFL 1       /* Buffer overflow error  */
#define ECHOTEST 2              /* Echo test error               */

#define SBUFSIZ 1024    /* Serial buffer size     */

int SError = NOERROR;
int portbase = 0;
void interrupt(*oldvects[2])();

static char ccbuf[SBUFSIZ];
unsigned int startbuf = 0;
unsigned int endbuf = 0;


/*************************************************************************
  Handle communications interrupts and put them in ccbuf
*************************************************************************/
void interrupt com_int(void)
{
  disable();
  if ((inportb(portbase + IIR) & RX_MASK) == RX_ID)
  {
        if (((endbuf + 1) & SBUFSIZ - 1) == startbuf) SError = BUFOVFL;
        ccbuf[endbuf++] = inportb(portbase + RXR);
        endbuf &= SBUFSIZ - 1;
  }
  /* Signal end of hardware interrupt */
  outportb(ICR, EOI);
  enable();
}

/*************************************************************************
  Output a character to the serial port
*************************************************************************/
```



```
int SerialOut(char x)
{
  long int timeout = 0x0000FFFFL;
  outportb(portbase + MCR,  MC_INT | DTR | RTS);
  /* Wait for Clear To Send from modem */
  while ((inportb(portbase + MSR) & CTS) == 0)
        if (!(--timeout))
          return (-1);
  timeout = 0x0000FFFFL;
  /* Wait for transmitter to clear */
  while ((inportb(portbase + LSR) & XMTRDY) == 0)
        if (!(--timeout))
          return (-1);
  disable();
  outportb(portbase + TXR, x);
  enable();
  return (0);
}


/*************************************************************************
  This routine returns the current value in the buffer
*************************************************************************/
int getccb(void)
{
  int res;
  if (endbuf == startbuf)
        return (-1);
  res = (int) ccbuf[startbuf++];
  startbuf %= SBUFSIZ;
  return (res);
}

/*************************************************************************
  Install our functions to handle communications
*************************************************************************/
void setvects(void)
{
  oldvects[0] = getvect(0x0B);
  oldvects[1] = getvect(0x0C);
  setvect(0x0B, com_int);
  setvect(0x0C, com_int);
}

/*************************************************************************
  Uninstall our vectors before exiting the program
*************************************************************************/
void resvects(void)
{
  setvect(0x0B, oldvects[0]);
  setvect(0x0C, oldvects[1]);
}

/*************************************************************************
  Turn on communications interrupts
*************************************************************************/
void i_enable(int pnum)
{
  int c;
  disable();
  c = inportb(portbase + MCR) | MC_INT;
  outportb(portbase + MCR, c);
  outportb(portbase + IER, RX_INT);
  c = inportb(IMR) & (pnum == COM1 ? IRQ4 : IRQ3);
  outportb(IMR, c);
  enable();
```

```
}

/*************************************************************************
  Turn off communications interrupts
*************************************************************************/
void i_disable(void)
{
  int c;
  disable();
  c = inportb(IMR) | ~IRQ3 | ~IRQ4;
  outportb(IMR, c);
  outportb(portbase + IER, 0);
  c = inportb(portbase + MCR) & ~MC_INT;
  outportb(portbase + MCR, c);
  enable();
}

/*************************************************************************
  Tell modem that we're ready to go
*************************************************************************/
void comm_on(void)
{
  int c, pnum;
  pnum = (portbase == COM1BASE ? COM1 : COM2);
  i_enable(pnum);
  c = inportb(portbase + MCR) | DTR | RTS;
  outportb(portbase + MCR, c);
}

/*************************************************************************
  Misc functions
*************************************************************************/
void comm_off(void)
{
  i_disable();
  outportb(portbase + MCR, 0);
}

void initserial(void)
{
  endbuf = startbuf = 0;
  setvects();
  comm_on();
}

void closeserial(void)
{
  comm_off();
  resvects();
}

int c_break(void)    /* Ctrl-break interrupt handler */
{
  i_disable();
  printf("\nStill online.\n");
  return(0);
}

/*************************************************************************
  Set the port number to use
*************************************************************************/
int SetPort(int Port)
{
  int Offset, far *RS232_Addr;
  switch (Port)
  { /* Sort out the base address */
```



```
        case COM1 : Offset = 0x0000;
                                break;
        case COM2 : Offset = 0x0002;
                                break;
        default   : return (-1);
  }
  RS232_Addr = MK_FP(0x0040, Offset);  /* Find out where the port is. */
  if (*RS232_Addr == NULL) return (-1);/* If NULL then port not used. */
  portbase = *RS232_Addr;              /* Otherwise set portbase      */
  return (0);
}

/*************************************************************************
  This routine sets the speed; will accept funny baud rates.
  Setting the speed requires that the DLAB be set on.
*************************************************************************/
int SetSpeed(int Speed)
{
  char c;
  int divisor;
  if (Speed == 0)            /* Avoid divide by zero */
        return (-1);
  else
        divisor = (int) (115200L/Speed);
  if (portbase == 0)
        return (-1);
  disable();
  c = inportb(portbase + LCR);
  outportb(portbase + LCR, (c | 0x80)); /* Set DLAB */
  outportb(portbase + DLL, (divisor & 0x00FF));
  outportb(portbase + DLH, ((divisor >> 8) & 0x00FF));
  outportb(portbase + LCR, c);          /* Reset DLAB */
  enable();
  return (0);
}

/*************************************************************************
  Set other communications parameters
*************************************************************************/
int SetOthers(int Parity, int Bits, int StopBit)
{
  int setting;
  if (portbase == 0) return (-1);
  if (Bits < 5 || Bits > 8)     return (-1);
  if (StopBit != 1 && StopBit != 2)     return (-1);
  if (Parity != NO_PARITY && Parity != ODD_PARITY && Parity != EVEN_PARITY)
        return (-1);
  setting  = Bits-5;
  setting |= ((StopBit == 1) ? 0x00 : 0x04);
  setting |= Parity;
  disable();
  outportb(portbase + LCR, setting);
  enable();
  return (0);
}

/*************************************************************************
  Set up the port
*************************************************************************/
int SetSerial(int Port, int Speed, int Parity, int Bits, int StopBit)
{
  if (SetPort(Port)) return (-1);
  if (SetSpeed(Speed)) return (-1);
  if (SetOthers(Parity, Bits, StopBit)) return (-1);
  return (0);
}
```

```
/*************************************************************************
  Main program
*************************************************************************/
main(int argc, char **argv)
{
  int port;
  int speed;
  int parity = NO_PARITY;
  int bits = 8;
  int stopbits = 1;
  int c = FALSE;
  int done = FALSE;
  unsigned int count = 0;
  unsigned int i = 0;
  FILE *in;
  unsigned char data;

  if (argc < 4)
  {
        printf("\nDSP RAM downloader by Steven J. Merrifield\n");
        printf("Syntax :  %s <ComPort> <BaudRate> <filename>\n",argv[0]);
        return(99);
  }
  port = atoi(argv[1]);
  if ((port < 1) | (port > 2))   /* Also covers if port == 0 */
  {
        printf("ComPort must be either 1 or 2\n");
        return(99);
  }
  if (port==1) port = COM1; else port = COM2;
  speed = atoi(argv[2]);
  if ((speed < 150) | (speed > 19200))  /* Also covers speed == 0 */
  {
        printf("BaudRate must be in the range 150-19200\n");
        return(99);
  }
  if (SetSerial(port, speed, parity, bits, stopbits) != 0)
  {
        printf("Serial port setup error.\n");
        return (99);
  }
  initserial();
  ctrlbrk(c_break);
  in = fopen(argv[3],"rb");
  if (!in)
  {
        printf("Error opening file.\n");
        return(99);
  }
/***************************************/
  clrscr();
  printf("¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿
¿¿¿¿¿¿¿¿¿¿");
  printf("¿      DSP RAM downloader     v1.3 Sept 1994     by  Steven J. Merrifi
eld      ¿");
  printf("¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿
¿¿¿¿¿¿¿¿¿¿\n");
  textcolor(14);
  printf("Filename  : "); cprintf("%s",strupr(argv[3]));
  printf("\nCOM port  : "); cprintf("COM%d",port);
  printf("\nBaud rate : "); cprintf("%d",speed);
  printf("\n\nPress <ENTER> to Start, or <ESC> to Cancel ");
  if ((c=getch())==27) done = TRUE;
  printf("\n\nSending data now : ");
/***************************************/
```



```
  fseek(in,0,0);    /* Check length of file, so we can subtract eof char */
  while (!feof(in) & !done)
  {
        fread(&data,1,1,in);
        count ++;
  }
  count--;       /* subtract eof character */
  fseek(in,0,0);
  while ((i<count) & (!SError) & (!done))    /* loop until eof or error */
  {
        fread(&data,1,1,in);
        SerialOut(data);
        delay(5);  /*   get overrun errors at 'C25 end without this delay */
        c = getccb() & 0x00FF;
        cprintf("*");
        if (kbhit())            /* test if keypressed during download */
        {
          if ((c=getch()) == 27)        /* was it <ESC> */
          {
                done = TRUE;
                printf("\nDownload aborted by user!");
          }
        }

/*   printf("Byte = %4u  DataSent = %2x  DataReceived = %2x\n",i,data,c); */
/*      if (data != c) SError = ECHOTEST; */

        i++;
  }
  fclose(in);
  printf("\nSent %u bytes (%X hex).\n",i,i);
  /* Check for errors */
  switch (SError)
  {
        case NOERROR: closeserial();
                                  return (0);
        case BUFOVFL: printf("\nBuffer Overflow.\n");
                                  closeserial();
                                  return (99);
        case ECHOTEST: printf("\nEcho test error.\n");
                                   closeserial();
                                   return(99);
        default:      printf("\nUnknown Error, SError = %d\n", SError);
                                  closeserial();
                                  return (99);
  }
}
```

```
/*---------------------------------------------------------------------*
   FILENAME:                    SERIAL.H

                 Some definitions used by SERIAL.C

 *---------------------------------------------------------------------*/

#define COM1            1
#define COM2            2
#define COM1BASE        0x3F8   /*  Base port address for COM1  */
#define COM2BASE        0x2F8   /*  Base port address for COM2  */

/*
    The 8250 UART has 10 registers accessible through 7 port addresses.
    Here are their addresses relative to COM1BASE and COM2BASE. Note
    that the baud rate registers, (DLL) and (DLH) are active only when
    the Divisor-Latch Access-Bit (DLAB) is on. The (DLAB) is bit 7 of
    the (LCR).

        o TXR Output data to the serial port.
        o RXR Input data from the serial port.
        o LCR Initialize the serial port.
        o IER Controls interrupt generation.
        o IIR Identifies interrupts.
        o MCR Send contorl signals to the modem.
        o LSR Monitor the status of the serial port.
        o MSR Receive status of the modem.
        o DLL Low byte of baud rate divisor.
        o DHH High byte of baud rate divisor.
*/
#define TXR             0       /*  Transmit register (WRITE) */
#define RXR             0       /*  Receive register  (READ)  */
#define IER             1       /*  Interrupt Enable          */
#define IIR             2       /*  Interrupt ID              */
#define LCR             3       /*  Line control              */
#define MCR             4       /*  Modem control             */
#define LSR             5       /*  Line Status               */
#define MSR             6       /*  Modem Status              */
#define DLL             0       /*  Divisor Latch Low         */
#define DLH             1       /*  Divisor latch High        */


/*---------------------------------------------------------------------*
  Bit values held in the Line Control Register (LCR).
        bit             meaning
        ---             -------
        0-1             00=5 bits, 01=6 bits, 10=7 bits, 11=8 bits.
        2               Stop bits.
        3               0=parity off, 1=parity on.
        4               0=parity odd, 1=parity even.
        5               Sticky parity.
        6               Set break.
        7               Toggle port addresses.
 *---------------------------------------------------------------------*/
#define NO_PARITY       0x00
#define EVEN_PARITY     0x18
#define ODD_PARITY      0x08



/*---------------------------------------------------------------------*
  Bit values held in the Line Status Register (LSR).
        bit             meaning
        ---             -------
        0               Data ready.
        1               Overrun error - Data register overwritten.
```



```
        2               Parity error - bad transmission.
        3               Framing error - No stop bit was found.
        4               Break detect - End to transmission requested.
        5               Transmitter holding register is empty.
        6               Transmitter shift register is empty.
        7               Time out - off line.
 *---------------------------------------------------------------------*/
#define RCVRDY          0x01
#define OVRERR          0x02
#define PRTYERR         0x04
#define FRMERR          0x08
#define BRKERR          0x10
#define XMTRDY          0x20
#define XMTRSR          0x40
#define TIMEOUT         0x80

/*---------------------------------------------------------------------*
  Bit values held in the Modem Output Control Register (MCR).
        bit             meaning
        ---             -------
        0               Data Terminal Ready. Computer ready to go.
        1               Request To Send. Computer wants to send data.
        2               auxillary output #1.
        3               auxillary output #2.(Note: This bit must be
                        set to allow the communications card to send
                        interrupts to the system)
        4               UART ouput looped back as input.
        5-7             not used.
 *---------------------------------------------------------------------*/
#define DTR             0x01
#define RTS             0x02
#define MC_INT          0x08


/*---------------------------------------------------------------------*
  Bit values held in the Modem Input Status Register (MSR).
        bit             meaning
        ---             -------
        0               delta Clear To Send.
        1               delta Data Set Ready.
        2               delta Ring Indicator.
        3               delta Data Carrier Detect.
        4               Clear To Send.
        5               Data Set Ready.
        6               Ring Indicator.
        7               Data Carrier Detect.
 *---------------------------------------------------------------------*/
#define CTS             0x10
#define DSR             0x20


/*---------------------------------------------------------------------*
  Bit values held in the Interrupt Enable Register (IER).
        bit             meaning
        ---             -------
        0               Interrupt when data received.
        1               Interrupt when transmitter holding reg. empty.
        2               Interrupt when data reception error.
        3               Interrupt when change in modem status register.
        4-7             Not used.
 *---------------------------------------------------------------------*/
#define RX_INT          0x01


/*---------------------------------------------------------------------*
  Bit values held in the Interrupt Identification Register (IIR).
```

```
        bit             meaning
        ---             -------
        0               Interrupt pending
        1-2             Interrupt ID code
                        00=Change in modem status register,
                        01=Transmitter holding register empty,
                        10=Data received,
                        11=reception error, or break encountered.
        3-7             Not used.
 *-----------------------------------------------------------------*/
#define RX_ID           0x04
#define RX_MASK         0x07


/*
    These are the port addresses of the 8259 Programmable Interrupt
    Controller (PIC).
*/
#define IMR             0x21   /* Interrupt Mask Register port */
#define ICR             0x20   /* Interrupt Control Port       */


/*
    An end of interrupt needs to be sent to the Control Port of
    the 8259 when a hardware interrupt ends.
*/
#define EOI             0x20   /* End Of Interrupt */


/*
    The (IMR) tells the (PIC) to service an interrupt only if it
    is not masked (FALSE).
*/
#define IRQ3            0xF7  /* COM2 */
#define IRQ4            0xEF  /* COM1 */
¿pt */


/*
    The (IMR) tells the (PIC) to service an interrupt only if it
    is not masked (FALSE).
*/
#define IRQ3            0xF7  /* COM2 */
#define IRQ4            0xEF  /* COM1 */
```

```
/*************************************************************************
  REVISED VERSION !!! - This document contains code which was added after
  submission of the original thesis. Where there are discrepancies between
  this and the original version, the code presented here should take
  precedence.

  This program handles both the encoding and decoding of the DTMF codes
  to and from the TMS320C25 DSP board. It reads characters from the keyboard
  and sends them via the serial port to the C25 which generates the DTMF
  tone. It also reads back the decoded tone from the C25 and displays it on
  the screen.
*************************************************************************/

#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "serial.h"       /* communication routines */

#define FALSE 0
#define TRUE !FALSE

#define NOERROR 0
#define BUFOVFL 1         /* buffer overflow error  */
#define RET_ERROR 99      /* all return(RET_ERROR); statements */

#define SBUFSIZ 1024      /* serial buffer size */

#define PadX 25           /* X position of keypad */
#define PadY 6            /* Y position of keypad */
#define back_color 1      /* color of background */
#define key_color 14      /* color of digits in keypad */
#define pad_color 15      /* color of key & keypad borders */

int key_count = 0,        /* number of keys pressed during encoding */
    send = FALSE,         /* flag for encoding routine */
    receive = FALSE,      /* flag for decoding routine */
    quit = FALSE,         /* flag to quit the current routine */
    EXITDOS = FALSE;      /* flag to quit the whole program */


void init_screen(void);   /* draws a fancy heading and background */
void ask_routine(void);   /* menu which prompts for the required routine */
void decode(void);        /* decoding routine */
void encode(void);        /* encoding routine */

void interrupt(*oldvects[2])();

int SError = NOERROR;
int portbase = 0;
static char ccbuf[SBUFSIZ];
unsigned int startbuf = 0;
unsigned int endbuf = 0;

/*************************************************************************
  Handle communications interrupts and put them in ccbuf
*************************************************************************/
void interrupt com_int(void)
{
  disable();
  if ((inportb(portbase + IIR) & RX_MASK) == RX_ID)
  {
    if (((endbuf + 1) & SBUFSIZ - 1) == startbuf) SError = BUFOVFL;
    ccbuf[endbuf++] = inportb(portbase + RXR);
    endbuf &= SBUFSIZ - 1;
  }
  /* Signal end of hardware interrupt */
  outportb(ICR, EOI);
  enable();
}

/*************************************************************************
  Output a character to the serial port
*************************************************************************/
int SerialOut(char x)
{
  long int timeout = 0x0000FFFFL;
  outportb(portbase + MCR,  MC_INT | DTR | RTS);
  /* Wait for Clear To Send from modem */
  while ((inportb(portbase + MSR) & CTS) == 0)
  if (!(--timeout))
    return (-1);
  timeout = 0x0000FFFFL;
  /* Wait for transmitter to clear */
  while ((inportb(portbase + LSR) & XMTRDY) == 0)
  if (!(--timeout))
    return (-1);
  disable();
  outportb(portbase + TXR, x);
  enable();
  return (0);
}

/*************************************************************************
  This routine returns the current value in the buffer
*************************************************************************/
int getccb(void)
{
  int res;
  if (endbuf == startbuf)
    return (-1);
  res = (int) ccbuf[startbuf++];
  startbuf %= SBUFSIZ;
  return (res);
}

/*************************************************************************
  Install our functions to handle communications
*************************************************************************/
void setvects(void)
{
  oldvects[0] = getvect(0x0B);
  oldvects[1] = getvect(0x0C);
  setvect(0x0B, com_int);
  setvect(0x0C, com_int);
}

/*************************************************************************
  Uninstall our vectors before exiting the program
*************************************************************************/
void resvects(void)
{
  setvect(0x0B, oldvects[0]);
  setvect(0x0C, oldvects[1]);
}

/*************************************************************************
  Turn on communications interrupts
*************************************************************************/
void i_enable(int pnum)
{
```

```
  int c;
  disable();
  c = inportb(portbase + MCR) | MC_INT;
  outportb(portbase + MCR, c);
  outportb(portbase + IER, RX_INT);
  c = inportb(IMR) & (pnum == COM1 ? IRQ4 : IRQ3);
  outportb(IMR, c);
  enable();
}

/*************************************************************************
  Turn off communications interrupts
*************************************************************************/
void i_disable(void)
{
  int c;
  disable();
  c = inportb(IMR) | ~IRQ3 | ~IRQ4;
  outportb(IMR, c);
  outportb(portbase + IER, 0);
  c = inportb(portbase + MCR) & ~MC_INT;
  outportb(portbase + MCR, c);
  enable();
}

/*************************************************************************
  Tell DSP board that we're ready to go
*************************************************************************/
void comm_on(void)
{
  int c, pnum;
  pnum = (portbase == COM1BASE ? COM1 : COM2);
  i_enable(pnum);
  c = inportb(portbase + MCR) | DTR | RTS;
  outportb(portbase + MCR, c);
}

/*************************************************************************
  Misc functions
*************************************************************************/
void comm_off(void)
{
  i_disable();
  outportb(portbase + MCR, 0);
}

void initserial(void)
{
  endbuf = startbuf = 0;
  setvects();
  comm_on();
}

void closeserial(void)
{
  comm_off();
  resvects();
}

int c_break(void)    /* Ctrl-break interrupt handler */
{
  i_disable();
  printf("\nWarning!   Ctrl-Break pressed... still online.\n");
  return(0);
}
```



```
/*************************************************************************
  Set the port number to use
*************************************************************************/
int SetPort(int Port)
{
  int Offset, far *RS232_Addr;
  switch (Port)
  { /* Sort out the base address */
    case COM1 : Offset = 0x0000; break;
    case COM2 : Offset = 0x0002; break;
    default   : return (-1);
  }
  RS232_Addr = MK_FP(0x0040, Offset);  /* Find out where the port is. */
  if (*RS232_Addr == NULL) return (-1);/* If NULL then port not used. */
  portbase = *RS232_Addr;              /* Otherwise set portbase      */
  return (0);
}

/*************************************************************************
  This routine sets the speed; will accept funny baud rates.
  Setting the speed requires that the DLAB be set on.
*************************************************************************/
int SetSpeed(int Speed)
{
  char c;
  int divisor;
  if (Speed == 0)                      /* Avoid divide by zero */
    return (-1);
  else
    divisor = (int) (115200L/Speed);
  if (portbase == 0)
    return (-1);
  disable();
  c = inportb(portbase + LCR);
  outportb(portbase + LCR, (c | 0x80)); /* Set DLAB */
  outportb(portbase + DLL, (divisor & 0x00FF));
  outportb(portbase + DLH, ((divisor >> 8) & 0x00FF));
  outportb(portbase + LCR, c);          /* Reset DLAB */
  enable();
  return (0);
}

/*************************************************************************
  Set other communications parameters
*************************************************************************/
int SetOthers(int Parity, int Bits, int StopBit)
{
  int setting;
  if (portbase == 0) return (-1);
  if (Bits < 5 || Bits > 8)     return (-1);
  if (StopBit != 1 && StopBit != 2)     return (-1);
  if (Parity != NO_PARITY && Parity != ODD_PARITY && Parity != EVEN_PARITY)
    return (-1);
  setting  = Bits-5;
  setting |= ((StopBit == 1) ? 0x00 : 0x04);
  setting |= Parity;
  disable();
  outportb(portbase + LCR, setting);
  enable();
  return (0);
}

/*************************************************************************
  Set up the port
*************************************************************************/
int SetSerial(int Port, int Speed, int Parity, int Bits, int StopBit)
```

```
{
  if (SetPort(Port)) return (-1);
  if (SetSpeed(Speed)) return (-1);
  if (SetOthers(Parity, Bits, StopBit)) return (-1);
  return (0);
}

/***************************************************************************
  End of serial handling and start of graphics/DTMF code - uses extended
  ASCII characters and these may not print correctly on paper.
***************************************************************************/
void init_screen()
{
  int i,j;
  textbackground(0);
  clrscr();
  textcolor(15);
  textbackground(back_color);
  cprintf("¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿
¿¿¿¿¿¿¿¿¿¿¿");
  cprintf("¿      Electronics IV (Honours) Project 1994    -    DTMF Encoder/Dec
oder      ¿");
  cprintf("¿                               by  Steven J. Merrifield
          ¿");
  cprintf("¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿
¿¿¿¿¿¿¿¿¿¿¿");
  for (i=5;i<25;i++)
  for (j=1;j<81;j++)
  {
    gotoxy(j,i);
      cprintf("¿");  /* ASCII 176 - may not print correctly on paper */
  }
}

/***************************************************************************
  Menu screen which waits for a response - uses extended ASCII characters
  and these may not print correctly on paper.
***************************************************************************/
void ask_routine()
{
  int x = 22, y = 10;
  char c;
  send = FALSE;
  receive = FALSE;
  EXITDOS = FALSE;
  gotoxy(x,y);   cprintf(" ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ ");
  gotoxy(x,y+1); cprintf(" ¿ 1. Encode (send) DTMF tones    ¿ ");
  gotoxy(x,y+2); cprintf(" ¿ 2. Decode (receive) DTMF tones ¿ ");
  gotoxy(x,y+3); cprintf(" ¿ 3. Quit to DOS                 ¿ ");
  gotoxy(x,y+4); cprintf(" ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ ");
  gotoxy(x+4,y+6); cprintf(" Enter your choice (1-3) :   ");
  gotoxy(x+31,y+6);  /* put cursor at prompt postion */
  do
  {      /* There's probably a better way to do this.... */
    c = getch();
  } while ((c != 49) & (c != 50) & (c != 51));

  switch (c)
  {
    case 49: send = TRUE; break;        /*  key = '1'  */
    case 50: receive = TRUE; break;     /*  key = '2'  */
    case 51: EXITDOS = TRUE; break;     /*  key = '3'  */
  }
}

/***************************************************************************
```



```
  DTMF decoding section - uses extended ASCII characters.
***************************************************************************/
void decode()
{
  int ch, key, x=12, y=10;
  init_screen();
  gotoxy(x,y);   cprintf(" ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿
¿ ");
  gotoxy(x,y+1); cprintf(" ¿ Received number :
¿ ");
  gotoxy(x,y+2); cprintf(" ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿
¿ ");
  gotoxy(x+21,y+1);  /* put cursor at prompt postion */
  quit = FALSE;
  while ((!SError) & (!quit))
  {
    ch = getccb();    /* read char from serial port buffer */
    if (ch != -1)
    {
      switch(ch)
      {
        case 85: key = '1'; break;   /* DSP end sends 55h to represent 0 */
        case 1:  key = '2'; break;
        case 2:  key = '3'; break;
        case 3:  key = 'A'; break;
        case 16: key = '4'; break;
        case 17: key = '5'; break;
        case 18: key = '6'; break;
        case 19: key = 'B'; break;
        case 32: key = '7'; break;
        case 33: key = '8'; break;
        case 34: key = '9'; break;
        case 35: key = 'C'; break;
        case 48: key = '*'; break;
        case 49: key = '0'; break;
        case 50: key = '#'; break;
        case 51: key = 'D'; break;
      }
    textcolor(14);
    cprintf("%c",key);
    }
    if (kbhit())
      if (getch()==27) quit = TRUE;
  }
}

/***************************************************************************
  This function "flashes" a button when it is pressed
***************************************************************************/
int flash_key(char k)
{ int x,y;
  switch(*strupr(&k)) {
    case '1': x = PadX + 5;  y = PadY + 2;  break;
    case '4': x = PadX + 5;  y = PadY + 5;  break;
    case '7': x = PadX + 5;  y = PadY + 8;  break;
    case '2': x = PadX + 12; y = PadY + 2;  break;
    case '5': x = PadX + 12; y = PadY + 5;  break;
    case '8': x = PadX + 12; y = PadY + 8;  break;
    case '0': x = PadX + 12; y = PadY + 11; break;
    case '3': x = PadX + 19; y = PadY + 2;  break;
    case '6': x = PadX + 19; y = PadY + 5;  break;
    case '9': x = PadX + 19; y = PadY + 8;  break;
    case 'A': x = PadX + 26; y = PadY + 2;  break;
    case 'B': x = PadX + 26; y = PadY + 5;  break;
    case 'C': x = PadX + 26; y = PadY + 8;  break;
    case 'D': x = PadX + 26; y = PadY + 11; break;
```

```
    case '*': x = PadX + 5;  y = PadY + 11; break;
    case '#': x = PadX + 19; y = PadY + 11; break;
    default: return(1);  /* invalid key pressed */
  }
  key_count ++;          /* so we know where to put the next character */
  gotoxy(x-1,y);
  textcolor(back_color);
  textbackground(3);
  cprintf(" %c ",k);     /* flash the background color */
  delay(200);
  gotoxy(x-1,y);
  textcolor(key_color);
  textbackground(back_color);
  cprintf(" %c ",k);    /* then restore it to the way it was */
  gotoxy(PadX+16+key_count,PadY+15);
  cprintf("%c",k);      /* write the key pressed at the correct spot */
  return(0);
}

/*************************************************************************
  DTMF encoding section - uses extended ASCII characters
*************************************************************************/
void encode()
{
  int SendChar,ch;
  char k;
  init_screen();
  textcolor(pad_color);
  gotoxy(PadX,PadY);    cprintf("¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿");
  gotoxy(PadX,PadY+1);  cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+2);  cprintf("¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿");
  gotoxy(PadX,PadY+3);  cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+4);  cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+5);  cprintf("¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿");
  gotoxy(PadX,PadY+6);  cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+7);  cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+8);  cprintf("¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿");
  gotoxy(PadX,PadY+9);  cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+10); cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+11); cprintf("¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿   ¿  ¿");
  gotoxy(PadX,PadY+12); cprintf("¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿¿¿¿¿  ¿");
  gotoxy(PadX,PadY+13); cprintf("¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿");
  textcolor(key_color);
  gotoxy(PadX+5,PadY+2);   cprintf("1"); gotoxy(PadX+12,PadY+2);  cprintf("2");
  gotoxy(PadX+19,PadY+2);  cprintf("3"); gotoxy(PadX+26,PadY+2);  cprintf("A");
  gotoxy(PadX+5,PadY+5);   cprintf("4"); gotoxy(PadX+12,PadY+5);  cprintf("5");
  gotoxy(PadX+19,PadY+5);  cprintf("6"); gotoxy(PadX+26,PadY+5);  cprintf("B");
  gotoxy(PadX+5,PadY+8);   cprintf("7"); gotoxy(PadX+12,PadY+8);  cprintf("8");
  gotoxy(PadX+19,PadY+8);  cprintf("9"); gotoxy(PadX+26,PadY+8);  cprintf("C");
  gotoxy(PadX+5,PadY+11);  cprintf("*"); gotoxy(PadX+12,PadY+11); cprintf("0");
  gotoxy(PadX+19,PadY+11); cprintf("#"); gotoxy(PadX+26,PadY+11); cprintf("D");
  gotoxy(PadX-1,PadY+15);
  textcolor(15);
  cprintf(" Number to dial :                  ");
  gotoxy(PadX+17,PadY+15);
  quit = FALSE;
  key_count = 0;
  while ((!SError) & (!quit))
  {
    ch = getccb();             /* read char from serial port buffer */
    if (ch != -1) putch(ch);   /* if buffer is not empty, then write char */
    k = getch();
    flash_key(k);
    switch(*strupr(&k))
    {
      case 27: quit = TRUE; break; /* key = 'ESC' */
```

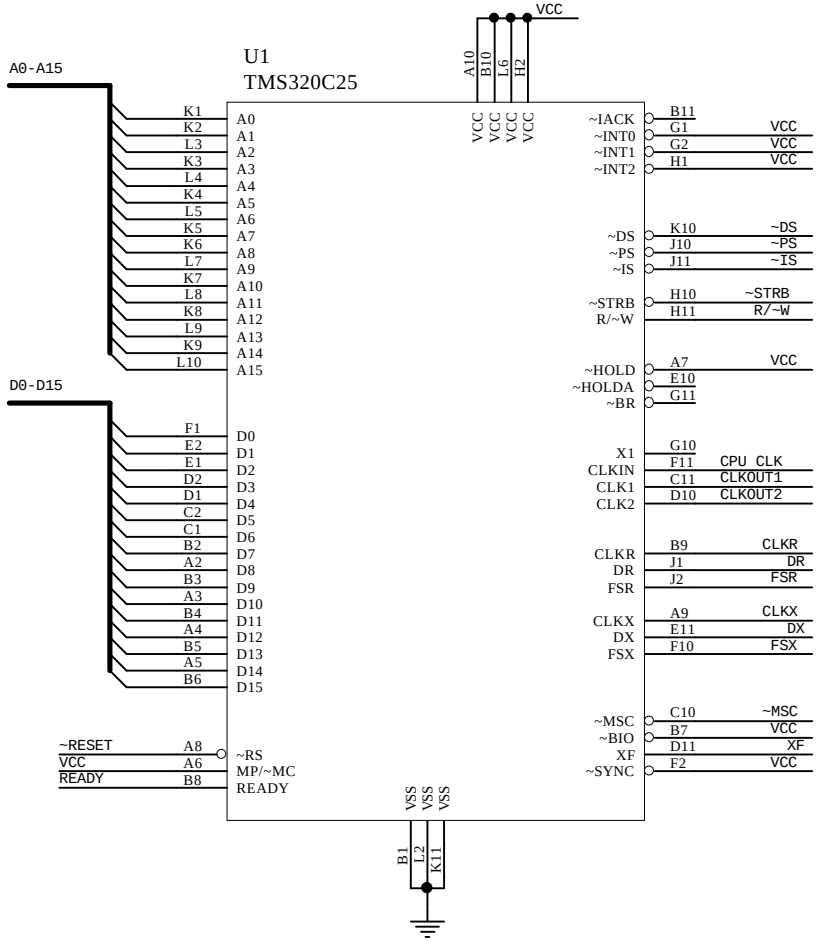


```
      case 48: SendChar = 0x00; break;
      case 49: SendChar = 0x01; break;
      case 50: SendChar = 0x02; break;
      case 51: SendChar = 0x03; break;
      case 52: SendChar = 0x04; break;
      case 53: SendChar = 0x05; break;
      case 54: SendChar = 0x06; break;
      case 55: SendChar = 0x07; break;
      case 56: SendChar = 0x08; break;
      case 57: SendChar = 0x09; break;
      case 65: SendChar = 0x0A; break;
      case 66: SendChar = 0x0B; break;
      case 67: SendChar = 0x0C; break;
      case 68: SendChar = 0x0D; break;
      case 42: SendChar = 0x0E; break;   /*    key = '*'    */
      case 35: SendChar = 0x0F; break;   /*    key = '#'    */
    }
    if (k != 27) SerialOut(SendChar);
    delay(5);    /* get overrun errors at 'C25 end without this */
  }
}

/*************************************************************************
  Main program
*************************************************************************/
main(int argc, char **argv)
{
  int port;
  int speed;
  int parity = NO_PARITY;
  int data_bits = 8;
  int stop_bits = 1;
  if (argc < 3)
  {
    printf("DTMF encoder/decoder front end for TMS320C25 DSP board.\n");
    printf("Syntax :  %s <ComPort> <BaudRate>\n",argv[0]);
    return(99);
  }
  port = atoi(argv[1]);
  if ((port < 1) | (port > 2))   /* also covers if port == 0 (error) */
  {
    printf("Com port must be either 1 or 2\n");
    return(RET_ERROR);
  }
  if (port==1) port = COM1; else port = COM2;
  speed = atoi(argv[2]);
  if ((speed < 150) | (speed > 19200))  /* also covers speed == 0 (error) */
  {
    printf("Baud rate must be in the range 150 - 19200\n");
    return(RET_ERROR);
  }
  if (SetSerial(port, speed, parity, data_bits, stop_bits) != 0)
  {
    printf("Error setting up serial port.\n");
    return (RET_ERROR);
  }
  initserial();
  ctrlbrk(c_break);

  do
  {
    init_screen();
    ask_routine();
    if (send == TRUE) encode();
    if (receive == TRUE) decode();
  } while (EXITDOS != TRUE);
```

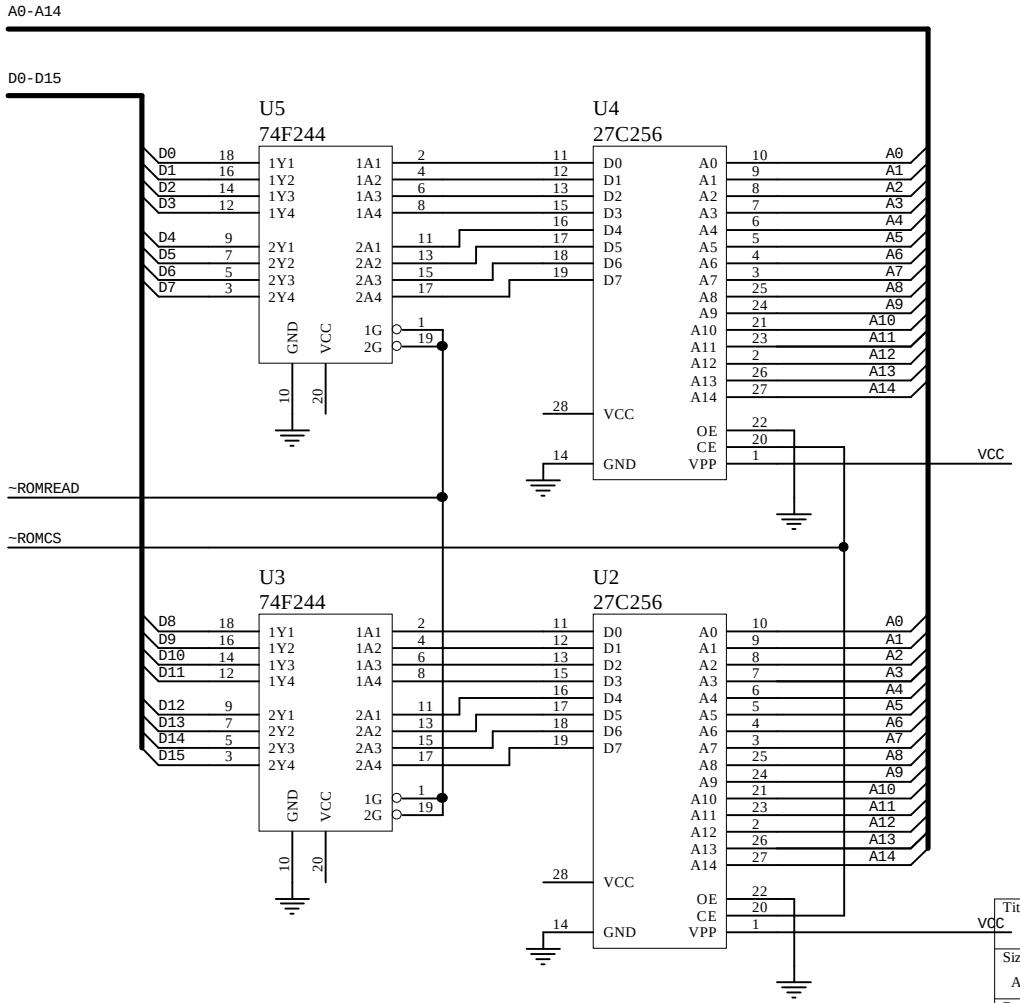
```
  textcolor(7);
  textbackground(0);
  clrscr();
  /* Check for errors */
  switch (SError)
  {
    case NOERROR: closeserial();
                  return (0);
    case BUFOVFL: printf("\nBuffer Overflow.\n");
                  closeserial();
                  return (RET_ERROR);
    default:      printf("\nUnknown Error, SError = %d\n", SError);
                  closeserial();
                  return (RET_ERROR);
  }
}
```

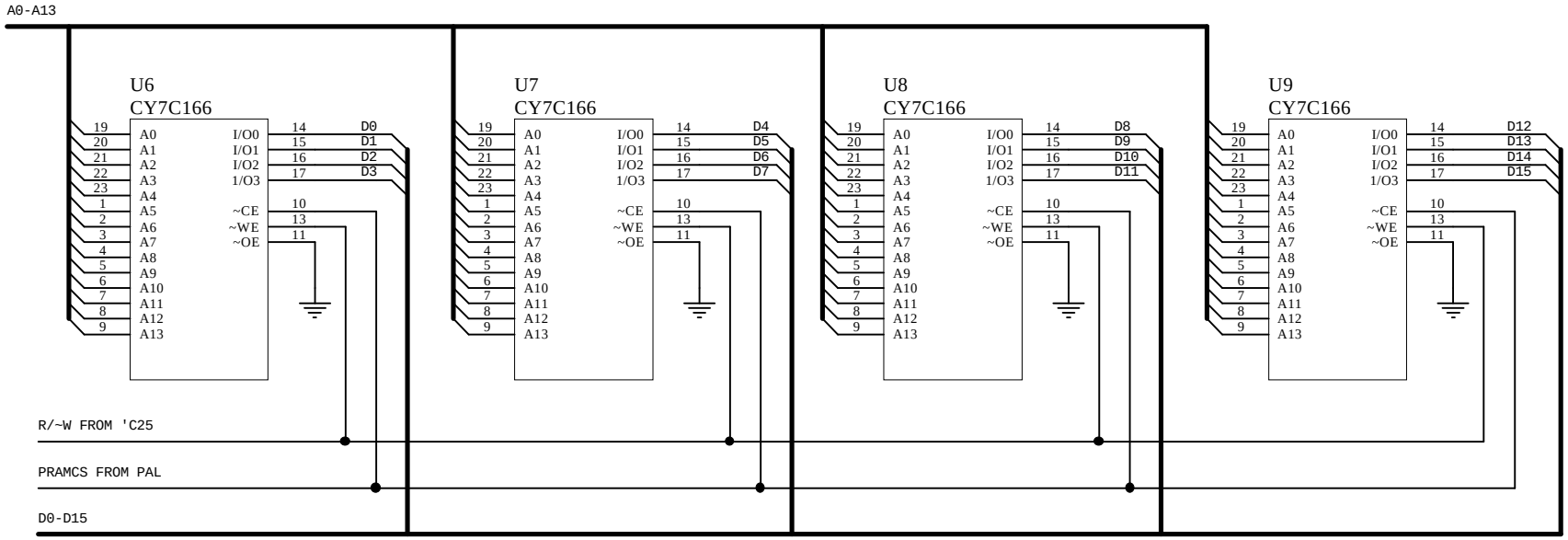


TMS320C25 DIGITAL SIGNAL PROCESSOR

STEVEN J. MERRIFIELD

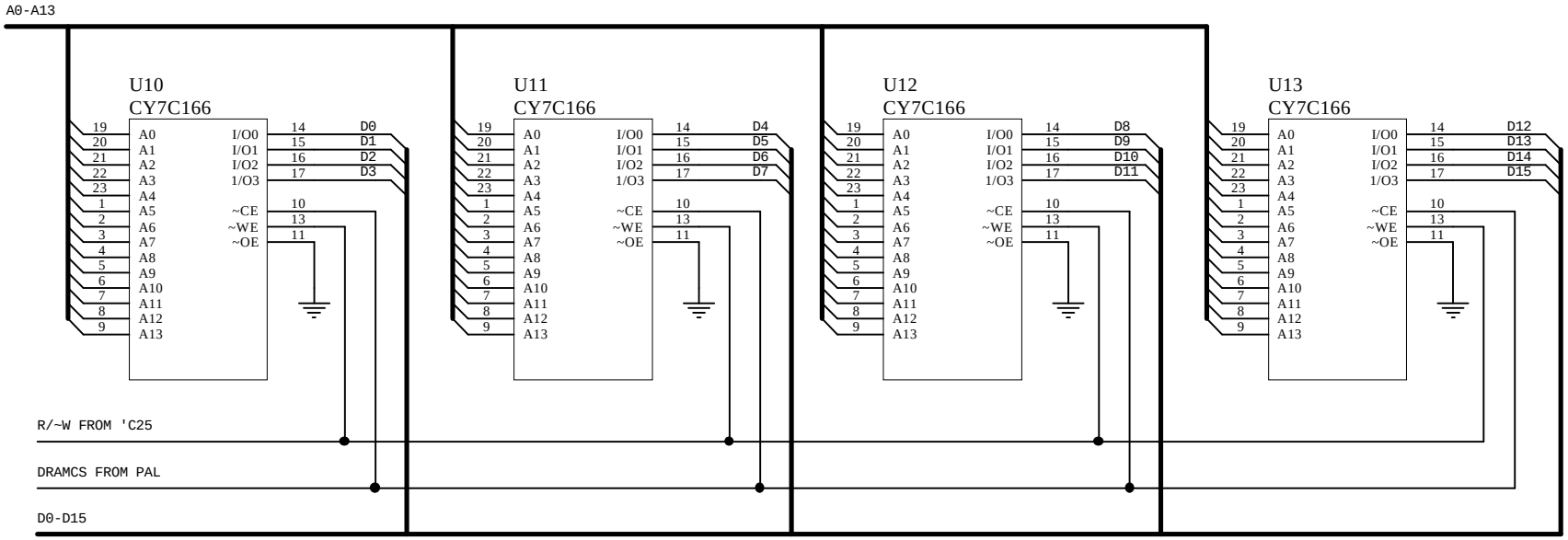
Title		
Size: A4	Number	Revision
Date: 19-Jun-1995		Sheet of
File: C:\TEMP2\DSP-CHIP.S01		Drawn By:


A0-A14
D0-D15
U5
74F244
U4
27C256
U3
74F244
U2
27C256
~ROMREAD
~ROMCS
VCC
EPROM
CIRCUITRY
Title
Size
A4
Number
Revision
Date: 19-Jun-1995
Sheet of
File: C:\TEMP2\ROM.S01
Drawn By:



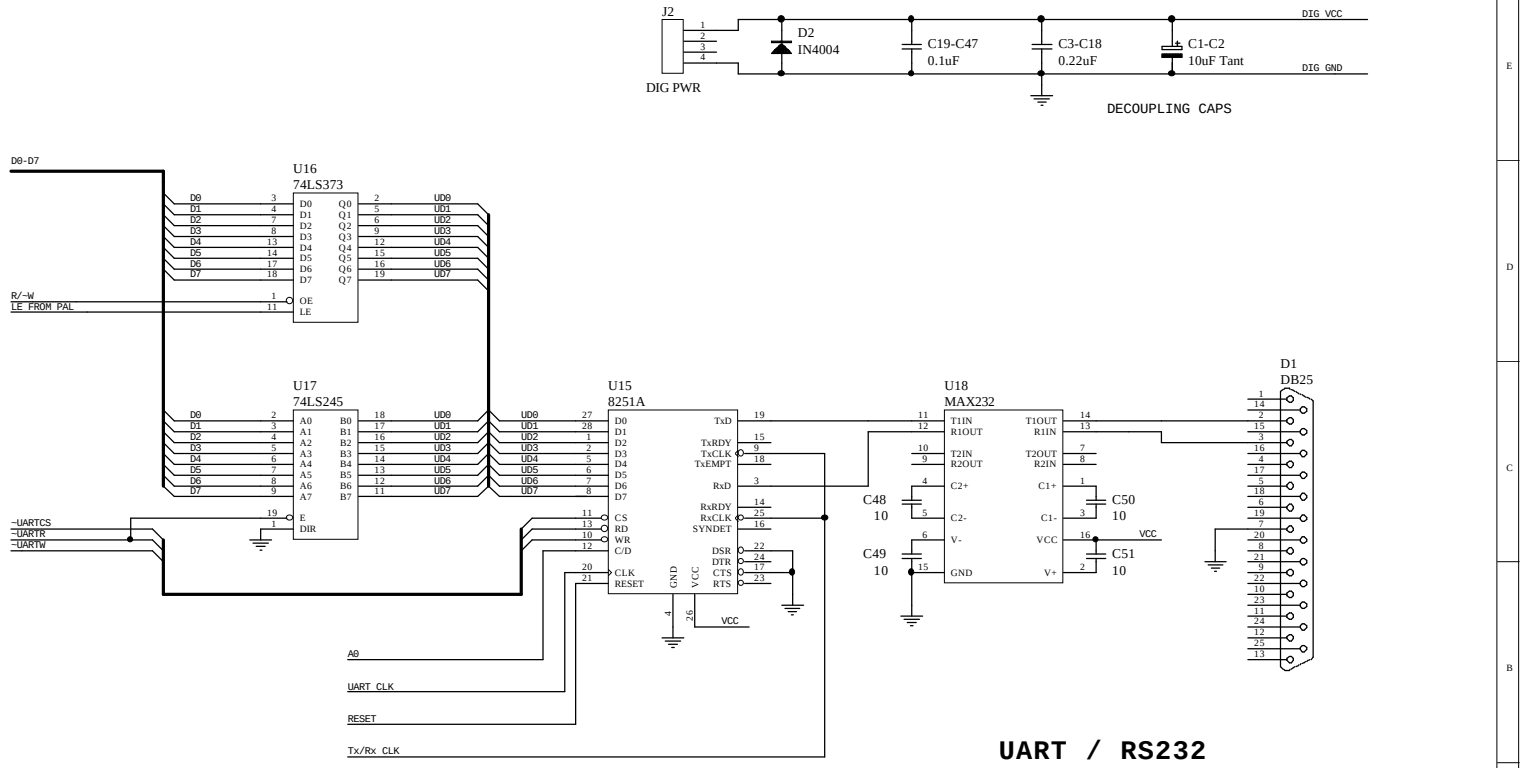
PROGRAM RAM

Title		
Size: A4	Number	Revision
Date: 19-Jun-1995		Sheet of
File: C:\TEMP2\PRAM.S01		Drawn By:



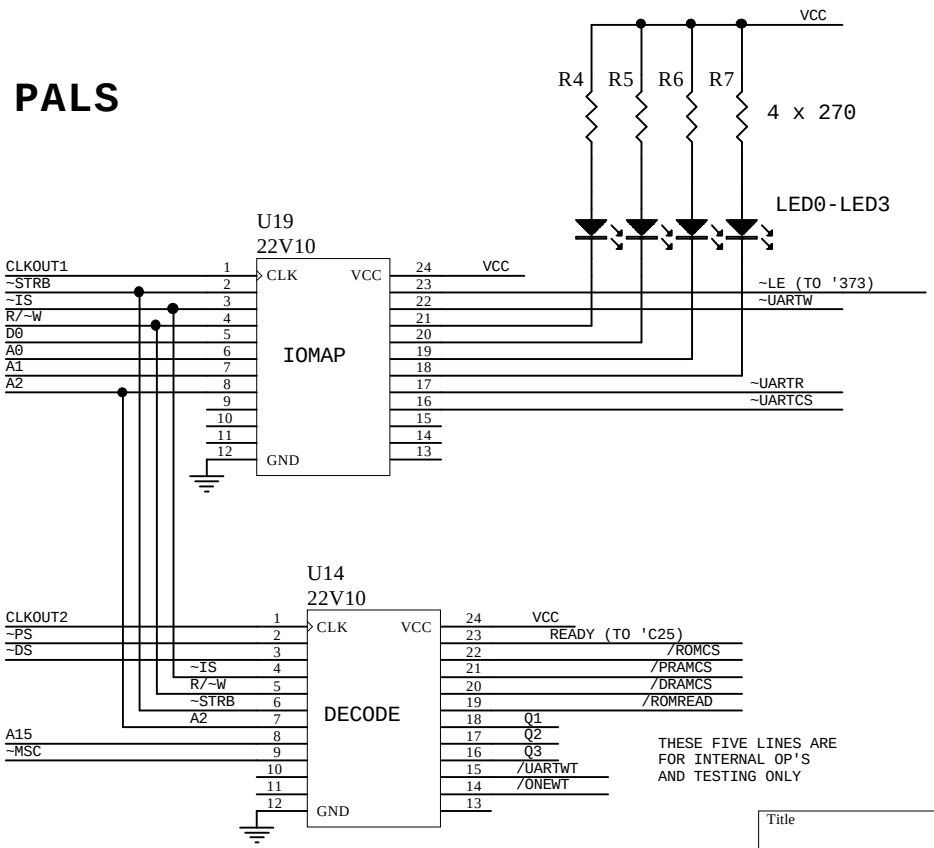
DATA RAM

Title		
Size: A4	Number	Revision
Date: 19-Jun-1995		Sheet of
File: C:\TEMP2\DRAM.S01		Drawn By:

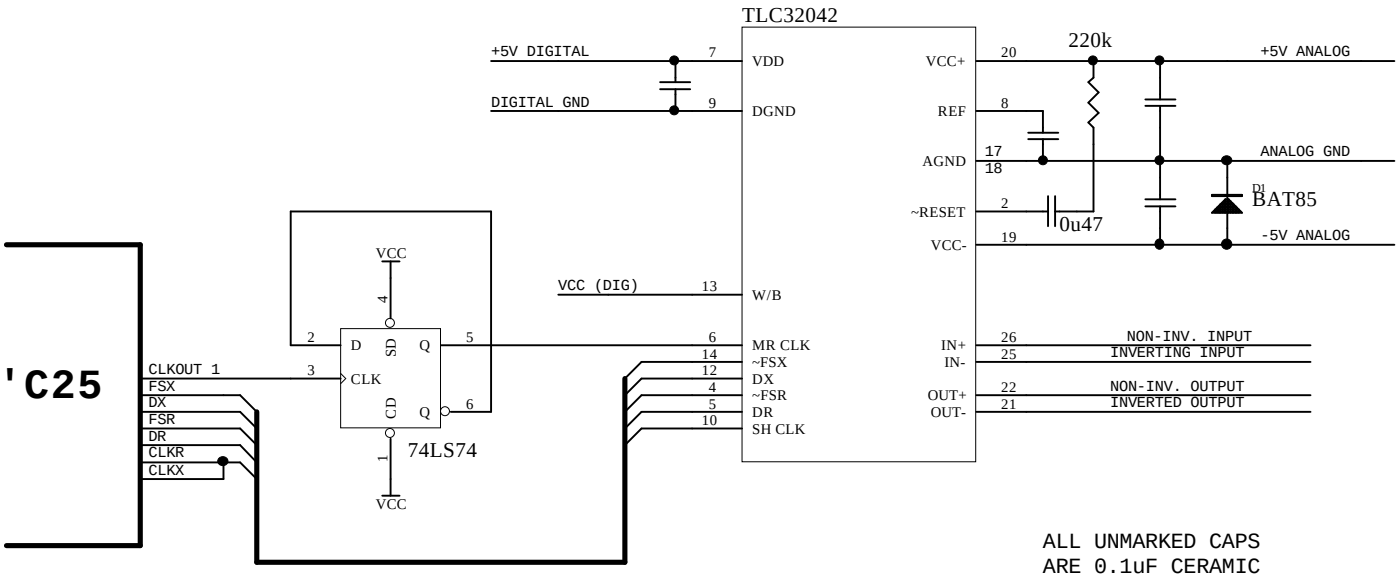


Title			
Size A3	Number		Revision
Date: 19-Jun-1995		Sheet of	
File: C:\TEMP2\SER-INT.S01		Drawn By:	

PALS

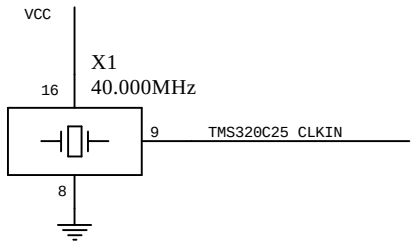


Title		
Size: A4	Number	Revision
Date: 19-Jun-1995		Sheet of
File: C:\TEMP2\22V10S.S01		Drawn By:

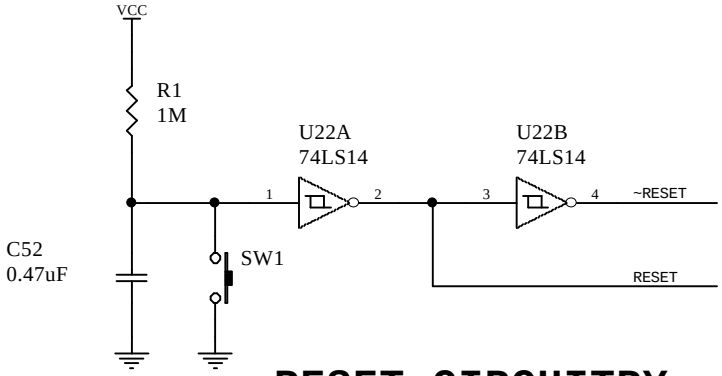


ANALOG INTERFACE

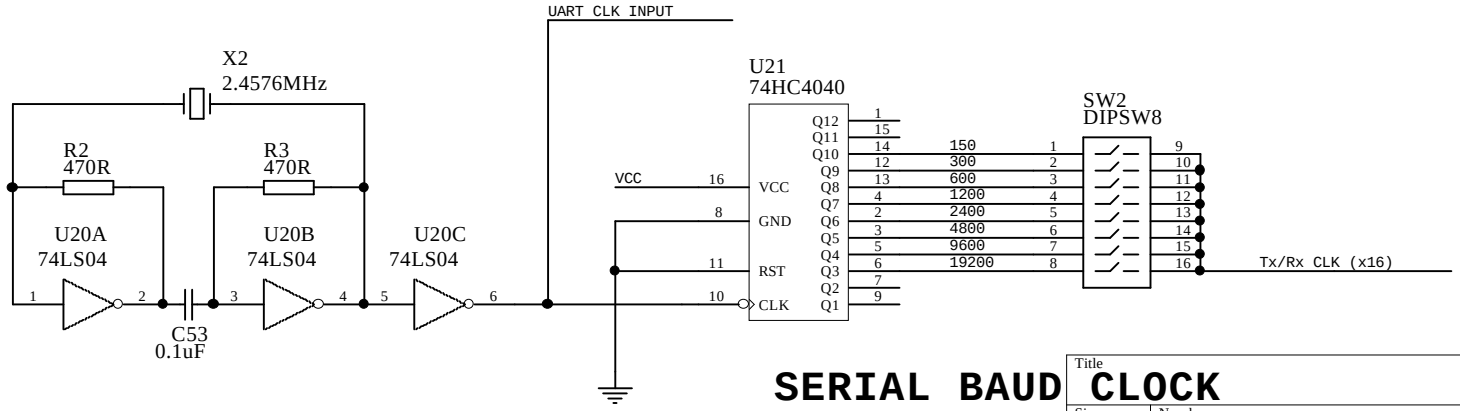
Title		
Size: A4	Number	Revision
Date: 19-Jun-1995		Sheet of
File: C:\TEMP2\AIC.S01		Drawn By:



SYSTEM CLOCK

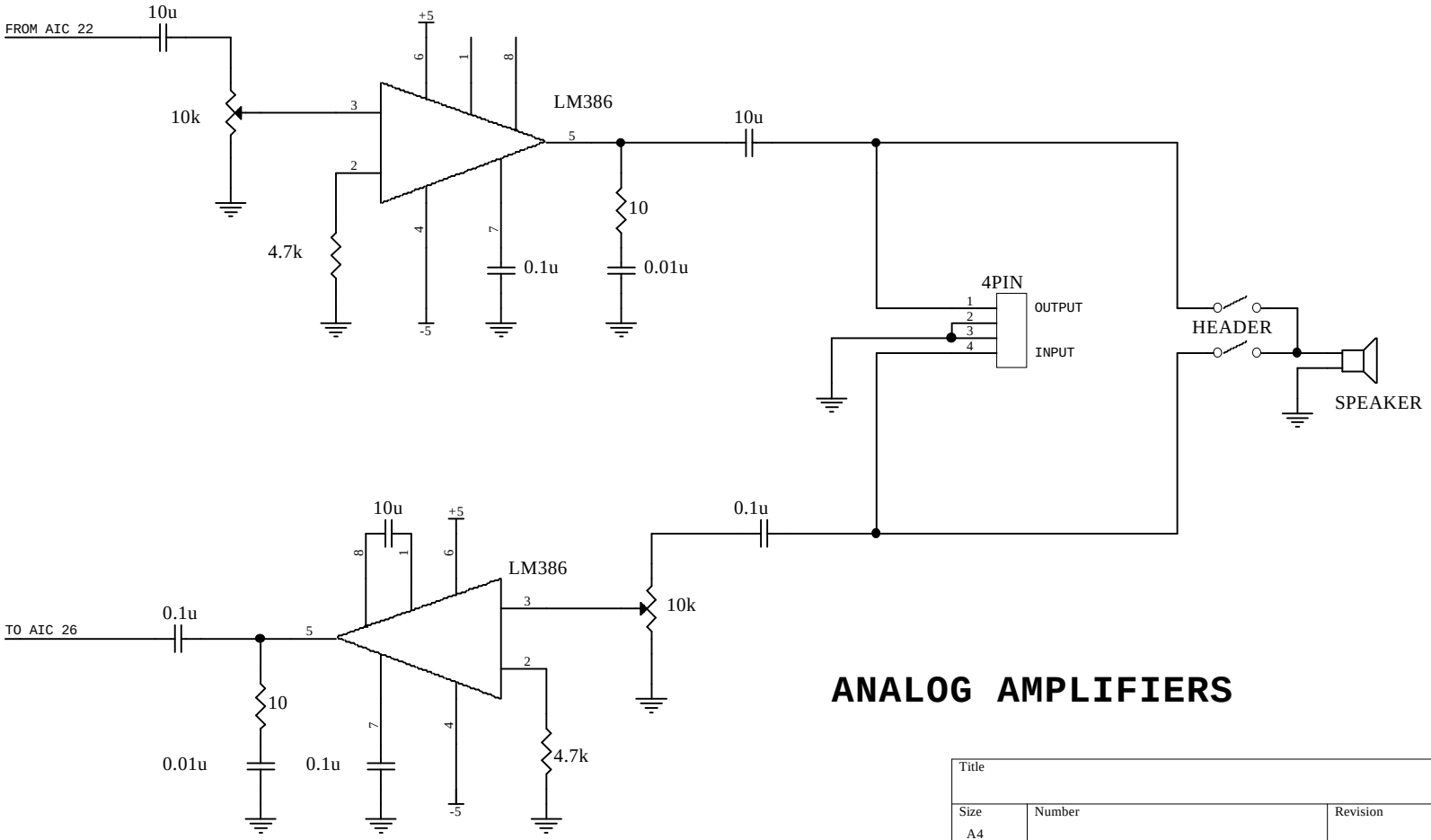


RESET CIRCUITRY



SERIAL BAUD CLOCK

Title: CLOCK		
Size: A4	Number:	Revision:
Date: 19-Jun-1995		Sheet of
File: C:\TEMP2\CLOCKS.S01		Drawn By:



ANALOG AMPLIFIERS

Title		
Size A4	Number	Revision
Date: 19-Jun-1995		Sheet of
File: C:\TEMP2\AMPS.S01		Drawn By: