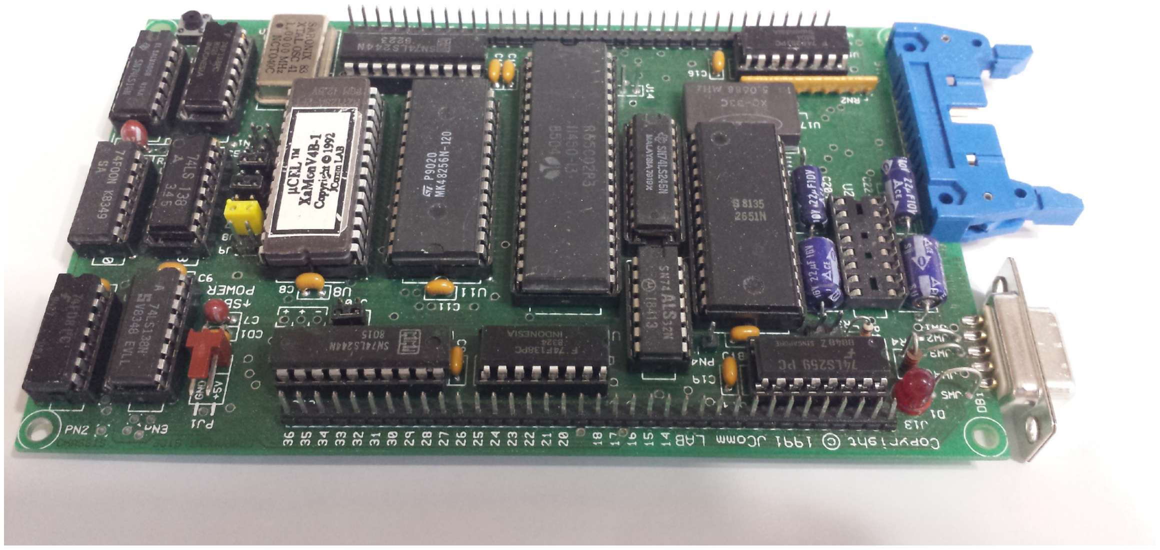
Experimenting With Embedded Control Using the μCEL ™ Development System

by Johnathan Conley


SBC65V1B Single Board Computer

Chapter 1

Quick Summary

This chapter is for those of you who want to get right to work, and already have some experience with microprocessors or microcontrollers. The following is a summary of the **µCEL** ™ development system.

Getting Started

1.) Hook up the **SBC65V1B** controller
2.) Initiate the power up blink test
3.) Browse through memory and exercise I/O using the SBC monitor & **JTerm**

Development Cycle Overview

1.) Copy Template.ASM to your new source code file name: "YourFile.ASM"
2.) Use your editor to edit your source code
3.) Assemble your source code using **JAsm**, the cross-assembler
4.) Download your "ramware" object .ROC file created by **JAsm** using **JTerm**
5.) Run Your Program Using **JTerm**
6.) Debug your code
7.) Repeat steps 2 through 6 until done

Applications:

Later we'll put the **µCEL** ™ system to use by making a capacitance meter. We'll combine a little bit of hardware with some software to implement this project. In later experiments, we'll either use a component to implement a simple function or we will combine components to make a more complex interface.

Make it DO something!!

Design Ideas:

- High Speed serial interface
- Battery backed static ram
- Real time clock
- Push button switch interface
- Keypad and LCD Display driver
- Intelligent phone circuits (answering machines, etc.)
- Plotter buffer/translator
- Eprom burner/memory extention EPROM emulator
- Custom remote terminal
- Remote temperature monitoring interface
- Intelligent Stepper motor driver
- Programmable drilling/routing engraving

- Pulse-electroplating controller
- Data logger
- Remote test equipment
- Weather Station
- Hydroponic garden and greenhouse experiments
- On-board automobile/dragster computer
- Air pollution monitor
- Custom HAM radio devices
- Laboratory experiment monitor
- Remote intelligent relay control
- Custom burglar alarm
- Custom sprinkler systems
- Networking Controller
- Single-chip µController development system
- Single-chip µC emulator
- Multiple Time Delay Relays
- Model railroad controller
- Industrial Control

Applications Limited only by your imagination! (and bucks & time)

Getting Started-Details

Hook up the SBC65V1B

Connect the power cable. This SBC (single board computer) runs on +5V power. Plug in the connector PJ1, included with your SBC, into its header. Note that the connector is "keyed" so that the flat part of the connector contacts the pc board (the 'nib' will be on the topside). Use the enclosed documentation to locate power & ground on the pcb - side of the connector, or use an ohmeter. Use pin16 of U6 to detect the +5V line and pin 8 of U6 for ground.

Measure your power supply voltage with a voltmeter before you connect it to your SBC. It should read +5V. Once you are certain you know where to connect power, attach the two leads from the SBC connector to your power supply with the power *off*. Do a visual and an ohm check before you apply power. Verify that your connections are correct. Now, apply power.

Your first indication of a working unit will be led D1. It should be lit. If not, press the reset button S1. If there is no led indication, check the voltages on the IC's with a voltmeter.

Your power supply should be able to supply at least 350 milli-amps (mA).

Power-up Blink Test

Locate the 2-pin header J15 near pin 4 of U19. Slide a shorting jumper over J15 (new units will come with this jumper installed) and then apply power and depress the reset button S1. The led D1 should blink at about a once-per three-second rate.

Browse Through Memory

Connect the female end of the serial cable supplied, to the male nine-pin connector on your IBM or compatible. The male end of the cable plugs into the female nine-pin serial connector on the **SBC65V1B**.

Run the terminal program "**JTerm**." You should observe a menu with lots of distractors. Near the bottom of the screen you should see a "1" being sent to the screen from the SBC each time the led blinks. This verifies that your serial connection is complete and your SBC is working.

Remove the shorting jumper from J15 on the SBC. Within a few seconds, you should see the message **JComm LAB uCEL (TM) XaMonV4B-1**. You are now ready to browse.

First type the "4" command. You will see the buffer address, and the buffer size. Now, type the "6" command. This is a memory dump. Notice the address at the left of the screen, and the corresponding sixteen bytes. Now type the "7" command, followed by two hex letters "A-F" or "0-9." This allows you to fill memory with any hex value you would like. Notice that all of these operations begin at the starting address specified, and are as long as the buffer size specified. Control-N will give you the **JTerm** menu. Take time to read and test each command. The worst thing that can happen while you are exploring is that you will have to momentarily disconnect power from the SBC and depress its reset button.

One exception, the Control - D 'download' command might hang up your PC if you do not have a ROC file present (in your directory) ready to download.

Development Cycle Details

Copy the template.

The Template.ASM can be copied to a different file name at the start of each new project. This file gives you a foundation for some consistent documentation. You may decide on a different format or have some ideas of your own. If you do, don't hesitate to implement them. Doing regular, readable and consistent documentation is a **REAL** good habit to get into!!

Edit your source code

Some like to use the editor in a "shell" like Directory Commander (shareware) to both edit their source code and run **JAsm** or **JTerm**.

Cross-assemble your source code

by running JAsm. A menu will present itself of all the .ASM files contained in the directory you run JAsm from. Use the up and down arrow keys to highlight the desired file, then press the "return" key.

There is a file called Xasm.CFG that instructs the assembler as to the type of files to generate. If all four values are "FALSE", the assembler will only generate a .ROC file. The .ROC file is the executable or *object* code. If this object code is burned into an Eprom, the code is called **firmware**. If the code is downloaded into the **SBC65V1B** ram, it may be called "ramware."

Edit the Xasm.CFG file and used different BOOLEAN (TRUE or FALSE) values, and see what types of files **JAsm** generates.

Run JTerm and Download your Ramware

Run **JTerm** and notice the menu presented. There are several commands to explore, but you don't have to be overwhelmed. Just take each command one-at-a-time. For now, type control-D and notice the menu of the .ROC files on your directory. Select the file you want to download and press return. The downloader will automatically set the buffer size, and send your code starting at the "ORG" address previously set by using the "1" command. The default value for **ORG is 0500 hex.**

Run your program using JTerm. All you have to do now is press **control-z** and then "**G**" for "Go.'

Debug your code

There is some bad news and some good news. The bad news is that there is no debugger per se with this system. The good news is that you won't need one if you follow healthy *top-down structured programming* practices. There is a BRK instruction you can insert into your source code, that will halt your program and display the current registers.

There is even a vector you can install that will allow you to insert a "display-key-variables" routine so that you can monitor some of the things that a more complete debugger would let you look at.

A list of some of the more common errors made when assembly programming are included in the section on the **JAsm** cross-assembler.

Review

The development cycle is

1.) Edit
2.)Assemble
3.)Download
4.)Run
5.)Debug until your program behaves the way it is supposed to, you give up, or run out of time.

You will find many useful subroutines/procedures included in the **XaMonV4B** monitor that should ease your programming burden. The Hello.ASM source code is included so that you can "practice" going through the cycle. Just a note of comparison: One C compiler's executable just to put "Hello World" on the screen was 3K bytes, C++ 18K! **µCEL** includes "Hello World" source code so you can see how much code this system uses. (Its not much) **P.S. Don't forget to back up your files!**

Chapter 2

Introduction

How many different computers do you use each day? For many of you the answer may be one or two. In reality you may be using more computing power than you are aware of. Many types of appliances and machinery "hide" a CPU or micro-controller within, *dedicating* it to a specific task.

A term used to describe the use of a computer in this way is *embedded control*; the computer is literally built right into the device or product. By using built-in "intelligence," your design can have more features, flexibility, and be easier to use.

Embedded control is used all around you. Devices such as microwave ovens, burglar alarms, automated sprinkler systems and laser printers have dedicated microcontrollers or computers as part of their "guts."

Embedded control has been identified as a key technology for the 90's, and can be just plain fun and useful for widget builders like me and you!

This article describes **µCEL**; a micro Control Experimenters Laboratory. At the heart of this development system is a 65C02-based single board computer/controller. This system includes a firmware monitor, communication software and a cross-assembler that runs on an IBM PC or compatible. The entire system can be put together for around $250.00; and way less than that if you are willing to raid your parts bins and do the construction yourself!

Keeping with the "hacker" spirit, **µCEL** is an open design. The controller has built-in features to make it easy to program and interface. The goal was to make the hardware and software both available and understandable.

µCEL's computer/controller can be used as a platform to learn about many aspects of embedded control and interface design, and can be literally embedded into your project! You can use the **µCEL** development system to teach yourself about:

Hardware design: How a microprocessor, clock, RAM and ROM and some "glue" chips combine to make a small computer system

**Interface design: How to get that small system to interact with the outside world, using only a common plug board, a few external parts and some #22 solid
wire.**

Machine language programming: How to hand assemble the CPU's hex instruction codes, hand-enter and run them using the **µCEL** monitor.

Assembly language programming: Writing code that the 65C02 microprocessor executes at the lowest level; developing that code using a Cross-Assembler that runs on a PC.

Monitor design: How small sequences of assembly language instructions can be combined to make subroutines.

These subroutines can then be combined to create procedures that mimic some of the operations used in high-level languages, such as **Pascal** and **Modula-2**.

Assembler design: How a high level language can be used to write a program that automates the generation of assembly language programs.

Development utilities: How a simple dumb-terminal program works.

How to up and download programs/data between the **µCEL** controller and a host computer. How to use a monitor to debug your assembly programs.

You don't need to let all this learning potential scare you. **µCEL** was designed so that you can be up and running very quickly. You can implement an interface design (such as the capacitance meter described later,) in an evening! From the above list, you can see that it is possible to explore embedded control and related topics in depth. The fun part of all this is that you can learn a little bit at a time. Stick with it, and you can find yourself designing and building some pretty sophisticated **PROGRAMMABLE** projects!

System Overview:

The µCEL SBC65V1B controller

For a frame of reference, the **SBC65V1B** can be viewed as a 3.5" X 5.8" one slot "mother board" (that in many ways mimics the 'old' Apple II+) that uses an IBM compatible computer for its keyboard, disk drive and screen. Before you get too excited I'll add this disclaimer; This is ***NOT*** an Apple clone. It does ***NOT*** use any of the Apple's monitor routines. Your old Apple II+ assembly language program will require some modification before it will run on this board. Fans of the Apple II+ will probably find the monitor provided with the **SBC65V1B** easy to learn, and the routines easy to adapt to your particular application.

The "slot" for your interface is simple to use as well. This interface is called "**H-Buss**." (it kinda looks like an 'H') To make your interfacing task easier, the **H-Buss** includes buffered address and data lines, decoded select lines and various clock and control lines. You can implement that hot idea of yours and build-ityourself rather than buy-from-the-shelf. Along with some parts, all you need for small interfacing experiments is a commonly available breadboard and number 22 solid wire, and a good idea.

µCEL TM System Features:

Size:

- 3.5" X 5.8".

CPU:

- 1 MHz 65C02

RAM:

- Jumper programmable 2K,8K, 32K.

Eprom:
- Jumper programmable 2K, 4K, 8K, 16K, 32K OR 64K.

Components:
- Uses common, readily available TTL 'LS' parts.

I/O:
- 8-individually programmed inputs
- 8-individually programmed out puts.
- Serial port with Xon handshaking, and user-programmable baudrate.

H-Buss:
- A "home builder" computer bus that receives an interface card.
- It is possible to create a tight and solid module measuring 2" X 3.5" X 5.8".
- It provides buffered address and data lines, extra decodes and clock and control signals for your inter face.
- Simple interfaces become much easier to breadboard, using a "3M" type plugboard and #22 (.025 dia) solid hook up wire.

XaMonV4B Monitor:
- More than 132 subroutines, using 4K of the Eprom address space.
- Contains routines to handle low level I/O operations.
- Includes interrupt routines, block memory moves, Ascii memory dump to screen, upload and downloading, entering data, manipulating and redirecting I/O from commands given by a host computer.
- Routines can be modified for use by Atari, Pet, Kim, Sym, Aim, and Apple II+ users not having the 65C02 processor chip.

Power:
- 5V @ 200-450 ma, depending on components chosen.

JAsm Cross assembler:
- Runs on IBM compatibles.
- Does not require a hard disk.
- Generates 6502 and 65C02 object code.
- Accepts source code created from any word processor or editor that can generate ASCII text files.

JTerm Software:
- Use your IBM compatible to communicate with the **SBC65V1B**, "peek" and "poke" within the SBC's memory and registers, download and run your assembly programs and upload data from your SBC.

Documentation

Now that you have an idea of what is possible with this system, we will now discuss some of the things you will need to know to make the best use of **µCEL**. I have tried to make these explanations and this manual as painless as I know how.

Conventions and Terms

Signal Naming

Before going into the details of the circuit operation, let's define a few conventions and terms. All signals in this discussion (except clock signals) will follow the following format:

Signal Name . Assertion Level. The signal name describes the function of the signal (what it does.) The assertion level describes what polarity the signal will have when it performs that function.

For example,

Reset.L performs the function of reset and will be asserted (**L**)OW when it performs that function.
Read.H would perform a read function and would be asserted (**H**)IGH.

Numbering System

The following number representations will be used:

- **Binary** base 2. Example 0011B
- **Decimal** base 10. Example 65535Z (the Assembler uses suffix "Z")
- **Hexadecimal** base 16. Example 0FFFFH (use a 0 before a hex number beginning with A-F)
- If no suffix is given, use the context of the text as your guide. The **JAsm** assembler's default is hex, so the suffix "H" is used *only within **this** text* and **not** within your source code text.

Overview of the remainder of this Book

In **Chapters 3** through **Chapter 11** we are going to cover the hardware design and assembly of the **µCEL SBC65V1B** CPU I/O Module.

In **Chapters 12 - 15** we will discuss the **µCEL** monitor and development software. This includes a monitor listing.

To give you a flavor of embedded design, the section following **Chapter 1?** will discuss a simple capacitance meter interface.

Also included is a hardware netlist of the PCB used in the **SBC65V1B** computer.

Chapter 3

Block Diagram

The **SBC65V1B** hard ware can be divided into the following sections:

Refer to the block diagram of **Fig 1**

1. **Clock**
2. **Reset**
3. **CPU & buffers**
4. **Address decoding**
5. **Read/write logic**
6. **Memory**
7. **I/O and H-Buss**
8. **Eprom monitor.**

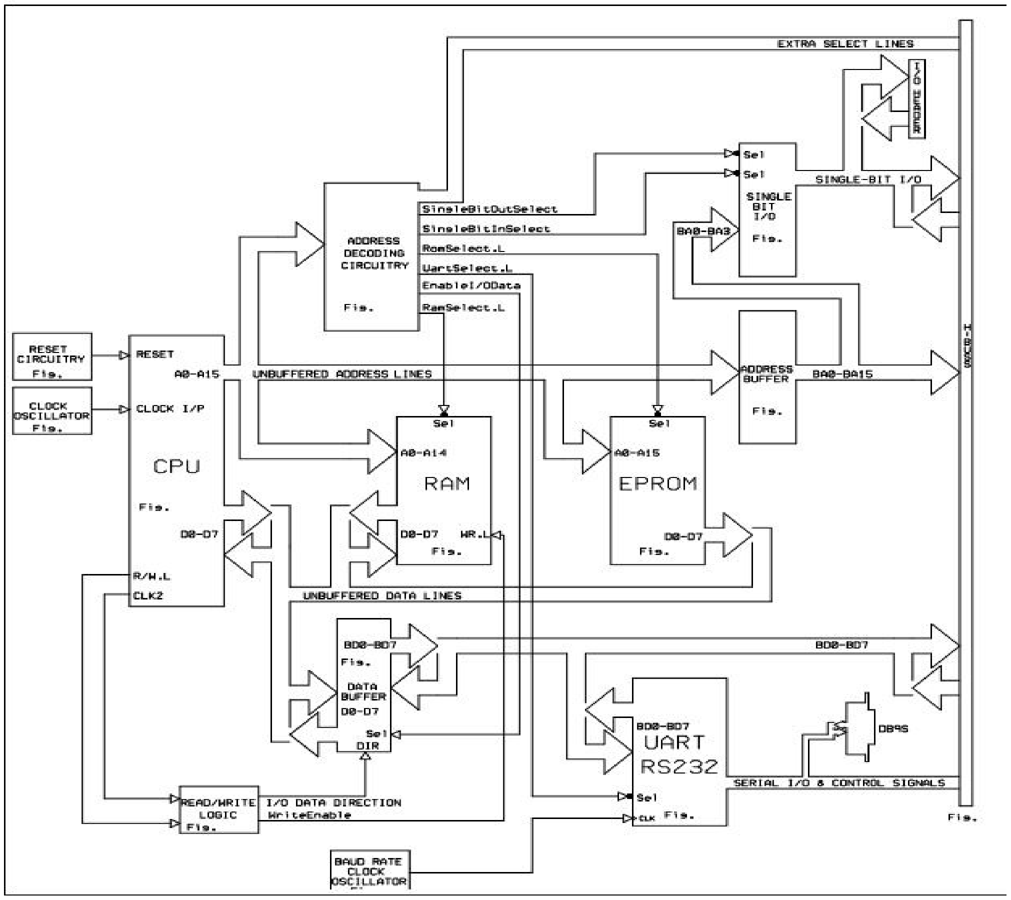


Figure 1 SBC Block Diagram

.

General Observations

CPU Clock
- Synchronizes the entire system

Baudrate Clock
- Synchronizes the uart (RS232).

Reset
- Starts system at a predictable point

CPU
- Directs the reading and writing of memory and I/O in sync with the clock.

Buffers
- Allow more TTL input lines to be driven by the **65C02 CPU**.

Address Decoding
- Provides the mechanism for selecting memory and I/O at the proper time.

Read/write Logic
- Sets the I/O data direction and proper read/write timing for the static ram.

Memory
- Provides a medium to store programs and data.

I/O
- A means by which you can communicate with the "outside world"

H-Buss
- Provides a convenient mechanical interface with the **SBC65V1B** module and your project.

Eprom Monitor
- Contains a program that is run every time you turn on power or press the reset button. This monitor also provides the mechanism for some elementary troubleshooting and also allows you to install your own program that will run after reset.
- Notice that devices such as ram, EPROM, get most address and data signals and a few control signals. Other devices such as buffers and I/O get smaller *combinations* of address, data, and control signals.
- Also notice that most of the timing is "built-in" to the address decoding. In other words, you won't get data from a device unless it is selected. You won't select a device unless the program you write *requests* that device by providing the *instructions* to do so.
- Also notice that memory is connected to the unbuffered CPU address and data lines, I/O to the buffered address and data lines.

Chapter 4

Clock Circuitry

This chapter will discuss the clock circuitry.

CLOCK

The 65C02 microprocessor, which will be referred to as the CPU (Central Processing Unit), requires a clock signal.

This clock signal is used to synchronize or pace all the operations of the CPU and its I/O. The 1 MHz clock **Fig 2.** is buffered by the 74LS14 inverter, U1f and presented to pin 37 of the CPU. The CPU in turn, outputs the clock as phase 1 and phase 2 on U12 pin 3 and pin 39 respectively.

Phase 2 is labeled CLK2. CLK2 is an important signal because all input and output operations of the CPU are referenced to this signal.

The SN2661 uart chip also requires a clock signal. Although a 5.068 MHz frequency is used to provide the uart clock, a 5MHz clock oscillator seems to work just as well.

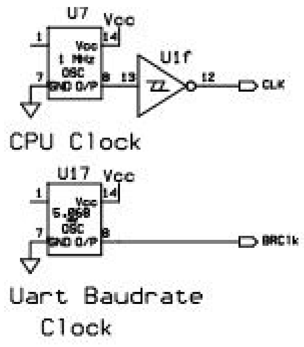


Figure 2 -- CPU and Uart clocks.

Using Other Clock Speeds

There are 65C02 parts that can operate at a clock rate of up to 8 MHz. The **SBC65V1B** was designed to operate at 1 MHz. The firmware timing routines built into the **µCEL** monitor depend on a 1MHz clock, although you might get twice the resolution at 2MHz.

It is possible to *experiment* with faster clock rates, but remember that you may have to 'tweek' the monitor's timing routines. This can be accomplished by using a frequency counter hooked to a toggled I/O bit. Use this bit to determine the accuracy of your firmware loops. You can achieve the desired accuracy by inserting NOPs and changing the value of the loop constants.

Also, the *manufacturer makes no warranty on units operated at higher frequencies*.

If you experiment in this direction, here are a few more things to look out for:

1) Make sure the access times of your EPROM and RAM are fast enough. At 8 MHz the CPU cycle time is only 125 nSec and your access time is less than that!

2) You may have to use faster glue logic, such as 74F138's, to get the con troller to operate properly.

3) Make sure you test your "new" board over the range of temperature that it will be used. As indicated elswhere, **DO NOT** use the **SBC65V1B** controller for anything ***that could possibly endanger life or limb!*** It is very unwise to depend on a computer of any kind for your ultimate safety! A better idea is to create a margin of safety that was not there before you entered the picture.

Always consider what would happen when something goes wrong (it will). Would you inadvertently get poked, pushed, smashed, prodded or hurt in any way?

Have fun, **pay attention**, and **be responsible and accountable** for what you are doing -- good design enhances life! (off my soapbox)

Chapter 5

Reset

There are two devices on the **SBC65V1B** that need to start from a predictable, repeatable and known place. These devices are the CPU and the uart. The reset circuit is shown in **Figure 3.**

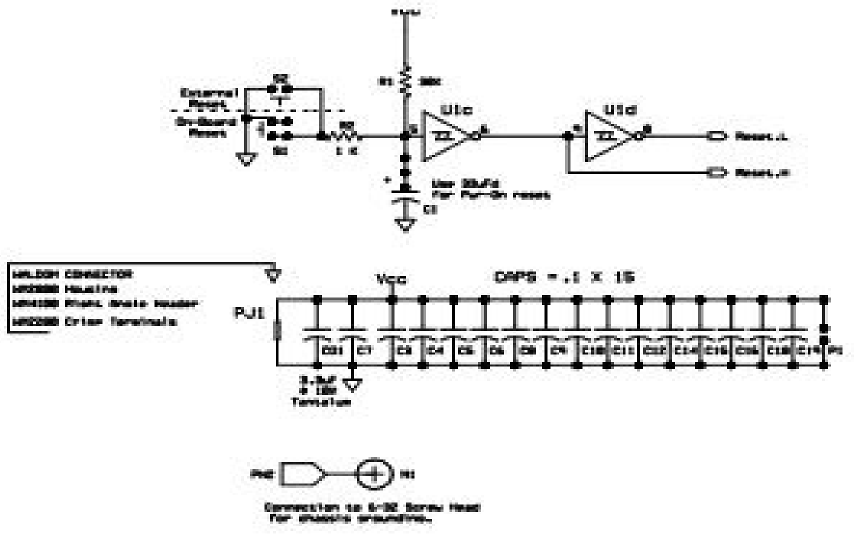


Figure 3 -- Reset circuitry, power connector and decoupling capacitors. Note chassis grounding point.

When power is first applied, or when reset button S1 (or optional extern a
LO level at the input of the TTL Schmidt inverter. When the switch is released, C1 will begin to charge through R1. The R1-C1 time constant determines how long a "**1**" or **HI** will be present at the output of U1c, and a "**0**" at the output of U1d. These signals reset the uart and the CPU respectively. When C1 has charged to approximately 2.0 V, the input to the inverter will "see" a "**1**," and the outputs U1c and U1d will revert to their resting states, "0" and "1."

When **Reset.H** asserts, it initializes the state machine inside the 2651 uart chip.

Reset.L initiates the CPU's starting point. You may have observed your personal computer going into "Na-Na-NooNoo Land" for no apparent reason. Depressing the reset button gets the CPU back to a point where it can "start over." Reset, signals the CPU to grab a special address out of the EPROM monitor and begin executing the program pointed to by that address. Special addresses such as these are called vectors and will now be discussed in more detail.

Vectors

A vector is a fixed location in the memory map of the 65C02 processor that the CPU accesses under any of the following conditions:

1) Reset button is pushed or power is turned on.
2) A non-maskable hardware interrupt (NMI) is asserted.
3) A maskable hardware interrupt (IRQ) is asserted.
4) A non-maskable software interrupt (SWI) is asserted.

Each type of vector occupies 2 bytes of the memory map, and are used to contain the address of the routine that you want to execute when any of these conditions are present. IRQ and SWI share the same vector. Only the reset vector will be discussed at this time.

Sequence:

1) Reset button is pushed.
2) CPU loads the 2 bytes at the vector locations 0FFFA and 0FFFB into its program counter.
3) CPU jumps to the routine pointed to by the vector and begins executing the code found there.

The **XaMonV4B** monitor vectors, point to locations in RAM. These RAM locations are initialized on power-up and point to reset and interrupt routines located within the monitor.

You can install your own vectors, making them point to your own routines in RAM for all interrupts. If you want to change the EPROM vectors, you must do so by burning a new monitor EPROM with those new vectors.

Chapter 6

CPU and Buffers

Each output pin on the 65C02 CPU has the capacity to drive only one TTL *unit load*. That means you can drive the inputs of one TTL or approximately two TTL-LS gates. Try to drive more loads than the CPU can handle, and you have an erratic or non-operating 65C02.

We buffer the address and data lines because we will be using them for the I/O interface. The address lines are buffered using U9 and U10. The enable lines to these 74LS244's are tied **LO**, hence the buffered address line signals are always present to the H-Buss interface J12 and J13.

U14 is a 74LS245 used to buffer the CPU's bi-directional data lines. R.L/W is used to control U14's direction line. When the direction signal is **LO**, data is read from the interface bus into the CPU. When the direction line is **HI**, data is written from the CPU out to the bus. Notice U14, pin 19 of **Figure 4**, labeled IODataEnable.L. Bus data is enabled (turned on) ONLY when this signal is asserted LO. This signal is low during I/O operations only. In **Chapter 7** you will learn how **IODataEnable.L** is created.

When you design your own interface, you will want to ensure that you don't put interface data onto the bus at the wrong time. For example, two interfaces trying to put data onto the bus at the same time would create a jamming condition called *bus contention*. Bus contention locks up the CPU and stops execution of the program. If your CPU I/O module will not boot when your interface is plugged in, this may be your problem.

For simple experiments, you can avoid data bus contention by using the **SBC65V1B**'s ready-to-use I/O select signals to drive your interface.

Another potential source of trouble is exceeding the drive limit or fanout of the data or address buffers. Consult the electrical characteristics table in your TTL data book (IIH and IIL) for the IC's you are planning to use.

The built-in address buffers can handle from 10 to 20 unit loads of their own series (such as 'LS). This means you can use an **H-Buss** buffered address line to drive up to 20 LS inputs on your interface. If you use some other logic family for your interface, be sure you have enough buffering to handle the loads encountered. For example, a 74LS00 nand gate input consumes about -.4 mA of current while the signal driving that input is low.

On the other hand, a 74S00 input consumes -2 mA, or about five times as much input current as an 'LS00 input!

Let's calculate.

- 20 74LS00 input loads @ -.4 mA ea = -8 mA.
- A 74LS00 can provide up to 8 mA output.
- A 74S00 input requires -2 mA.
- Therefore a 74LS00 could only drive 4 74S00 inputs!

- Fanout = -8mA/-2mA = 4.

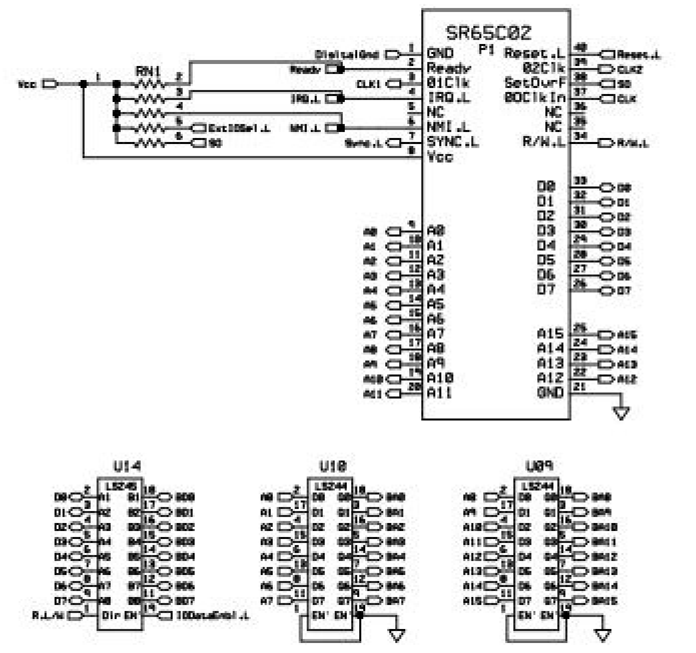


Figure 4 CPU and Address Buffering

Chapter 7

Address Decoding

Address decoding is the process of intercepting the address line signals generated by the CPU and using them to create control signals that enable memory or I/O devices. You may be asking yourself "what causes the 65C02 CPU to generate an address on these lines?" Good question! Actually, the CPU will present an address (generate signals) on its address pins each time it executes *an instruction of a program*. This can be a program in the EPROM (*firmware*), or a program you have downloaded into ram (*ramware*).We will discuss programming in more detail at a later time.

The 65C02 CPU has the capability of addressing 65,536Z different memory locations. Each of these locations has an address, just like each house in your neighborhood has an address. Instead of a house number, we represent each memory location with a 16 bit address, whose signal lines are labeled A0-A15. These 16 bits can also be represented by a four hex digit number in the range 0000H to FFFFH.

For a description of hex arithmetic see ***Radio Electronics*** May 1981 pg 46 and Aug 1984 pg 54.

The **SBC65V1B** controller uses *memory-mapped* I/O. This means you can locate input and/or output anywhere from location 0, to location 65,529Z (remember, in this text the "Z" suffix represents a decimal number). The last 6 locations of the memory map are used for vectors. Notice the vertical rectangular column of **Figure 5**. This memory map (sometimes referred to as an address map) is used to graphically represent the partitioning of RAM, I/O and EPROM in the CPU's address space.

Hexadecimal numbers will be used to describe addresses that the CPU can reference (access).

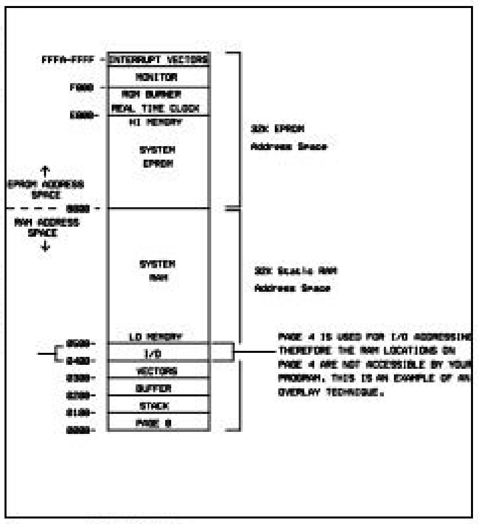
Note that location 0000H is at the bottom of the map and location FFFFH is at the top.

"Gotcha:" some data books represent the low-address portion of the address map at the top. This format is probably used to mimic a printed program listing .

The entire map consists of FFH (256Z) pages, each page containing FFH (256Z) locations. 256 X 256 = 65536, or 0 to 65535.

Memory Map

de-activate the RAM *and* the



EPROM.

Figure 5 System Memory Map

Please refer to the memory map in **Figure. 5**. Every 256 bytes of system memory is known as a ***page***.

The first chunk of 256Z locations is called *page 0*. The designers of the 65C02 created a special way for the CPU to access up to 128 16-bit registers, the ***zero-page addressing mode***.

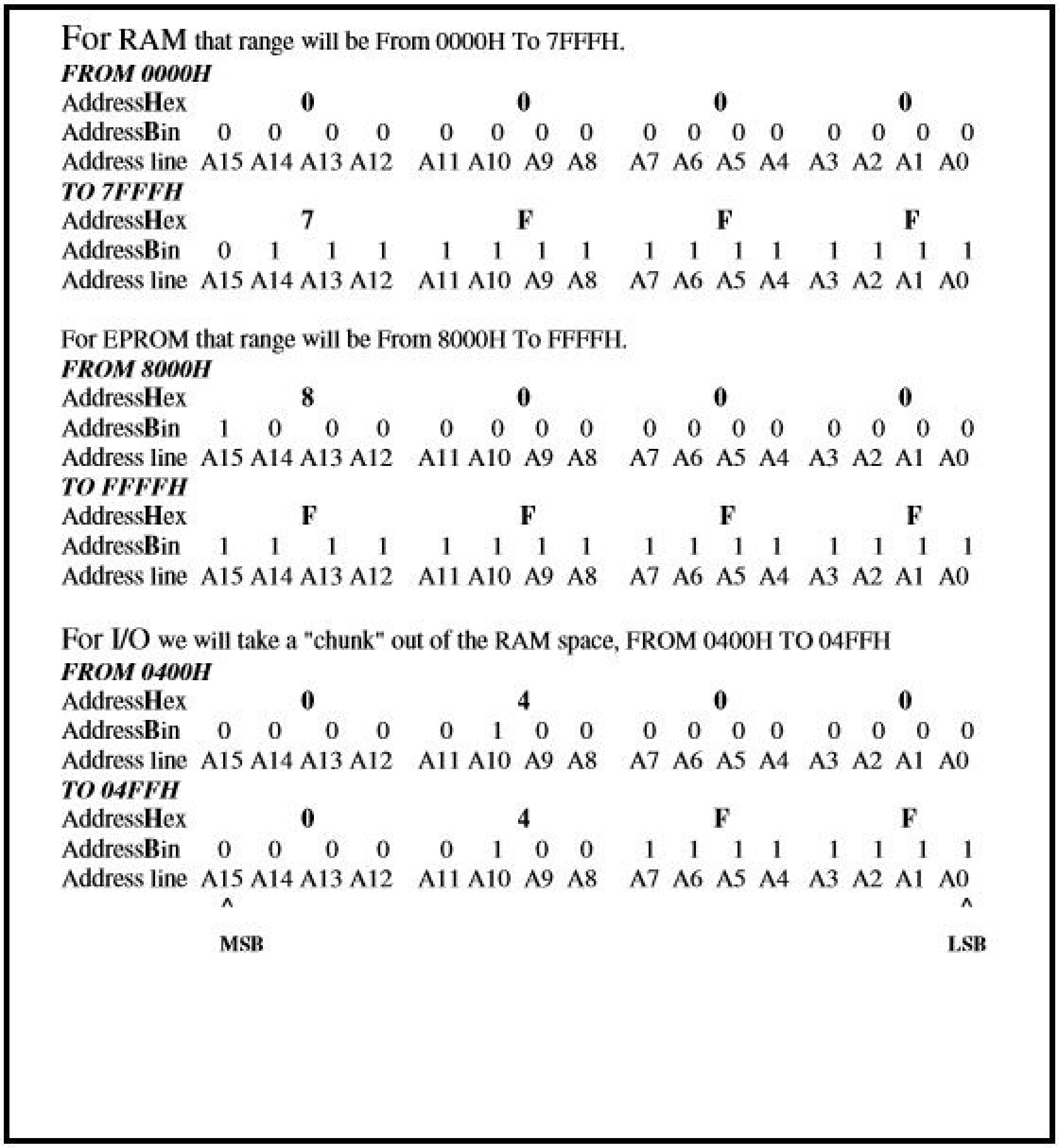
The next page, *page 1*, is also important because of its connection to the hardware. This page is used by the CPU for a stack. A stack is a mechanism that allows the CPU to use part of the sbc's memory as if it were part of the CPU itself. You will come to appreciate the stack when you get into programming.

1) Part of page 2 is used to hold system monitor variables and pointers.
2) All but the first 32 bytes of Page 3 are open for use as *buffer* space.
3) I/O (Input/output) for the **SBC65V1B** is located on page 4.
4) Program or data memory (RAM) begins at Page 5 and continues to location 7FFFH.
5) Eprom starts at location 8000H and continues to the top of the memory map FFFFH.

Another way of looking at this map is 32,768Z locations of RAM memory with Page 4 "stolen" or "sacrificed" for I/O which is not used to hold programs. What remains is 32,768Z locations used for EPROM memory.

We have almost half the memory space used for RAM and the other half as EPROM. Now we will now discuss the actual address decoder design.

Table 1 Address Table



For RAM that range will be From 0000H To 7FFFH.

FROM 0000H

AddressHex		**0**				**0**				**0**				**0**		
AddressBin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address line	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

TO 7FFFH

AddressHex		**7**				**F**				**F**				**F**		
AddressBin	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Address line	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

For EPROM that range will be From 8000H To FFFFH.

FROM 8000H

AddressHex		**8**				**0**				**0**				**0**		
AddressBin	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Address line	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

TO FFFFH

AddressHex		**F**				**F**				**F**				**F**		
AddressBin	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Address line	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

For I/O we will take a "chunk" out of the RAM space, FROM 0400H TO 04FFH

FROM 0400H

AddressHex		**0**				**4**				**0**				**0**		
AddressBin	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Address line	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

TO 04FFH

AddressHex		**0**				**4**				**F**				**F**		
AddressBin	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1
Address line	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
	^															^
	MSB															**LSB**

Decoder Design

Please study **Table 1**. Notice that each address line is actually a signal that can be either **HI** or **LO**. These binary values are used for the inputs of the address decoding circuitry in **Figure 6**. You can *statically* test an address decoder circuit without a CPU, by presenting Vcc and ground (**HI** or **LO)** to the appropriate pins of your design!

Step 1 of this address decoder design process is to decide where RAM, EPROM, and I/O are to reside in the CPU's memory map. These devices could be placed anywhere within the map, but there are **SOME** CPU requirements to be satisfied: Again, page 0 and page 1 need to be in RAM. The top 6 locations of Page # FF are used for reset and interrupt VECTORS and should reside in EPROM.

The memory map of **Figure 5** represents the memory layout we will design.

Looking at this memory map we can get a picture of what we want our decoding hardware to do. We want to create 3 control signals. Only one of the following signals will be asserted per CPU instruction.

(1) will select (turn on) the RAM when the CPU executes instructions accessing addresses in the range 0000-03FFH and 0500-7FFFH.
(2) will select the EPROM when the CPU references the 8000-FFFFH range.
(3) will simultaneously select the Input/Output circuits when in the address range of 0400-04FFH and *inhibit* (turn off) the RAM and EPROM.

For example, an instruction to load the CPU's accumulator at location 1000H "LDA 1000" would do the following:

1) The CPU would form the address 1000H on its address lines A0-A15.
2) These address lines are connected to the address decoding circuitry, generating a *memory select* control signal.
3) This signal would cause the RAM to be selected, hence only RAM program/data would be available to the CPU during that particular instruction cycle.

Summarizing by example:

- LDA 0500 instruction sequence would activate the RAM,
- LDA 8000 would activate the EPROM,
- LDA 0400 would activate the I/O section and simultaneously de-activate the RAM and the EPROM.

Step 2 envolves visualizing the address bits A0-A15 (signal lines). Referring to **Table 1**, note that the most significant bit (MSB), A15 is on the left, and A0 (LSB) is on the right. In this way we can represent the address in both Hex and Binary. Since we are dealing with a *RANGE* of addresses, let's start by representing the first and last address of each range.

Step 3, we make observations about our address ranges, and use the bit levels (HI or LO) present at each address line in those ranges to generate the select signals we want. **Figure 6** shows the schematic of the **SBC65V1B** decode circuitry. First we will focus on the control signals for the RAM and EPROM, then the signals for the I/O circuits.

Here are some observations that can be made from **Table 1** above:

A15 is ***always*** **LO** ("0") when RAM is being referenced.
A15 is ***always*** **HI** ("1") when EPROM is being referenced.
A8-A15 do not change when I/O is being referenced.

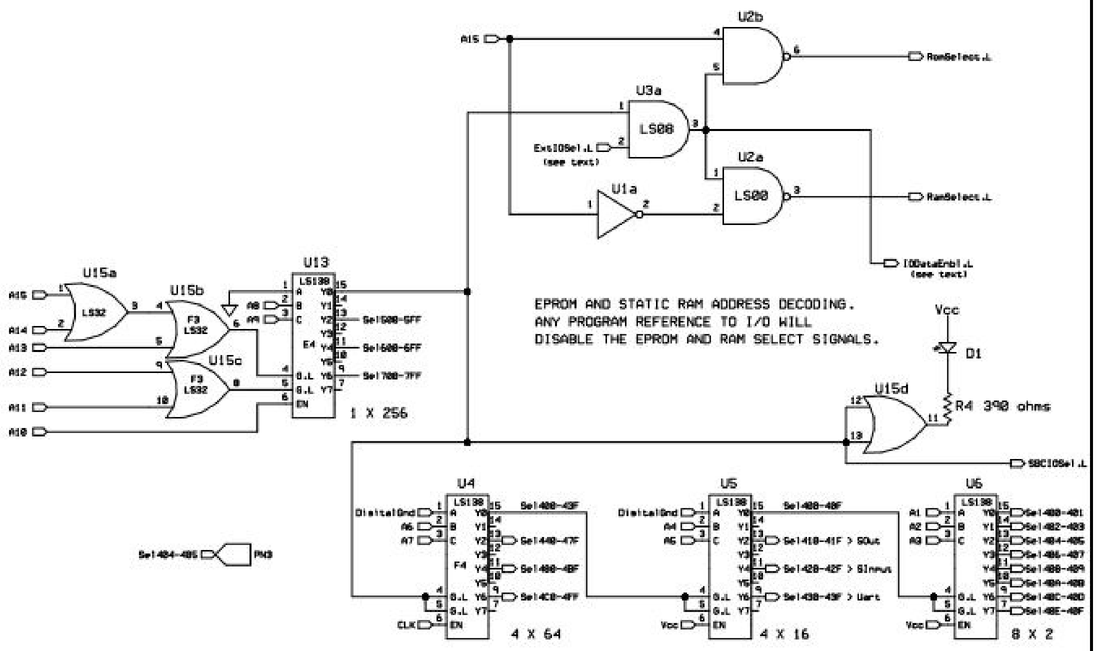


Figure 6 Decoding CPU Addresses

If we didn't have any I/O, our decoding task would be easy. We'd only use A15 to control the RAM's chip select line, and use A15 (inverted) to control the EPROM's select line.

A controller without I/O would be boring, so we modified the "A15" idea to include a provision for disabling both RAM and EPROM during I/O operations. U15 and U13 are used to intercept the upper 8 (A8-A15) address lines from the CPU and create a SBCIOSel.L signal. This signal is buffered by U15d AND gate wired as an inverter. The LED will light any time the I/O 'chunk' of memory space is accessed. This will provide us with an indication of when our program is accessing I/O, and let us know if our CPU is running.

We can use **SBCIOSel.L** to do two things:

1. We can use this signal to inhibit RAM and EPROM when I/O is selected. We accomplish this using U3a, U2a and U2b. The two inputs to U3a are normally HI. Any access made to either internal or external I/O will cause the output of U3a to go **LO**. This signal is named **IODataEnbl.L**. This will force U2b and the EPROM from being selected.

IODataEnbl.L is also used by the I/O data buffer U14 to enable data between the CPU and the I/O circuitry. No data goes to or from your interface without this signal.

The U3a pin2 input labeled **ExtIOSel.L** can be a nice feature. This input allows you to disable RAM and EPROM with a signal from **YOUR** interface. So if you don't like the memory map on the **SBC65V1B** , you can layout ***YOUR OWN!*** You do this by AND'ing the select lines from your interface, (yes, from your *own* decoding circuitry) and presenting the resulting signal to **ExtIOSel.L**. A typical use for this feature is to relocate I/O to a different part of the map, for example I/O to C000.

` 2. We can sub-divide the address lines within the I/O range 0400H to 04FFH to provide smaller I/O 'chunks' that we will later use to control different parts of the **SBC65V1B** controller. We use the 74LS138 decoder chips to do this. The 'LS138 has 6 inputs and 8 outputs. 3 of the inputs are called enable lines, and the other 3 inputs are select lines. The signal at pins 4 and 5 have to be asserted LO and pin 6 asserted HI in order for the chip's output to be enabled (on=asserted LO). When the chip is enabled, the signals at pins 1,2 & 3 deter
mine which output will be on. If 000 is presented to pins 1,2 & 3 , output Y0 will be asserted LO. A 111 presented to those same lines will cause output Y7 to be asserted LO.

The chip U4 is unique in that we use our **SBCIOSel.L** control signal and the system clock to gate this LS138 on. This insures that data from your interface circuits is presented to the CPU **ONLY** when the CPU is ready for your interface's data.

In summary, the upper 8 address lines give us our 'yes-it-is-I/O' coarse select. That address range is divided into 4 chunks of 64Z addresses. Three of these chunks are available to be used for *your interface circuits*. The next stage divides the remaining chunk of 64Z into 4 chunks of 16Z, and the last stage takes one of the chunks of 16Z and further divides that into 8 chunks of 2, which are available to you.

You may be wondering why these select lines are divided the way they are. For example, the Uart serial chip has several registers that we will want to access. If we were planning on having no other I/O, we could use the **I/OSelect.L** signal and our decoding chore would be complete. No fun!! We would be 'wasting' too much of the I/O space. So instead we use a 1 X 16Z select line. Therefore we have sixteen addresses to use to access the registers within the uart.

Remember, ROM or RAM is selected when there is no I/O being accessed, as determined by address line A15.

Another way to envision these locations is to compare them to a post office. Imagine a post office with 65,529 post office boxes numbered 0 to 65529. The post office box *number* would be the address and the *contents* of that box would be the data (the mail).

Chapter 8

Read Write Logic

There are two fundamental CPU operations, reading and writing.

The CPU is either reading or writing data or reading an instruction. Please refer to **Figure 7.**

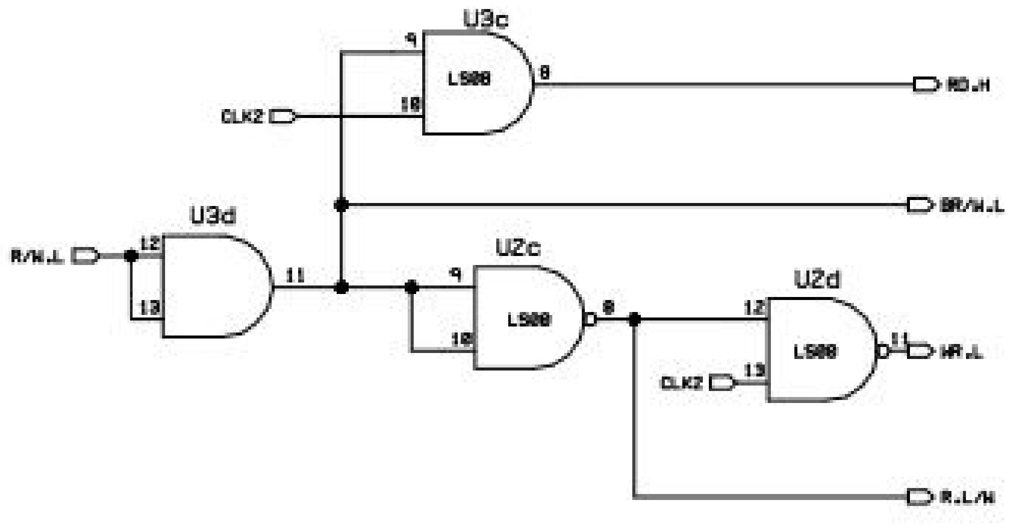


Figure 7 Read/Write Logic

When the CPU executes a read instruction like

LDA 7000

the signal R/W.L goes HI, indicating a 'read' instruction. CPU instructions such as LDX, LDY and PLA all generate a RD.H signal at U3c, pin 8. When the CPU executes a write instruction like

STA 7000

'write' instruction. CPU instructions such as STX, STY and PHA all generate a WR.L signal at U2d, pin 11.

It may serve you to recognize that read and write is relative to your reference point. For this discussion, ***the reference point is the CPU***.

A read means instructions or data goes *into* the CPU and a write means that data goes OUT *from* the CPU. Later we will describe the details of the I/O hardware when we look at the software that drives the hardware, hence the term, software ***driver***.

The R/W.L signal coming from the CPU U12, pin 34 is buffered by an AND gate, U3d. The output of U3d pin 11 is also AND'ed with CLK2 from the CPU pin 39 at U3c to create a read control signal, RD.H at the output of U3c, pin 8. Buffered R/W.L is inverted by U2c, and presented as an input to U2d. This signal AND'ed with CLK2 creates a WR.L signal used by the static RAM during its write operations.

BR/W.L is available at the H-Buss J12 pin 16 as is R.L/W on J13 pin 25.

R.L/W is used by the uart U18, pin 13 and for data direction control at U14 pin 1.

Chapter 9
Memory

The memory interface was designed so that you could select from many sizes of static RAM and EPROM. There are four jumpers for the RAM, and six for the EPROM. The table above each memory component describes how the jumpers are set for different types of memory. '**O**' stands for open (no jumper) '**S**' for shorted (jumper present).
For example, Let's say we want to use an 8K X 8 static RAM; a 6264. Reading from the chart below (**Figure 8**) you can see that J4 and J11 get a jumper and J5 and J10 do not. Jumpering the EPROM is similar, only there are more sizes of memory to select from.

Jumpers J2 and J3 are only used with the 64K X 8 EPROM as a bank select. Shortly we will show you how you can connect one of the output bits from the uart to control what 32K bank you use.

If you do not have any of the plastic block-style jumpers, you can simply wire wrap the header pins together to make the short. Leave them alone for the open.

Bank Selecting the Eprom

Bank selection makes it possible to use a 64K X 8 Eprom for U8. The **SBC65V1B** can only access 32K of Eprom at a time. Therefore, to use a 64K Eprom we must divide it into two 32K X 8 chunks.

Let me remind you that your CPU may stop dead in its tracks if you fail to initialize all the key variables in your program, or you call a monitor routine before having first copied the monitor into each 32K portion of the EPROM. What I'm saying is that using bank switching in order to have a very long program is possible, but takes some thought.

A simpler approach is to consider the low 32K as "application 1" and the upper 32K as "application 2."

First copy the monitor into F1DE to FFFF of the Eprom, using an Eprom burner . Don't forget to include the 6 bytes of interrupt vectors at FFFA-FFFF.

Next, copy the monitor's object code again, at Eprom locations 7000 to 7FFF. The 6 interrupt vectors go into locations 7FFA-7FFF.

Now by tying U18 pin 23's signal (RTS) PN4 to memory jumper J1, PN1 with a wire wrap wire, you can use the Monitor routines' **SetRTSBitLo** and **SetRTSBitHi** to switch between applications via **SOFTWARE** (actually your firmware or ramware)!

Maybe you can think of some clever new use for this feature.

	J6	J7	J8	J9	J2	J3	
2816	O	H	O	S			
2716	O	O	O	S			
2732	S	O	O	S			
2764	S	O	S	O			
27128	S	O	S	O			
27256	S	O	S	O			
27512	S	O	S	O	O	S	High 32K
27512	S	O	S	O	S	O	Low 32K

S=SHORT O=OPEN H=HIGH

	J4	J5	J10	J11
6116	O	S	S	O
6264	S	O	O	S
43256	S	O	O	S

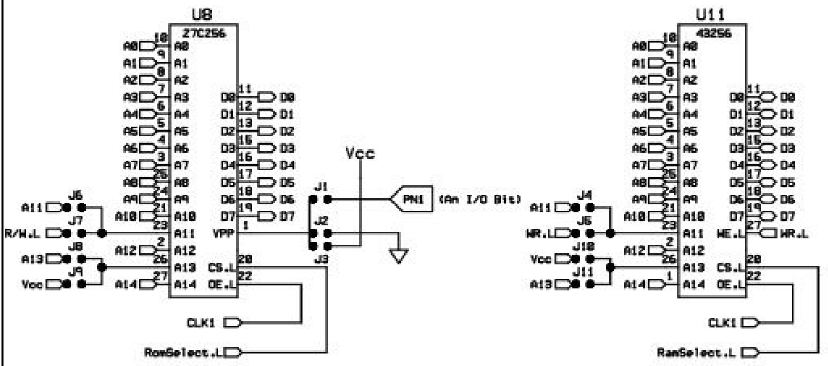


Figure 8 Ram and EPROM Bank Selection

Chapter 10

Input/Output

The **µCEL SBC65V1B** comes with two types of Input/Output (I/O), digital and serial.

Digital I/O

There are 16 bits of digital I/O, eight bits of input and eight bits of output. Each of these I/O bits has an unique address so that your source code can be more readable. For example, you could label an input bit TableLimitSwitch1 or an output bit could be labeled StepperMotorOff. The capacitance meter interface included in this documentation has an example of I/O bit utilization.

Refer to **Fig 9**. The 8 bits of input are provided by U16, an 74LS251. Each of the input bits is pulled-up HI by RN2. You address these inputs 0420 for bit 0 and 0427 for bit 8. Notice that the data bit goes to BD7. You can use the following sequence of instructions to read bit 0:

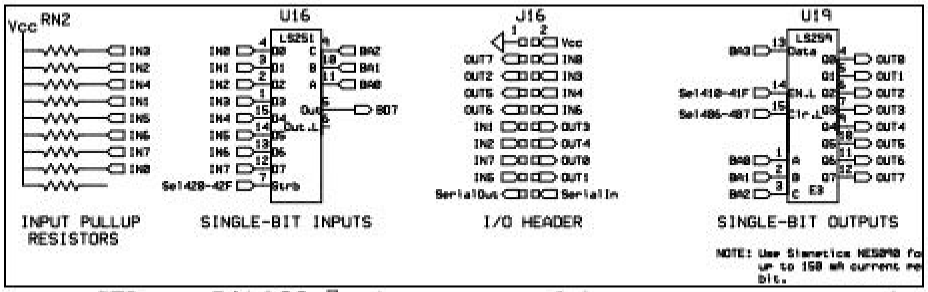


Figure 9 Digital I/O

```
CONSTant section
IgnitionSwitch ADR 0420.
CODE SECTION
          LDA  IgnitionSwitch
          BMI  testOne
          BPL  testTwo .
```

The 8 output bits are provided by U19, an 74LS259 addressable latch. If you want more driving current, a Signetics NE5090 can sink up to 150 ma per bit. This is sufficient current to drive a small relay.

There are two address ranges for this device. To turn bits 0-7 off, use 0410-0417. To turn bits on, use 0418-041F. Notice that your program can use offsets:

```
CONSTant section
BitOn   ADR 0410
BitOff  ADR 0418
CODE SECTION
          STA  BitOn+1   // comment turn bit 1 on
          STA  BitOff+7  // turn bit 7 off
     A reference to 0406 or 0407 will clear all the outputs to a '0'.
```

Serial I/O

Serial communication is handled by the 2651 uart chip U18 and the MAX232 (see **Figure 10**). The MAX232 U20, is a RS232 buffer/driver chip that uses a charge pump to generate the +/-10V RS232 levels. Consequently, the only power source needed is +5V. Hardware handshaking bits are available for you, but are not yet used by **JTerm**, the terminal software.

There is one general purpose input bit on the uart which is used to provide a self-test feature on the **SBC65V1B**. This bit is connected to **J15**. See **Figure 11**.

When the jumper **J15** is shorted, the SBC will run a test program upon being reset. If the jumper is not present, the SBC will run the monitor program.

A diagram of the IBM serial connections are also provided for you in **Figure 10**. Notice that the IBM pin outs for both the 9 pin and 25 pin "D" connectors have been included.

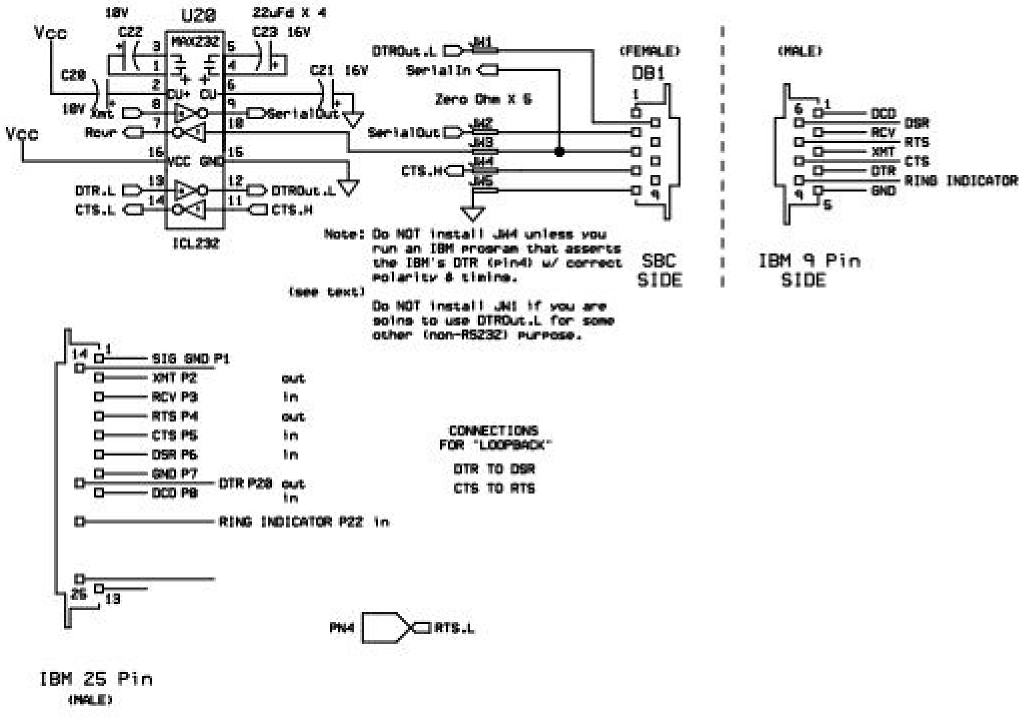


Figure 10 Serial I/O

J14 is present so that you can write a serial interrupt routine, using the uart's interrupt pin. When you connect **J14** (short it) you tie the uart's interrupt U18 pin 14 to the maskable software interrupt (**IRQ**) pin 4 of the CPU.

After you have learned to write your own programs, it will be possible to burn an EPROM containing your program. You will need an EPROM burner. Your program can replace the test program. All you will have to do to run your program (after you burn it into EPROM) is to turn on power! More will be said about this in **Chapter 12.**

*There are some excellent books on serial communications. The details of which are out of the scope of this manual. Go to your local college library for more details on this topic. The 'nitty-gritty' of the 2651 uart will be described in a future interface note.

The routines for initializing, reading and writing to the uart are provided in the monitor **XaMonV4B** described in **Chapter 12** and listed in **Chapter 13**, lines 449-517 and 544-559.

H-Buss

It's alot more fun when a single board computer provides some form of “expansion”. This allows you to design your own experiments and plug them onto the **SBC65V1B** cpu I/O module. This feature is facilitated by the ***H-Buss*** (**Figure 11)**, using "3M"-style plug strips. There are 72 positions on this bus, 36 per side. You can achieve different mechanical configurations depending on the size and polarity of the header strips you choose (see **Figure 14**). Regardless of the configuration you choose, the end result will be a nice tight rectangular 'block' of circuitry.

Another option is to lay a common plugboard breadboard on top of the SBC module, and make interface connections using #22 solid (.025 dia) hook up wire. You can obtain signals from the H-Buss by merely plugging a wire for each signal desired, into **J12** or **J13**.

An example of this technique is provided with the capacitance meter interface, included later in this manual.

Another I/O option available to you is the 20 pin header J16. You can use this header to get to the SBC's I/O resources. This header allows you to access Vcc and Gnd, the 16 bits of digital I/O, and serial transmit and receive. This makes it easier to connect to external relays, switches, etc.

This header can be basis for an expansion board series for the **SBC65V1B** module.

A lot more can be learned about I/O by using it! The application notes included with this manual should help you.

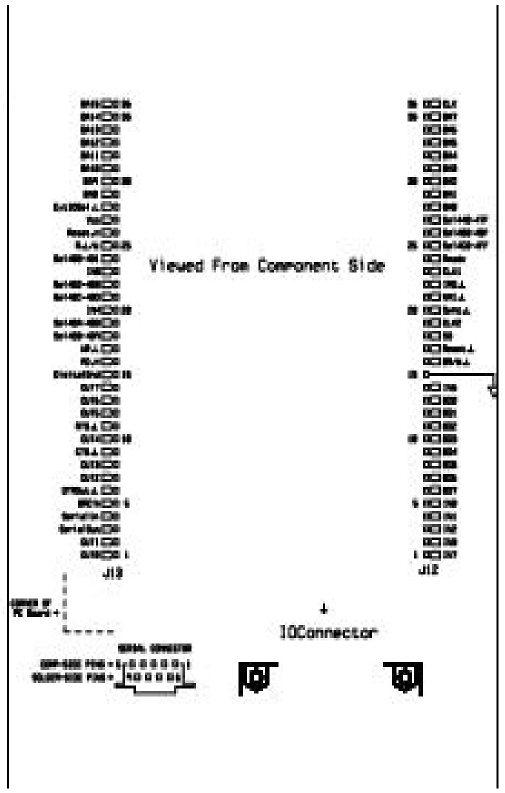


Figure 11 H-Buss

Chapter 11

Assembly and Test

If you have a rough time soldering or you have not undertaken a project of this scale, you may want to obtain the complete and socketed PC board offered on page . If you solder your own board, get some **Kester 2331** water soluble flux and Kester organic solder QQ-S-571 SN63PB37 (66 331). Another good alternative is the rosin-based Radio Shack part no. 64-009 solder.

After soldering, the 2331 flux residue comes off with warm tap water and a toothbrush. The flux residue from rosin based Radio Shack solder does not.

An important note concerning the 2331 flux, it is both corrosive and conductive, so be sure to wash both sides of the pcb, and do **NOT** use this flux system with phenolic-based pc boards or breadboards. The flux will soak into the paper base and create some **WEIRD** problems with circuit operation!

Sockets are highly recommended, especially for the first SBC you put together. All IC's on the **SBC65V1B** prototype were socketed, except for the two clock oscillators. Once you have an SBC running, the sockets provide a convenient means to test parts for later SBC projects.

Begin by soldering all the sockets. First, insert the socket onto the pcb. Then make sure that you do a visual check of the socket pins on the solder side of the PCB, ***BEFORE*** you solder the socket in. Insure that all socket pins are poking through the PCB and that you haven't invariably "folded" a pin under the socket (on the component side of the PCB). A little care at this stage of the project will save you alot of troubleshooting time later.

Next, install the push button switch, power connector and the headers. A special point about serial connector DB1; you are using the 'solder cup' connector as a surface mounted component. Carefully press the edge of the pcb between the two rows of pins on the connector, lining the connector pins up with the solder pads on the pcb. For correct orientation, the top row should have five pins (top or component side) and the bottom row should have four pins (on the solder side). When soldering, give each pin just enough heat so that the solder 'wicks' up on each side of each DB pin. If you do not want to use the 'D' connector, you may solder the wires from your serial cable directly to the holes provided for each connection. Should you take this route, be sure to make the appropriate connection for transmit, receive and ground, and also leave room for the jumper wires JW1-JW5. The SBC's 'transmit' should be connected to the host computers 'receive' and vise-versa.

The LED and remaining passive components should be soldered next. The two tantalum capacitors C1 and CD1, the LED, as well as the four electrolytics C20-C23 are *polarity sensitive*. Use the '+' marks on the silkscreen legend as your guide. The arrow on the electrolytics usually points to the '-' lead, so *pay attention* to what your doing! The 'flat' part of the LED denotes the cathode. Save yourself some trouble by installing these components correctly!   Using the lead clippings from your capacitors, form three short jumpers to be used for JW2, JW3 and JW5. DO NOT install JW1 or JW4. JW1 and JW4 are provided for the uart hardware handshaking signals. They will not be used at this time (they may come in handy later). By making a 1/4" or so loop on JW5, you can have a nice place to clip a logic lead or alligator clip. Use the '-' end of C20 for the Vcc connection to your logic probe.

Now install the **H-Buss** headers J12 and J13. You have several possibilities. here. You can solder these connectors either from the component or solder sides. If you mount these connectors on the solder side of the PCB (soldering on the component side), you can use the **SBC65V1B** as a module that can be plugged onto another printed circuit board. If you mount the connectors on the component side (soldering on the solder side), you can rest a 3M-type plugboard on top of the SBC for the breadboard experiment that will follow.

Once the soldering is complete, and the pcb is washed and dried (if you used the 2331 flux system) you can now plug the IC's into their sockets. There are a couple of "gotchas" to look out for. Notice that pin 1 for each IC does not face in the same direction or orientation. The main culprits are the RAM and EPROM. There are two clues you can use to insure proper insertion:

1) the 'U-hump' on the silkscreen and
2) the square pc board pad on each IC denoting pin one.

The IC manufacturers either put a 'notch' on the plastic at the pin-one-end of the IC or a 'dot' next to pin one. Also, if you are using the 2K RAM or EPROM, always plug the IC such that the ground (pin 12) is connected. The programming jumpers J1-J11 take care of the remaining connections. Study the pc board layout silkscreen (Found in the **Appendix**) if you have any doubts.

Cables

After your SBC is completed you still have some work to do. You need to buy or make two cables. One supplies Vcc and Gnd and the other connects the serial communication port to talk to your PC or dumb-terminal. A standard 9-pin monitor extention cable works well for the serial connection to a laptop and some AT's. If your connectors do not match up, it may be possible to buy cables with the required ends and use gender changers to finish the fit. The cable needs to have a male connection on SBC end, and a female connection on the PC end. If you make your own, you need some cable that has at least 3 conductors. Five conductors are required if you also want to include the uart handshaking signals.   The power cable is a little tricker. It takes some patience and care to successfully crimp the Waldom terminal pins to the wire, and then insert the pin into the corresponding terminal housing. The 'lip' of the pin fits into the slot on the housing. Remember to watch your polarity's so that Vcc and Ground are not crossed. Bad connections makes *fried* chips!

Check Out:

There are a few things to check out before powering up your SBC. Use an ohmeter or a low-power continuity checker to insure there is no short between Vcc and ground.

Next, insure that all the polarity sensitive components have been installed correctly. Check your power connector to insure that Vcc and ground are headed where they belong. +5V and GND is marked on the silkscreen of the PC board on the component side. Look at each IC and make sure pin 1 is oriented in the socket correctly.

One common problem is created by bending an IC pin upon insertion into the socket. This bent 'horses leg' usually will either poke out of the socket edge or 'wing' out. At other times you may have to lift the IC slightly with a screwdriver blade in order to spot the offending bent pin. The last step is to install a shorting jumper on J15. J15 signals the SBC to run a test program upon power-up.

Once the initial visual check out is complete, you may power up the SBC. If nothing smokes, that's a good sign! If the SBC is operating properly, the LED will flash once every two second or so .

Operation

Hook up the serial cable. The female end of the cable will plug into your PC's 9 pin serial connector, the male end into the **SBC65BV1B** female connector. Power up the SBC and run **JTerm**, the terminal software on the PC. If you have J15 shorted, you should see a '1' printing to the PC's screen with every flash of the SBC's LED.

Troubleshooting

A voltmeter and a logic probe are indispensable for fault-finding. Most of the suggested checks involve the use of the logic probe. Read your logic probe manual for suggestions on how to enterpret what you are seeing on the logic probe's LEDS.

First check the power and ground connections for all chips with a voltmeter (+/-.5V.)

Check every pin for 5v and ground using a logic probe.

Press the reset button S1 and check the reset signals for correct action. Pin 40 of the CPU is normally Hi, and goes Lo during reset. P 21 of uart U18 is normally Lo and goes Hi during reset.

Check the clock signals. Insure clock is present everywhere it is supposed to be, and that it is the correct frequency. If you do not have a scope or frequency counter, you will just have to trust what is "on the can."

Check the enabling signals on the decoding circuitry and buffers for proper polarity.

Check the R/W.L lines for activity.

Check the address and data line for activity. If no activity is present with a given signal, depress the reset button while monitoring the "dead" signal. If your 'pulse' LED on the logic probe doesn't at least blink, that signal line has a problem.

If the SBC is running a program (even a bad one), the address and data lines will be very busy. You can easily determine if each signal of the address and data buffers are correct. For example, if you are looking at signal line A0, if the input-side of the buffer causes your probe's pulse LED to blink, but the output of the buffer does not, you have a problem with the buffer. If there is no activity on any of the lines you may have to go to a power-off netlist ohm check.

With the CPU and memory removed, and power off, apply a logic '1' to each of the 16 address bits at the CPU socket. Check to see that only the indicated bit is asserted HI by monitoring any convenient point at another socket with the same signal (like A0). Now use a logic '0' and perform the same test. This technique works equally well with the data lines. If you make more than one of these SBC's, you can use the extra one to drive each signal of the device under test (DUT) under program control. You can make your own automated test equipment (ATE)! When you learn how to interface your **SBC65V1B** this idea should become more apparent to you.

Using the Netlist for Trouble Shooting

A net list containing every connection on the **SBC65V1B** can be found at the end of this chapter.

The signal names are itemized on the left of the page, and the connections follow to the right.

This list can be handy when you suspect a particular line has gone awry. If you have wire wrapped your SBC, you can use the netlist to check your connections. Otherwise the net list makes a convenient way to trace signals on the SBC with your logic probe.

Common Errors

Here is a partial list of some of the things that can go wrong:

- solder blob shorting two or more pins
- incorrect orientation of IC in socket
- polarity-sensitive part installed incorrectly (such as the LED)
- socket pin bent before socket is soldered onto pc board, thus creating an open between the IC and the pc board
- IC pin bent while inserting into socket
- wrong IC installed in a given socket location
- electrically or electrostatically damaged IC
- jumpers J1-J12 incorrectly programmed or missing
- 6502 used in place of a 65C02 (use only a 65C02)
- CPU reset or interrupt vectors incorrectly burned into EPROM
- clock oscillator wrong frequency
- poor solder connection (usually cold)
- wrong logic family of TTL part used (start with TTL-LS; once you have your SBC working, then you can experiment with other families)

Hardware vs software implementations

You may have heard the terminology "implemented in hardware" or "implemented in software." What this means is that there are many applications that can be realized by using either hardware (electronics parts) or software (code).

For example, say you wanted to generate a pulse every second. A hardware implementation might use a 555 timer wired as an astable multivibrator and 'tweaked' until it outputs the desired pulse. A software implementation might consist of a software loop that calls a subroutine named '**MilliSecDelay**' 1000 times, then turns an output bit on, then off via software.

Yes, you still use some hardware (the output bit) but you've replaced all those temperature sensitive and space consuming circuits with a few lines of a ***program***. The decision as to which approach or combination of approaches to use is up to you.

If you don't give yourself enough hardware on your interface project, the software might be overly difficult to write, or your response time may be too slow. If you use more hardware, your program may be easier to write (*if you think ahead* in terms of **BOTH** hardware & software), but your project might cost more than it needs to, or take up too much room on your pc board.

Concerning *cost of implementation*; there is both a time and money cost to every circuit you build. It is not wise to write a thousand lines of code in order to replace a single IC when you are only building 1 or 2 pcbs. On the other hand, the picture changes when the numbers get into the hundreds and thousands.

Make the best decision you can! You may find it fun to play with the extremes on this issue with a few simple projects, and get a 'feel' for the versatility you can achieve by using both hardware and software creativity.

Cross Assembler

By using the 65C02 instruction set and a lot of manual 'elbow grease,' it is possible to hand -write a sequence of instructions to make your project "do something."

This can be useful in getting you down the learning curve, but a hand programmer's life gets more difficult as the programs get longer.

The assembler "**JAsm**" mentioned in **Chapter 14** runs on IBM PC/AT compatibles. It generates 6502/65C02 code. This code will not run on the PC (wrong processor) hence the term Cross-assembler. The **SBC65V1B**'s CPU uses instructions in the form of HEX codes that are one to three bytes long.
For example, store the accumulator
STA
is HEX 85 in the page zero addressing mode.

Keeping track of all these codes can be a real pain, so instead we use the cross-assembler to generate the codes. **JAsm** also allows you to define memory locations and subroutines by giving them identifier and label names. The object code can be output in both binary and ascii file formats. A more complete description of the use of this assembler can be found in **Chapter 14**.

If you are using a different controller/computer than the **µCEL SBC65V1B**, you can still generate code for your target using the **JAsm** cross assembler, providing of course that the target processor is a 6502 derivative.

IBM serial interface

The cross-assembler will work fine on an PC without a serial port, but you won't be able to download your assembled code into the **SBC65V1B** using **JTerm**.

To enter a program using a dumb terminal, you will have to look up the 65C02 instruction codes in the `XaTestR` listing, and then hand- enter the codes using the **XaMonV4B** monitor, and hand-compute all relative jumps.

You can hand-compute relative jumps using the following:
O = Offset (adjustment)
D = Destination address
PC= Program Counter

Forward computation Example:
```
 F1F0 5A      |Pop   PHY
 F1F1 A4 E6   |      LDY   StackPointer
 F1F3 F0 04   |      BEQ   pSkip
 F1F5 C6 E6   |      DEC   StackPointer
 F1F7 A4 E6   |      LDY   StackPointer
 F1F9 B1 E4   |pskip LDA  (StackAdr),Y
```
Calculations:
```
O  =  D    - PC                                  PC  +   4
      F1F9 - F1F5 = 04   Therefore the BEQ instruction is  F1F3 F0 04
```

Backward computation Example:
```
F2EF 20 E4 F2 |DelaySecs JSR Delay1Sec
F2F2 CA       |          DEX
F2F3 D0 FA    |          BNE DelaySecs
F2F5 60       |          RTS
```
Calculations:
```
O  =  D+100H     - PC
     (F2EF+100H) - F2F5 Therefore the BNE instruction is  F2F3 D0 FA
```
To download a file into the **SBC65V1B**, you can either use the down load feature of **JTerm**, the terminal program included with the assembler, or you can write your own downloader. The protocol for the download is simple and mentioned in **Chapter 15**.

Optimizing power consumption

It is possible to minimize power consumption by sorting through various IC chips and determining how much current they are using. The simple adaptor in **Figure 14** can be used to accomplish this. The power hungry chips are the 2651 uart, the LS245, the LS244's, the LS138's and the EPROM. For example, the 2651 uarts varied in current consumption from 45 to 85 ma. By using a 74HCTLS245 instead of a 74LS245, you can reduce the current consumed by about 55 ma! Not all TTL families will work with this design.

A good approach is to get your board working using TTL-LS parts, and then begin experimenting with HCTLS or other TTL family parts.

Why would you even be interested in current usage? There is probably no reason if you are using a dc power supply that is plugged into the AC line. If you are running this board off of a battery source (remember, go above 5.5 V max and you fry chips), then the less current you use, the longer your nicads, fuel cell, etc. will last. A 4 AMP/HR battery will last you 20 hours or so at 200 mA/ hr, and only 4 hours if you are drawing 1 amp.

The author's prototype uses 315 mA, equating to about 12.7 hrs of battery use time using heavy duty 'D' cell nicads. Future designs will use more CMOS parts, even though they are often harder to get.

DISCLAIMER

The **µCEL** TM **SBC65V1B** controller was **NOT** designed to be used to control anything involving life or limb. So please don't use it to control aunt Marta's kidney machine!

The space shuttle uses four computers! Three of them talk to each other and 'vote.' The outcome of the vote determines the action taken. The fourth computer is used as a spare.

This redundancy is an acknowledgment of a nasty reality; a glitch in power or a faulty instruction can send your program into some not-so-funny place and send a robot arm flying at you--or fail to shut off a critical valve, etc.

Embedded control can be extremely useful, productive and safe...but you have to be conscious and responsible for how you implement computers into your equipment or invention! The life you save may be your ***OWN***!

Limitations

Depending on your point of view, there are lots of interesting applications for this controller. It may be your ticket to replacing that Apple II you have tied up doing some simple control task. I wouldn't want to waste the **SBC65V1B** controller by doing Word processing, or any other task that is best done by a general purpose computer.

There is no watch dog timer on the **SBC65V1B**. That means if your power glitches, or your program goes off into "Na-Na-Noo-Noo Land," it will stay lost until you manually press the reset button.

If a watchdog timer were present, it would generate a CPU reset pulse after a predetermined time if your program did not send a pulse to reset the watchdog.

Mastery of the **µCEL** system should prepare you to tackle other development systems that use different processors or higher-level languages. A partial listing of other companies offering single board computers and development systems is given in the Appendix.

The **µCEL** can also be a great warm-up if you are planning to design your own system.

The following is available from JComm

SBC65V1B Printed Circuit Board with JTerm and JAsm (limited supply)

Netlist

```
BA4          J12,32 U10,14
BA5          J12,33 U10,7
BA6          J12,34 U10,12
BA7          J12,35 U10,9
BA8          J13,29  U9,18
BA9          J13,30  U9,3
BA10         J13,31  U9,16
BA11         J13,32  U9,5
BA12         J13,33  U9,14
BA13         J13,34  U9,7
BA14         J13,35  U9,12
BA15         J13,36  U9,9
BD0          J12,13 U14,18 U18,27
BD1          J12,12 U14,17 U18,28
BD2          J12,11 U14,16 U18,1
BD3          J12,10 U14,15 U18,2
BD4          J12,9  U14,14 U18,5
BD5          J12,8  U14,13 U18,6
BD6          J12,7  U14,12 U18,7
BD7          J12,6  U14,11 U16,5  U18,8
BR/W.L       J12,16  U2,9   U2,10  U3,9    U3,11
BRClk        J13,5  U17,8  U18,20
CLK          J12,36  U1,12  U4,6  U12,37
CLK1         J12,23  U8,22 U11,22 U12,3
CLK2         J12,19  U2,13  U3,10 U12,39
CTS.H        JW4,1  U20,11
CTS.L        J13,9  U18,17 U20,14
D0            U8,11 U11,11 U12,33 U14,2
D1            U8,12 U11,12 U12,32 U14,3
D2            U8,13 U11,13 U12,31 U14,4
D3            U8,15 U11,15 U12,30 U14,5
D4            U8,16 U11,16 U12,29 U14,6
D5            U8,17 U11,17 U12,28 U14,7
D6            U8,18 U11,18 U12,27 U14,8
D7            U8,19 U11,19 U12,26 U14,9
DTR.L        U18,24 U20,13
DTROut.L     J13,6  JW1,1  U20,12
DigitalGnd    C1,4   C3,2   C4,2   C5,2  C6,2   C7,2   C8,2
              C9,2  C10,2  C11,2  C12,2 C14,2  C15,2  C16,2
             C18,2  C19,2  C21,1  CD1,2  J2,2  J12,15 J13,15
             J15,1  J16,1  JW5,1   P1,1 PJ1,2   S1,1   S2,2
              U1,7   U2,7   U3,7   U4,1  U4,8   U5,1   U5,8
              U6,8   U7,7   U8,14  U9,1  U9,10  U9,19 U10,1
             U10,10 U10,19 U11,14 U12,1 U12,21 U13,1
             U13,8  U14,10 U15,7  U16,8 U17,7  U18,4
             U18,16 U19,8  U20,15
ExtIOSel.L   J13,28 RN1,5   U3,2
IN0          J12,5  J16,4  RN2,9  U16,4
IN1          J12,4  J16,11 RN2,5  U16,3
IN2          J12,3  J16,13 RN2,3  U16,2
```

```
IN3          J12,2   J16,6    RN2,2    U16,1
IN4          J13,20  J16,8    RN2,4    U16,15
IN5          J13,23  J16,17   RN2,6    U16,14
IN6          J12,14  J16,10   RN2,7    U16,13
IN7          J12,1   J16,15   RN2,8    U16,12
IODataEnbl.L  U2,1    U2,5     U3,3    U14,19
IRQ.L        J12,22  J14,2    RN1,3    U12,4
NMI.L        J12,21  RN1,4    U12,6
OUT0         J13,1   J16,16   U19,4
OUT1         J13,2   J16,18   U19,5
OUT2         J13,7   J16,5    U19,6
OUT3         J13,8   J16,12   U19,7
OUT4         J13,10  J16,14   U19,9
OUT5         J13,12  J16,7    U19,10
OUT6         J13,13  J16,9    U19,11
OUT7         J13,14  J16,3    U19,12
R.L/W        J13,25   U2,8     U2,12    U14,1    U18,13
R/W.L         J7,1    U3,12    U3,13    U12,34
RD.H         J13,16   U3,8
RTS.L        J13,11  PN4,1    U18,23
RamSelect.L   U2,3   U11,20
Rcvr         U18,3   U20,7
Ready        J12,24  RN1,2    U12,2
Reset.H      J13,26   U1,6     U1,9     U18,21
Reset.L      J12,17   U1,8    U12,40
RomSelect.L   U2,6    U8,20
RxRdy        J14,1   U18,14
SBCIOSel.L    U3,1    U4,4     U4,5     U13,15  U15,12
             U15,13
SO            12,18  RN1,6    U12,38
Sel4C0-4FF   J12,25   U4,9
Sel40A-40B   J13,19   U6,10
Sel40C-40D   J13,21   U6,9
Sel40E-40F    U6,7
Sel400-401   J13,24   U6,15
Sel402-403   J13,22   U6,14
Sel404-405   PN3,1    U6,13
Sel406-407    U6,12  U19,15
Sel408-409   J13,18   U6,11
Sel410-41F    U5,13  U19,14
Sel420-42F    U5,11  U16,7
Sel430-43F    U5,9   U18,11
Sel440-47F   J12,27   U4,13
Sel480-4BF    12,26   U4,11
SerialIn     DB1,3   J13,4    J16,20   JW3,2
SerialOut    J13,3   J16,19   JW2,1    U20,9
Sync.L       J12,20  U12,7
Vcc           C3,1    C4,1     C5,1  C6,1  C7,1  C8,1  C9,1 C10,1
             C11,1   C12,1    C14,1 C15,1 C16,1 C18,1 C19,1
             C20,2   CD1,1     D1,2  J3,2  J9,1 J10,1 J13,27
             J16,2    P1,2    PJ1,1  R1,2  R3,2
             RN1,1   RN2,1      U1,14  U2,14   U3,14
              U4,16   U5,6      U5,16  U6,6    U6,16
              U7,14   U8,28     U9,20 U10,20  U11,28
             U12,8   U13,16    U14,20 U15,14  U16,16
             U17,14  U18,26    U19,16 U20,16
WR.L          J5,1   J13,17     U2,11 U11,27
Xmt          U18,19  U20,8
node01             U4,15   U5,4       U5,5
node02             U5,15   U6,4       U6,5
node03            U13,4   U15,6
node04            U13,5   U15,8
node05             D1,1    R4,2
node06             U1,2    U2,2
node07            U15,3   U15,4
node08             R4,1   U15,11
node09             U1,13   U7,8
node10             J6,2    J7,2      U8,23
node11             J1,1    J2,1      J3,1      U8,1
node12             J1,2   PN1,1
node13             J8,2    J9,2      U8,26
node14             J4,2    J5,2     U11,23
node15            J10,2   J11,2     U11,26
node16             M1,1   PN2,1
node17             R2,1    S1,4      S2,1
node18             C1,1    C1,2      C1,3      R1,1      R2,2
U1,5
node19            C22,1   U20,1
node20            C20,1   U20,2
node21            C22,2   U20,3
node22            C21,2   U20,6
node23            C23,1   U20,4
node24            C23,2   U20,5
node25            JW3,1   U20,10
node26            DB1,4   JW4,2
node27            DB1,6   JW1,2
node28            DB1,2   JW2,2
node29            DB1,5   JW5,2
node30            J15,2    R3,1     U18,22
```

Connection by Reference Designator and Signal Names

```
C1      1 node18 2 node18 3 node18 4 DigitalGnd
C20     1 node20              2 VCC
C21     1 DigitalGnd          2 node22
C22     1 node19              2 node21
C23     1 node23              2 node24
CD1     1 Vcc                 2 DigitalGnd
D1      1 node05              2 VCC
DB1     2 node28 3 SerialIn 4 node26 5 node29 6 node27
J1      1 node11              2 node12
J2      1 node11              2 DigitalGnd
J3      1 node11              2 VCC
J4      1 A11                 2 node14
J5      1 WR.L                2 node14
J6      1 A11                 2 node10
J7      1 R/W.L               2 node10
J8      1 A13                 2 node13
J9      1 Vcc                 2 node13
J10     1 Vcc                 2 node15
J11     1 A13                 2 node15
J12     1 IN7  2 IN3  3 IN2  4 IN1  5 IN0  6 BD7   7 BD68 BD5
        9 BD4 10 BD3 11 BD2 12 BD1 13 BD0 14 IN6  15 DigitalGnd
       16 BR/W.L     17 Reset.L    18 SO  19 CLK2 20 Sync.L
       21 NMI.L      22 IRQ.L      23 CLK1        24 Ready
       25 Sel4C0-4FF 26 Sel480-4BF 27 Sel440-47F  28 BA0
       29 BA1        30 BA2        31 BA3         32 BA4
       33 BA5        34 BA6        35 BA7         36 CLK

J13     1 OUT0   2 OUT1        3 SerialOut   4 SerialIn
        5 BRClk  6 DTROut.L    7 OUT2        8 OUT3
        9 CTS.L 10 OUT4       11 RTS.L      12 OUT5
       13 OUT6  14 OUT7       15 DigitalGnd 16 RD.H
       17 WR.L  18 Sel408-409 19 Sel40A-40B 20 IN4
       21 Sel40C-40D          22 Sel402-403 23 IN5
       24 Sel400-401          25 R.L/W      26 Reset.H
       27 Vcc                 28 ExtIOSel.L 29 BA8
       30 BA9                 31 BA10       32 BA11
       33 BA12                34 BA13       35 BA14  36 BA15
J14     1 RxRdy               2 IRQ.L
J15     1 DigitalGnd          2 node30
J16     1 DigitalGnd          2 VCC   3 OUT7   4 IN0   5  OUT2   6 IN3
        7 OUT5                8 IN4   9 OUT6  10 IN6  11 IN1    12 OUT3
       13 IN2                14 OUT4 15 IN7   16 OUT0 17 IN5    18 OUT1
       19 SerialOut          20 SerialIn
JW1     1 DTROut.L            2 node27
JW2     1 SerialOut           2 node28
JW3     1 node25              2 SerialIn
JW4     1 CTS.H               2 node26
JW5     1 DigitalGnd          2 node29

M1      1 node16 P1 1 DigitalGnd 2 Vcc PJ1 1 Vcc  2 DigitalGnd
```

```
PN1      1 node1
PN2      1 node1
PN3      1 Sel404-40
PN4      1 RTS.L
R1       1 node18              2 Vcc
R2       1 node17              2 node1
R3       1 node30              2 VCC
R4       1 node08              2 node05
RN1      1 VCC  2 Ready 3 IRQ.L 4 NMI.L  5 ExtIOSel.L 6 SO
RN2      1 Vcc  2 IN3   3 IN2   4 IN4    5 IN1        6 IN5
         7 IN6  8 IN7   9 IN0
S1       1 DigitalGnd         4 node17
S2       1 node17             2 DigitalGnd
U1       1 A15                2 node06  5 node18  6 Reset.H 7 DigitalGnd
         8 Reset.L            9 Reset.H          12 CLK    13 node09
        14 VCC
U2       1 IODataEnbl.L   2 node06      3 RamSelect.L      4 A15
         5 IODataEnbl.L   6 RomSelect.L 7 DigitalGnd       8 R.L/W
         9 BR/W.L        10 BR/W.L     11 WR.L            12 R.L/W
        13 CLK2          14 VCC
U3       1 SBCIOSel.L     2 ExtIOSel.L  3 IODataEnbl.L     7 DigitalGnd
         8 RD.H           9 BR/W.L     10 CLK2            11 BR/W.L
        12 R/W.L         13 R/W.L      14 VCC
U4      1 DigitalGnd      2 A6          3 A7               4 SBCIOSel.L
        5 SBCIOSel.L      6 CLK         8 DigitalGnd       9 Sel4C0-4FF
       11 Sel480-4BF     13 Sel440-47F 15 node01          16 VCC
U5      1 DigitalGnd      2 A4          3 A5               4 node01
        5 node01          6 Vcc         8 DigitalGnd       9 Sel430-43F
       11 Sel420-42F     13 Sel410-41F 15 node02          16 VCC
U6      1 A1              2 A2          3 A3               4 node02
        5 node02          6 Vcc         7 Sel40E-40F       8 DigitalGnd
        9 Sel40C-40D     10 Sel40A-40B 11 Sel408-409      12 Sel406-407
       13 Sel404-405     14 Sel402-403 15 Sel400-401      16 VCC
U7      7 DigitalGnd      8 node09     14 VCC
U8      1 node11          2 A12         3 A7               4 A6
        5 A5              6 A4          7 A3               8 A2
        9 A1             10 A0         11 D0              12 D1
       13 D2             14 DigitalGnd 15 D3              16 D4
       17 D5             18 D6         19 D7              20 RomSelect.L
       21 A10            22 CLK1       23 node10          24 A9
       25 A8             26 node13     27 A14             28 VCC
U9      1 DigitalGnd      2 A8          3 BA9              4 A10
        5 BA11            6 A12         7 BA13             8 A14
        9 BA15           10 DigitalGnd 11 A15             12 BA14
       13 A13            14 BA12       15 A11             16 BA10
       17 A9             18 BA8        19 DigitalGnd      20 Vcc
U10     1 DigitalGnd      2 A0          3 BA1              4 A2
        5 BA3             6 A4          7 BA5              8 A6
        9 BA7            10 DigitalGnd 11 A7              12 BA6
       13 A5             14 BA4        15 A3              16 BA2
       17 A1             18 BA0        19 DigitalGnd      20 Vcc
```

```
U11      1 A14       2 A12        3 A7       4 A6       5 A5
         6 A4        7 A3         8 A2       9 A1      10 A0
        11 D0       12 D1        13 D2      14 DigitalGnd
        15 D3       16 D4        17 D5      18 D6      18 D6
        19 D7       20 RamSelect.L          21 A10     22 CLK1
        23 node14   24 A9        25 A8      26 node15 27 WR.L 28 VCC
U12     1 DigitalGnd             2 Ready    3 CLK1     4 IRQ.L
        6 NMI.L                  7 Sync.L  8 Vcc       9 A0
       10 A1                    11 A2      12 A3      13 A4
       14 A5                    15 A6      16 A7      17 A8
       18 A9                    19 A10     20 A11     21 DigitalGnd
       22 A12                   23 A13     24 A14     25 A15
       26 D7                    27 D6      28 D5      29 D4
       30 D3                    31 D2      32 D1      33 D0
       34 R/W.L                 37 CLK     38 SO      39 CLK2
       40 Reset.L
U13     1 DigitalGnd             2 A8       3 A9            4 node03
        5 node04                 6 A10      8 DigitalGnd  15 SBCIOSel.L
       16 Vcc
U14     1 R.L/W                  2 D0       3 D1            4 D2
        5 D3                     6 D4       7 D5            8 D6
        9 D7                    10 DigitalGnd             11 BD7
       12 BD6                   13 BD5                    14 BD4
       15 BD3                   16 BD2                    17 BD1
       18 BD0                   19 IODataEnbl.L           20 VCC
U15     1 A15                    2 A14                     3 node07
        4 node07                 5 A13                     6 node03
        7 DigitalGnd             8 node04                  9 A12
       10 A11                   11 node08                 12 SBCIOSel.L
       13 SBCIOSel.L            14 VCC
U16     1 IN3                    2 IN2                     3 IN1
        4 IN0                    5 BD7                     7 Sel420-42F
        8 DigitalGnd             9 BA2                    10 BA1
       11 BA0                   12 IN7                    13 IN6
       14 IN5                   15 IN4                    16 Vcc
U17     7 DigitalGnd             8 BRClk  14 VCC
U18     1 BD2                    2 BD3      3 Rcvr 4 DigitalGnd
        5 BD4                    6 BD5      7 BD6  8 BD7
       10 BA1                   11 Sel430-43F    12 BA0      13 R.L/W
       14 RxRdy                 16 DigitalGnd    17 CTS.L
       19 Xmt                   20 BRClk         21 Reset.H 22 node30
       23 RTS.L                 24 DTR.L         26 VCC     27BD0 28 BD1
U19     1 BA0                    2 BA1             3 BA2      4 OUT0
        5 OUT1                   6 OUT2            7 OUT3     8 DigitalGnd
        9 OUT4                  10 OUT5           11 OUT6    12 OUT7
       13 BA3                   14 Sel410-41F    15 Sel406-407   16 Vcc
U20     1 node19                 2 node20          3 node21        4 node23
        5 node24                 6 node22          7 Rcvr          8 Xmt
        9 SerialOut             10 node25         11 CTS.H
       12 DTROut.L              13 DTR.L 14 CTS.L  15 DigitalGnd 16 VCC
```

Chapter 12

XaMonV4B Firmware Monitor

All computers, regardless of their size, need a program to run when power is first turned on. This controller uses the upper 4K of the available EPROM space to hold such a program, the **XaMonV4B**-1 monitor.

This program resides in EPROM. Software that is burned into ROMS or EPROM'S is called *firmware*. This distinction becomes blurred when EEPROM and battery backed ram are considered. Occasionally, code loaded into ram will be referred to a "ramware."

This monitor can be viewed as a small *operating system*, that supports your using the on-board hardware of this machine in a more convenient way.

The **XaMonV4B** monitor program was designed to be *modular* in construction. This monitor was implemented by using subroutines that perform a relatively simple duty. These subroutines were in turn, used to build more complex *procedures* or *subroutines*. The terms procedure and subroutine can be used interchangeably in this text. You may want to use the term 'subroutine' when referring to a piece of code that does not have much possibility for reuse, and 'procedure' for a routine that does.

What's nice about the **XaMonV4B** procedures is that you can call these prewritten routines from *your* software. This can make your programming task a lot easier. The interface example at the end of this manual can show you how this is done.

After the following definitions is a description of the **XaMonV4B** routines. Not much distinction will be made as to whether a routine qualifies as a subroutine or a procedure. We'll let you decide!

Definitions Addressing Modes:

Different ways of instructing the CPU to access data or instructions within the memory map of the processor.

Accumulator: Central register of the 65C02. Most instructions act on this register in some way. Can be referred to as either Accum, Acc or A.

Index Registers: Referred to as the X and the Y registers. They, like the accumulator are 8 bits long. X & Y may be used as loop counters. The Y register, when used with the page 0 addressing mode, can use any two consecutive bytes on page 0 as a 16 bit register. This partially makes up for the lack of general purpose 16 bit registers.

With some addressing modes the X or Y registers act as indices to obtain an address. That address is then used to access some other location in memory.

No explanation will be given of the 65C02 instruction set. There are good books available on this topic. Included with the documentation is a file named **XaTestR.ASM** that is a source code listing of all the assembler psuedoOps and *every* 65C02 instruction in *every* 65C02 addressing mode. This listing can be used to check out your assembler and show you the syntax of all the available instructions. The subroutines to be described can be divided into seven basic categories;

1) **Utility**	: Low level timing, etc
2) **Serial I/O Routines**	: Uart - related
3) **Input Routines**	: For getting data
4) **Output Routines**	: For outputting data
5) **Conversion Routines**	: For acting on the data
6) **Data Manipulation**	: For user interfacing
7) **Higher-level Routines**	:For performing more complex operations

The following is a brief description of the subroutines and procedures used in the **XaMonV4B** monitor EPROM. Again, the term *subroutine* can be used to describe a lower-level operation. The term procedure can be used to describe a higher-level operation.

The distinction between these terms as used here is arbitrary; the reader is encouraged to make their own assessments.

From time to time you may see a "**CALL**" mnemonic used in some of the source code provided instead of the usual "**JSR**." The CALL is typically used to denote a call to a procedure and the **JSR** a jump to a subroutine. If you look at the listing, the hex object code generated for either of these mnemonics is the same. (20 hex) The intention here is twofold:

To communicate with *yourself*.

Many years ago I wrote a small inventory system in Fortran. (I said *many*) It represented about 12 weeks of work and I was quite proud of it. Five years later I dug the 10 pound listing out in order to sort through some ideas. I was left wondering *what*, *how* and *why* I wrote this code the way I did. The listing and its content was *useless*! Consequently; write your code well enough so that you can understand it 10 years from now. This is easy to do once you clean up any bad programming habits that you might have.

To communicate with others.

Think about what would happen if your life depended on someone else understanding your source code. This is a lofty ideal and a goal that is worth pursuing! Your documentation skills will grow.

The idea of *granularity* can be useful as you go about partitioning and structuring your source code for maximum effect.

Granularity is an important software design issue related to the concept of *code reuse* and *modularity*. Make a subroutine too specific (fine grained) and you lower the prospect of re-using it.

A similar result can occur when a procedure is too general (coarse grained.) As you examine and use the routines in **XaMonV4B**, consider these issues for yourself. You may find that some of these routines are "finely tuned" and that others could use some more "elegance," such as ease-of-use, understandability or efficiency.

The neat part of all this is that you can eventually write your *own* monitor, your own "bios." The ideal monitor would do everything you wanted it to, would take up zero EPROM space, and, after five years since looking at the source code, you would be able to understand what your program does after about 10 minutes of study!

Your reality will depend on how well you implement ***structured programming*** design techniques. Your future career may depend on how elegant your code writing style becomes. Many of the "big girls & boys" are writing embedded control programs using RISC processors (Reduced Instruction Set Computer) that use 90,000 + lines of source code and take up megabytes of memory. Now is probably the best time for you to practice writing compact, efficient, re-useable, understandable and maintainable code!! **Best to you!!**

XaMonV4B Routines

Immediately following the routine name is the routine's address, and the CPU register(s) affected in brackets []. The space following is empty if no registers are affected.

ToggleFlag F1DE [A]:Inverts the LSB in the accumulator (Acc.)

Push F1E1 [A]:Takes contents of Acc and pushes it onto the "user" stack located in memory locations C0-D0. Do not push more than 30 bytes onto the user stack or your program might demonstrate some really weird symptoms.

Pop F1F0 [A]:Takes last value on user stack and pops it into the Acc.

FastSwap F201 [A,Y]:Takes the "top" (last) two values on the user stack and swaps them.

SaveRegs F21C [A]:Saves the 65C02 CPU registers using pushes. Any call to SaveRegs must be accompanied by a call to RestoreRegs somewhere else in your program.

RestoreRegs F22D [A]:Restores the 65C02 registers using pops. Must be used if a prior call to SaveRegs has been made.

SaveSXYAP F240 [A]:Saves the 65C02 CPU registers to fixed locations in memory. (See interrupt routines starting at F8ED)

RestoreSXYA F25C [A]:Restores the 65C02 CPU registers from fixed locations in memory. (See interrupt routines)

FastWait F26D :An arbitrary unit of software-programmed time delay.

TenthMilliSec F271 :Delay one hundred micro Seconds. These software delays were calibrated using a frequency counter and were originally designed to be used for EPROM programming algorithms.

HalfMilliSec F280 :Delay five hundred micro Seconds.

MilliSecDelay F298 :Delay 1000 micro Seconds.

MSecDelay F29F [Y]:Enter with number of milli Seconds of delay desired in the Y register, maximum value = FF.

TenthSec F2BE :Delay one tenth of a second. (100 milli Seconds)

Delay1Second F2E4 :Delay one second. (1000 milli Seconds)

DelaySeconds F2EF :Enter with number of seconds of desired delay in X register, max = FF hex, 255 decimal.(over 4 minutes!)

ReadDSRBit F2F6 [A]:Read value of 2651 uart's DSR bit. RETURN TRUE (01=High) or FALSE (00=Low) in Acc.

SetRTSBit F301 [A]:Enter with desired BOOLEAN value (TRUE or FALSE) in Acc. RTS bit will be set accordingly. SetDTRBit F315 [A]:Enter with desired BOOLEAN value (TRUE or FALSE) in Acc. DTR bit will be set accordingly.

BusyReadRS232 F329 [A]:Check the 2651 uart's status register. If a character is present in the receive holding register RETURN TRUE in Acc with character in VAR "HoldAByte," otherwise RETURN FALSE in Acc.

ReadRS232 F33B [A]:Keep checking uart (forever if necessary) until a character is received. When character is found, pass in VAR HoldAByte.

WriteRS232 F343 [A]:Check to see if transmit holding register (thr) on 2651 is busy. If not busy, send character located in HoldAByte via thr. IF busy then check to see IF WriteTilSent is TRUE. IF TRUE then keep waiting until char can be sent, otherwise quit trying to send.

UnLockTransmit F356 [A]:Sends XOn char to **JTerm**. **JTerm** expects an XOn character before it will continue certain operations.

QuickEscTest F35E [A]:Checks RS232 port for an Escape character. IF found RETURN TRUE in Acc, otherwise RETURN FALSE. This subroutine is a useful way to break out of tight loops used for testing purposes.

SetBaud F37E [A]:See InitUart F38A. The 2651 uart chip can be set for baudrates from 50 to 19,200 baud. The VARiable "BaudRate" has to be loaded with the baud rate code before calling this routine. The baudrate code for 19,200 baud is 3F, and 3E for 9,600 baud. Consult the 2651 data sheet for other baud rate codes.

InitUart F38A [A]:This routine does the necessary initialization of the 2651 uart. It initializes the 2651's MODE and COMMAND registers.

InitCkSum F39B [A]:Clears the 16 bit VAR named "Checksum." CheckSumAdd F3A2 [A]:This routine will take the value in the Acc and add it to "Checksum," using modulo 16 arithmetic. For example, if the value in "Checksum" were FFFF and then you added 09 (hex), the result would be 0008 hex. The actual computed value would be 10008, but the carry bit is lost, hence 0008.

CheckSum F3AC [A]:Calculates the checksum of the currently defined buffer of memory starting at 'Start Address' and ending at (Start Address + ByteCount.) You can invoke the checksum from **JTerm** by pressing the 's' key.

SetOutput F3D4 [A]:This routine uses the low nibble in the Acc to set the output destination. **XaMonV4B** default is to direct I/O to the RS232 port, but you may re-direct the output to any device, using any device driver you wish. For example, the procedure "WriteString" may be used to output a string to the PC using **JTerm**, an LCD display, a pen plotter or even memory, depending on the IO re-direction CONSTant you use and the output driver you install.

SetInput F3DA [A]:This function is similar to SetOutput, but deals with the input destination. In this case, the IO redirection constant loaded from the Acc determines where the input routine will get its data. A byte could come from your custom interface, memory, the RS232 port or a keypad. The ability to install your own input drivers gives you lots of flexibility.

SetInOut F3E0 [A]:This routine takes the two nibbles in the Acc and sets both
the input and output destination for I/O. The upper nibble sets the input and the lower nibble sets the output destination.

SetIOToRS232 F3F1 [A]:This routine sets the default I/O re-direction to the RS232 port.

SetIOToRSInUser1Out F3F7 [A]:Notice in this example of redirection, (see source code) that the upper nibble (the "7" of 73) is used to set the input source. The "3" is used to set the output destination to your "user" routine.

GetSingleInput F3FD [A]:You enter this routine with the input bit number in the Acc. This routine returns a BOOLEAN value in the Acc corresponding to the value of the bit you choose.

```
  Ex:   LDA  #03 choose bit 3
        JSR  GetSingleInput
        BEQ  itIsLow
        BNE  itIsHigh
```

For some implementations, this routine is definitely overkill. See "ciL" at FE48 to see how it can be used.

SInputByte F40C [A]:One call to this routine reads all 8 of the single-input values as one byte-wide input value. This allows you to read 8 input levels (switches, TTL values, etc) at a time.

GetInput F42D [A]:This is the generalized input procedure that provides the mechanism for input re-direction.

AssertSingleOut F472 [A]:Creates an offset to the single output base address using the Y register. If the value in the Acc is a value from 0 to 7, the output bit will be turned off. If the Acc value is 8 to F the output bit will be turned on. The goal here was to create a higherlevel of abstraction.

TurnBitOff F479 [A]:Enter with bit number you want to turn off in Acc. Use any value from 0 to 7. You can turn a bit off from **JTerm** by pressing the 'o' key (lower-case) and then the bit number.

TurnBitOn F47E [A]:Enter with bit number you want to turn on in Acc. Any value from 0 to 7. You can turn a bit on from **JTerm** by pressing the "O" key (upper-case) and then the bit number.

SOutputByte F486 [A]:Enter with byte value in Acc that you want to present to the single bit output port as a byte-wide value.

SendOutput F4AE [A]:This is the generalized output procedure that provides the mechanism for output re-direction.

Wait4Char F4F3 [A]:There is a VAR named "Key" that this routine waits for. The default Key is XOn or 14 hex. You will loop in this routine until the input device currently installed presents "Key."

Wait4Space F502 [A]:This is an example of using Wait4Char with a different "Key" VAR value.

SCROrYesCk F513 [A]:Enter this procedure with the ascii char in Acc. A BOOLEAN value will be RETURNed in the Acc: TRUE if the character was a Space (20h),a carriage return (0Dh), or a 'Y' (59h), and FALSE for any other character.

DisableXOnHandShake F52B [A]: **JTerm** and **XaMonV4B** use a "software handshake" so that the PC will not lose characters sent to it while it is writing to the screen or performing some other system operation. This handshaking can be turned off by using this routine.

EnableXOnHandShake F531 [A]:Of course you get to turn handshaking back on if you want to. If you get garbled data on the screen using JTerm, press Cntrl-Z or reset the **SBC65V1B** by pressing S1.

ReadChar F537 [A]:Forces the MSB bit low, checks for the handshaking XOn char. If an XOn, it looks for another character. Character read is placed in VAR HoldAByte.

WriteChar F545 [A]:Checks to see if XOn protocol is being used. If it is, WriteChar waits for an XOn before sending the character stored in the VAR HoldAByte, out the (CIOD) currently installed output device. Otherwise it does not wait.

SpaceEscWait F557 [A]:Will loop until a space character is presented to the (CIID) currently installed input device.

SendChars F563 [A,X]:Sends the character found in the Acc, X number of times. See "Send3Spaces" at F59B.

WriteLn F56A [A]:Sends a carriage return (0D hex) to the CIOD.

SendCRLF F570 [A]:Sends a carriage return and a linefeed (0A hex) to the CIOD.

SendLNFD F573 [A]:Sends only the linefeed character.

ReleaseHost F579 [A]:Sends a control C character (03 hex) to the CIOD.

ClearScreen F57F [A]:Sends a formfeed control-L (1A hex) to the CIOD.

SendABarSave F587 :Saves the contents of the Acc, sends a "|" (7C hex) character, then restores the Acc to its original value.

SendADash F58F [A]:Sends a "-" character (2D hex) out the CIOD. Subroutines such as these are used to make the source code more readable.

SendASpace F595 [A]:Sends a (20 hex) character out.

Send3Spaces F59B [X]:Sends three space characters, clobbers your X register while doing so. (An ideal routine would not require you to know what registers are being used. A routine that does not "byte you in the hind-end" is said to have no side-effects. When you are NOT sure what registers are being used, it is often a good idea to save your registers before making a call to an "unknown" subroutine.)

InitTerminal F5A3 [A]:Initializes the 2651 uart and clears the screen.

WriteString F5AA [A]: Sends the ascii string following the call to WriteString out the CIOD. Remember to end the string with a null terminator character (00 hex) so that the routine knows when to quit. A string can be up to 255 characters including the terminator too.

PrintID F5CC [A]:An example of the use of WriteString. It is a good idea to "ID" all of your eprom firmware projects. **JTerm** invokes the PrintID routine by the "?" key. This idea can help you manage eprom version control.

DoneMessage F5EF [A]:Send the message "Done" out the CIOD.

PressSpaceBarMsg F5F9 [A]:Sends a "Press Space Key" message. Used by NMIRoutine.

GetYResponse F60E [A]:Prompts user with "(yes)." If a "y," a "spacebar" or "carriage return" key is received by the currently installed input device, this routine RETURNs a BOOLEAN TRUE in the Acc, else a FALSE if any other character is presented. The proper feedback character is also sent. **GetNoResponse** F634 [A]:Prompts user with "(no)." A "Y" response RETURNs a BOOLEAN FALSE in Acc, an "N" a BOOLEAN TRUE. This is an example of negative logic. See XaDebugRoutine at F8ED.

Conversion routine Definitions

Some confusion may exist when describing conversion routines. Here are some definitions you may find useful:

- **Ascii char** :A 1-byte code that represents a printable ascii CHARacter.
- **AlphaAscii** :Printable alphabetic Ascii char 'A'..'Z'
- **NumAscii** :Printable numeric Ascii char '0'..'9'
- **AlphaNumAsc**:An AlphaAscii or NumAscii character.
- **CntrlAscii** :Non-printable ascii chars EX: 'escape' = 1B hex
- **OtherAscii** :A printable non- AlphaNumeric ascii character EX: ' * '
- **HexAscii** :Printable AlphaNumAsc within range '0'..'9' or 'A'..'F' Hex Nibble :Representation of a 4-bit binary value, range 0..F. EX: 0111B (binary) = 7H (hex)
- **Hex Byte** :Representation of an 8-bit binary value, range 00..FF. EX: 0001 0111B = 1 7H

Some confusion may arise in the fact that a **BYTE** = **CHAR**. Some routines will act on *any* 8-bit entity, regardless of how us humans interpret them. Hex is the representation generally used for the 65C02, and ascii is generally intended to be printed onto a crt screen or a printer for human consumption, though there is no hard and fast rule.

Keep your data ***TYPES*** as clear as possible. Focus on your intention of what the data is for, and strive for clarity and ease of understanding. Please share with others any improvements you might make in this direction.

Remember, the computer industry per say can make things C ryptic and C omplicated and C onfusing. You can be part of the solution rather than part of the problem. Maybe it is possible to make assembly language as easy to read as a higher-level language?

I think that for whatever reason, the computer industry dropped the ball with regard to computer languages

I'm glad to see that cross-assemblers writers are getting away from forcing us to use those inane 8 character labels and identifiers and allowing *both* upper and lower case letters to be used.

The listing generated by an assembler should be as clean and readable as possible, with no redundant or useless information.

Comment entry should be structured in such a way as to not interfere with the readability of the text.

And finally, source code should not be position dependent. You should have a wide latitude of options for entering your source code text.

IsItAscii09 F656 [A]:Enter with an ascii char in Acc. RETURNs TRUE if Printable NumAscii character, FALSE if it is not.

IsItAsciiAF F666 [A]:Similar to IsItAscii09, but AlphaAscii char range is within 'A..F' IsItAscii0F F676 [A]:Similar to previous two routines, RETURNs TRUE in Acc if char is a HexAscii character.

IsItAscii F689 [A]:RETURNs TRUE if character is a printable AlphaNumAsc char.

LwrCase2UprCAsc F6A3 [A]: Converts char in Acc from lower case to upper case ascii.

HexAsciiToHexNibble F6AF [A]:Enter with a printable HexAscii character in Acc, RETURN lower Hex Nibble in Acc.

HexNibbleToHexAscii F6C9 [A]:Enter with lower Hex Nibble in Acc, RETURN printable HexAscii character in Acc.

GetAsciiNumber F6D9 [A]:Read a char from currently installed input device until a printable HexAscii number is encountered, which is RETURNed in Acc and VAR "HoldAByte."

GetHexAscii F6E4 [A]:Reads a character from input device. Filters out all characters but a HexAscii char. RETURNS char in Acc and HoldAByte. (location FC hex)

GetSingleHexNum F6EF [A]:Read a HexAscii number, send Ascii code to output device, and then convert it to its Hex Nibble equivalent in Acc. **GetHexByte** F6FE [A]:Get 2 HexAscii characters and create a Hex Byte representation. EX: 'E' '3' becomes hex E3 in Acc and HoldAByte.

PrinteByte F71D [A]:Creates the HexAscii characters representing a Hex Byte and sends them to the CIOD.

PrintNible F72B [A]:Creates the HexAscii character representing the lower Hex Nibble and sends it to the output device.

PrintByteSave F732 :Same as PrintByte, but Acc is not clobbered.

GetZPWord F738 [A]:Support routine for PrintWord and WriteCard.

PrintWord F74C [A]:Creates the HexAscii characters representing two Hex Bytes. The two bytes together form a 16 bit word. The HexAscii chars are sent in the following order: "high byte" upper nibble, "high byte" lower nibble, "low byte" upper nibble, "low byte" lower nibble. This routine works only on zero page locations 00-FF.

PrintSBCAdrs F75C [A]: An example of the use of PrintWord, sends the current Program Counter out to the CIOD.

IncCardNumber F762 [A]:Increments a 5 byte (temp)buffer of hex characters using modulo 10 addition. Routine starts with least significant digit and adds one. If there is a base 10 overflow, it continues to increment the next higher digit. Notice that there is an entry point to increment any of the 5 digits.

SkipLeadingZeroes F7B8 [A,Y]:Will send a space char out the installed output device for every leading zero encountered in the 5 digit array.

PrintCardArray F7C6 [A,Y]:Sends the 5 digit array out as 5 HexNumber characters to the CIOD. (currently installed output device)

ClearTempBufLocs F7D8 [A,X,Y]:Clears 16 bytes of the TempBuffer array to zero.

CountByteDec F7E8 [A]:Enter with the amount to add to array in Acc. Tempbuffer contains the 5 BCD digit array result.

WriteByteDec F7F6 :Sends decimal representation of a byte in NumAscii form to the CIOD.

WriteCard F806 [A]:Sends decimal representation of a word to the CIOD.

HexToDecimal F84D [A]:Converts a two-byte hex word to its ascii decimal equivalent and sends it to the CIOD.

PrintPC F86E [A]:Send Program Counter to CIOD in HexAscii form.

PrintRegs F884 [A]:Display the status of CPU regs via CIOD.

XaDebugRoutine F8ED [A]:This is the routine installed by the monitor that is called when a software interrupt (BRK instruction) is encountered. You may install your own debug routine. A user installed debug routine might include a "printout" of the variables and memory locations of interest to you. See InstalDBRoutine at FFE0.

IsItSWI F91B [A]: Two types of interrupts occupy the same vector, the hardware IRQ and the software (SWI) BRK instruction. This routine RETURNs a TRUE in Acc if a BRK and FALSE if interrupt is an IRQ.

NMIRoutine F92A:Non-maskable interrupt routine installed by the monitor. You may install your own. See InstalNMIRoutine at FF9E.

SWIRoutine F948:Software Interrupt Routine installed by the monitor. You may install your own. See InstalSWIRoutine at FFBF.

SelectInterrupt F995 [A]:This routine determines the type of maskable interrupt encountered and calls the appropriate (SWI or IRQ) routine.

ShowAllParameters F9AD [A]:Displays the current buffer information, i.e. starting address, and bytecount by sending info in ascii form to CIOD.

GetFillWP F9DC [A]:WP = "with prompt." Prompts you for the fill (memory) hex character going to VAR "FillChar."

GetFillChr F9EC [A]:Similar to GetFillWP, without the prompt.

GetNumberOfBytes F9F3 [A]:VARiable "ByteCount" gets (:=) hex byte representing the number of unit-bytes for the buffer.

GetNumberOfPages F9FA [A]:VAR "PageCount" (ByteCount+1) gets hex byte representing the number of even pages for the buffer. GetStartAdr FA01 [A]:VAR "StartOfBuffer" gets two bytes to become the currently defined buffer's starting address.

GetDestAdr FA11 [A]:Similar to GetStartAdr, this routine loads the address of the destination buffer used for the "Move" command.

GenEndAdr FA1E [A]:Given the starting address and size of the current buffer, this routine computes the end address of the buffer.

GetAddrWP FA32 [A]:Prompt for starting address and GetAddress FA3D [A]:initialize necessary buffer variables.

GetDestAdWP FA44 [A]:Prompt for destination address and GetDestAddr FA4F [A]:initialize necessary destination buffer variables.

GetByteCntWP FA56 [A]:Prompt for byte count and GetByteCnt FA63 [A]:initialize necessary buffer variables.

GetPgCntWP FA6A [A]:Prompt for page count and

GetPageCnt FA77 [A]:initialize necessary buffer variables.

GetAllPar FA7E [A]:Prompt for all buffer variables and initializes buffer.

ResetBufferPointers FA8B [A,Y]:Reset the current buffer's pointers back to their starting location. Call this routine before you perform any action on the currently defined buffer area.

SaveParams FAA2 [A]:Save the current buffer settings.

RestoreParams FAC7 [A]:Restore the formerly saved buffer settings. This routine can be useful for rom burner firmware.

CkEndOfBuffer FAEC [A]:This routine increments the source and destination buffer pointers and checks to see if the end of buffer has been reached. If end-of-buffer, RETURNs TRUE in Acc otherwise FALSE.

XferBuffer FB0A [A]:Transfer a buffer-worth of bytes from currently installed input device to (CIOD) currently installed output device.

MoveMemory FB16 [A]:An example of the use of XferBuffer. FillMem FB1D [A]:Takes the hex byte in VAR "FillChar" and sends it out CIOD "bytecount" number of times.

FillRoutine FB2C [A]:An example of I/O re-directed to memory.

ClearSBCRam FB40 [A]:Load any part of RAM with any hex byte, without affecting the current buffer settings.

DownLoad FB50 [A]:Procedure used by **JTerm** to load files from the PC into the **SBC65V1B's** memory. This routine uses a XOn protocol, so that **JTerm** only sends a character (byte) to the SBC when the SBC is ready to receive it.

UpLoad FB68 [A]:Procedure used by SBC to load buffer ram area up to a PC UpLoad.ROC file.

LoadHexData FB7C [A]:Load hex data into SBC ram buffer area via currently installed input device. This is the routine you use to hand-load keyboard data into the **SBC65V1B** using **Jterm**.

CkPtrForHex0 FB97 [A]:RETURNs TRUE if lower nibble in Acc is 00, FALSE if any other hex number. IncPointersSave FBA3 :This routine is used by the PrintMem procedure. It increments the temporary buffer pointer and the currently defined buffer pointer. The VAR "ExitFlag" is set to TRUE if the buffer is at the end, FALSE if not. WriteTempBfrChar FBC9 [A]:Writes byte in Acc to the temp buffer. RETURNs TRUE in Acc if seventh character, FALSE if not.

GetTempBfrChar FBDB [A]:Get a char from temp buffer and store in VAR HoldAByte. RETURN TRUE in Acc if eigth character, FALSE if not.

GenerateSpaces FBED [A]:Sends spaces out to CIOD. See PrintMem below.

PrintLineDump FC0E [A]:Send one line of the current buffer area to CIOD as formatted HexAscii.

PrintAscii FC3F [A,X]:Send ascii equivalent of the previously dumped hex line to CIOD.

PrintMem FC6D [A]:This procedure dumps the entire contents of the currently defined buffer to the CIOD.

NewResetVector FC90 [A]:Takes the starting address and installs it into the **SBC65V1B** reset vector. Subsequent depressions of the reset button S1 on the SBC will vector to the program you have loaded beginning at the currently defined buffer's starting address. J15 on the SBC must be open.

NewTestVector FCBC [A]:Similar to NewResetVector above, but is activated upon reset only when J15 is shorted. It is thus possible to have two different programs available with reset. If you use a toggle switch to open/close J15, you can control which program you run from a "front panel."

Monitor-Specific Routines

InitProcedures FCE4 [A]:Initialize buffer address and I/O.

InitServiceRoutines FCF1 [A]:Puts a JMP instruction (4C hex) into the leading byte of each user-installable service routine.

InstallRoutines FD06 [A]:Installs the necessary pointers for all of the user installable routines of the **XaMonV4B** monitor EPROM.

InitMonitor FD1F [A]:Initializes monitor flags and pointers. Executed upon power-up or with the JTerm "+" command.

InitVarsEaReset FD50 [A]:Initializes the VARs that need to be set each time the SBC is reset with the push of S1.

Initialize FD62 [A]:This is a very important routine. This routine is called each time the SBC is reset. The very first time this routine is run, it does a full system reset of all variables and procedures. After that, only a "partial" reset is performed. This action may be described as a "Hard" or "Cold-Start" with the first reset, and a "warm" start upon subsequent resets.

Begin FD78 [A]:This is the entry point of the monitor. It all starts here!

CmdInterp FD8E [A]:This is the command interpreter of **XaMonV4B**. It waits for a character to be input from the currently installed input device. When a character is received, it determines which procedures to run by doing a simple compare of the character to the available commands. If a command is valid, it executes the command. Use **JTerm** and press Cntrl-N for a list of the available commands.

TestProgram FF4A [A]:This is the blink test routine that you can run merely by shorting J15 on the **SBC65V1B** and pressing the reset button S1.

IRQRoutine FF7C [A]:This routine is run every time a hardware interrupt is activated on pin 4 of the 65C02.

Installing Routines

There are several routines that may be replaced by your own routines. "InstalNMIRoutine" will be used as an example. Notice the instruction

```
LDA #>NMIRoutine
```

This instruction may be read as "load the accumulator, with the immediate high byte of the address of the label "NMIRoutine."

What happens is the assembler obtains the address of "NMIRoutine" from the **SYM**bol table.

In this case that address is F92A. Since the high-byte of F92A is 'F9,' F9 is loaded into the accumulator of the 65C02. That data is then put into the high-byte of the NMI routine's pointer, "NMIPointer+1."

Next, the instruction

```
LDA #NMIRoutine
```

gets the low-byte of the address loaded into the Acc, and then stores the low-byte of NMIRoutine's address into the low byte of the NMIPointer.

As you will see later, the monitor's hardware vector for an NMI is 0200. When a low-level is present on the CPU's non-maskable interrupt pin, the program will jump to 0200. Location 0200 has a 4C hex "JMP" instruction, which was loaded by the "InitServiceRoutines" subroutine. Program control will now go to the following two bytes, which contain the address of the routine you want to execute. In this case, the NMIRoutine. See lines 158-177 of monitor listing for additional information.

InstalNMIRoutine FF9E [A]:When a low level is present on pin 6 of the 65C02, an non-maskable interrupt is initiated. The monitor is set up to run the program/procedure/routine whose address is loaded into the SBC's NMI pointers.
InstalProgram1 FFA9 [A]:With J15 open, the routine installed here is run when reset button S1 on the SBC is pushed.
InstalProgram2 FFB4 [A]:With J15 shorted, the routine installed here is run when reset button S1 on the SBC is pushed.
InstalSWIRoutine FFBF [A]:The routine installed here will run every time a break instruction (BRK) is executed in your program.
InstalIRQRoutine FFCA [A]:The routine installed here will run every time an IRQ low pulse is sensed at pin 4 of the 65C02. The interrupt enable bit must be activated in the CPU status register with the "CLI" instruction.
InstalSelectRoutine FFD5 [A]:This vector was used to allow you a little more flexibility as to how you get to your IRQ and SWI routines, by allowing you the possibility of installing your own "Select Routine."
InstalDBRoutine FFE0 [A]:The routine here will run every time a BRK instruction is encountered and the "DebugFlag" is TRUE.
InitTempBuffer FFF1 [A]:This routine set the location of your tempbuffer used for WriteCard and PrintMem procedures.

CPU Hardware Vectors FFFA-FFFF

The CPU vectors on this monitor are set to the following locations:

```
  Loc:  FFFF FFFE  FFFD FFFC  FFFB FFFA
 Data:  02   09    FD   78    02   00
```

The CPU will jump to 0200 when a non-maskable interrupt is encountered, FD78 when a reset is encountered and 0209 when a maskable interrupt is encountered. Notice that two of the vectors jump to locations in SBC ram. This allows you to place ***YOUR*** vectors in ram, hence you can install your own interrupt service routines!

One final note. The F000 to F1DD portion of the memory map is not actually part of the monitor. It is a "scratchpad" area that can be used for anything you want. **JComm LAB** has used this area to put LCD display and Dallas "SmartWatch" initialization routines. You can clear this area by reading the **XaMonV4B** into your rom burner and putting FF into locations F000-F1DD (as mapped in the SBC not in your rom burner), and then burning a new monitor.

If you decide to use a 64K EPROM, be sure to burn the monitor image into the high-portion of each 32K byte "chunk." Then you can set up the bank switching scheme which will give you even more versatility. Maybe there is room enough for a really big project?

Your program's starting address will be placed in the reset vector @ FFFC FFFD. All you will have to do to run your program after you have burned your program and new reset vector into an EPROM is to install the EPROM into the socket and turn on the power! (Be sure to include properly working initialization routines!)

The uart also has a general purpose output bit which is available to wire into the bank select pin of the EPROM, **PN1**. This will allow you to have two 28K (32K minus monitor) programs available in EPROM, under software control as discussed in **Chapter 9.**

Chapter 13

XaMonV4B Firmware Monitor Listing

A monitor listing can serve you in several ways. Here are some suggestions.

- By using your editor's 'search' command on the monitor listing, you can find actual examples of how different 65C02 instructions are used. For example, you could search on 'TXA' and observe how this instruction is used by this monitor.
- You might use a monitor routine as the basis for ***your own original subroutine*** or procedure.
- You may want to modify or optimize some of the routines in this monitor; to make a routine shorter in length, faster, or easier to use or understand.
- This listing can serve as a starting point for designing and writing ***your own*** monitor. There may be routines in this monitor you don't want or need. You may have some original routine that begs to be part of ***YOUR*** monitor!
- Few implementations are optimum and this monitor is no exception. If you are designing a monitor -- regardless of the type of processor you are using -- this listing might give you some ideas. At least you won't have to start from ground zero!

Listing Organization

Notice the line numbers at the far left of the listing. These numbers range from 0001 to 2200. Each number represents a line of code or a comment. Reference to these numbers are made in this chapter.

The first source code line (0001) contains the **MODULE** name. This name, as well as all labels and identifiers in the body of the source code, can be up to 31 characters long.

All text is upper and lower case sensitive. The label "LoopPoint" and "Looppoint" would show up in the SYMbol table as being two different entries. Label names beginning with lower case letters do not show up in the symbol table. This removes the clutter of local labels from the symbol table.

This points to a possible programming style: If a label is local to a procedure, begin the label with a lower case letter. Important procedures should begin with an upper case letter. A numeric character may not be used at the beginning of a label or other identifier.

Information Block

This block start on line 0002 and continues to line 0014. This text does not have to be present, but is highly recommended. This text contains your company name, the current date and time, a short description of what the code does, the file name and size, and other useful information. Oftentimes you may start a project from code previously written. (remember code reuse) I like to include the 'source' of this source code in this section. The size of both the source and object code can be important to you as well.

During a project, I like to keep a record of how many edit-assemble-run cycles I initiate on a given day. This total is in turn added to a grand total, along with the number of lines of code written. This can provide information as to how long a project took to complete.

Project history (lines 15-44)
can be a useful part of you firmware documentation. This can assist you with the all important version control. Any time you add or modify the source code, you may find it helpful to include the date of the modification and what the heck it is you did. Notice **XaMonV4** B was started in Aug of 1991 and its earliest predecessor in July of 1988!

CONSTant Section (lines 45 through 116)
contain the monitor's constants. Notice that there are several categories of constants. These constants are "classified" so as to make the source code more readable and/or understandable.
Ascii constants are generally used for the "non-printable" ascii characters. A printable character is usually declared in single quotes. For example LDA #'A' loads an ascii "A" into the CPU's accumulator.
Boolean constants can be used a parameters in subroutines or procedures that pass a TRUE or FALSE back to the calling program (usually through the Accumulator) and as values for flags. **Mask constants** are used in routines that operate on one or more bits of an 8-bit byte. (see line 510)
I/O redirection constants are used by the monitor to change the source and destination of input and output in a more readable way. For example, see lines 1627-1635 of this listing.
Serial constants are used by the RS232 uart routines for the 2651 uart, U18. These are the initial values used by the monitor to set the uarts's registers controlling baud rate, etc.

Page Zero Variables
The memory locations from 0000H to 00FFH are very important, because of the 65C02's zero page addressing mode. Instructions operating on page zero locations execute more rapidly. They take two bytes rather than three.
Page zero locations can also operate as (up to 128) 16-bit registers. The author's goal was to use as few of these valuable locations as possible. 70 out of the possible 256 were used, 32 of which are dedicated to a "User Stack." That means that only 15% of page zero is actually used by this monitor. This usage is found on lines 117 to 154.
Page 2 Variables
To save page zero space, many of the monitor's variables were placed in locations 200H to 2FFH (lines 155 to 230) It was determined that most of these variables are not time-critical. In other words, they are not called often enough to warrant being located on page zero. Notice that there are 159 locations at the end of page 2 for your program's variables.

Hardware Equates
These are a special type of constants that represent the addresses of the I/O section of the **SBC65V1B** hardware. (lines 234-250)

System Equates (lines 251-257) are to remind you of the locations of the reset and interrupt vectors of the CPU.

Code Section

At reset, code begins executing at address FD78 (hex). Notice that the supporting procedures found on lines 265-1893 are skipped over to the beginning of the actual monitor program on line 1895, (ending at line 2091). This was done intentionally to mimic the **Modula-2** style of programming, which places the supporting procedures before the main body of the program.

Notice the four-digit hex number to the right of each line number. This number represents the CPU's program counter. Following the program counter are one to three bytes which represent either CPU instructions or data. These hex codes are generated by the cross-assembler.

Comments

Comments are entered in one of two ways.

1) If a line begins with an asterisk, all characters to the right of the asterisk are considered to be comments.
2) All characters to the right of an instruction mnemonic are considered comments. The only exception; some single byte commands like ASL. These instructions will give you an error message if you place a comment on the same line with it.

Instructions for running this monitor out of SBC65V1B RAM

Step 1 Change the **XaMonV4B**.ASM monitor source code. Change the ORG value from 0F1DE to ORG:= 0500 and the "34" in line 1864 of InitMonitor and line 1886 of Initialize to "43." Modify 'PrintID' procedure by adding an "R." The message will now read "**XaMonV4B**-1R." Re-assemble **XaMonV4B**.ASM using **JAsm** on your PC.
Step 2 Run **JTerm**. Press the reset button on the **SBC65V1B** module. Press '+' key on keyboard to reset the monitor's VAR's.
Step 3 Download object code **XaMonV4B**.ROC (using **JTerm's** 'Control-D' command) into **SBC65V1B** ram starting at location 0500H.
Step 4 Press control-Z key.
Step 5 Run monitor program from **JTerm** by pressing 'G' (for go). CRT should display ID message "**JComm LAB uCEL (TM) XaMonV4B-1R**"

Notes:

Because of the initialization structure used, reset will take you back to the Eprom **XaMon4B**-1 monitor. You have to use the 'G' go command to reenter your "ram" monitor.
JTerm commands are case-sensitive, so put your caps lock 'ON', and use the shift key to get to the lower case letters.
Monitor modifications are best done one routine at a time. Rather than use the entire monitor in RAM, a better approach is to make a small set of programs to test out each new idea. The technique envolves writing two programs. One is exclusively for the new procedure of interest, the other is a test program. For example, say you wanted to write a new "WriteCard" procedure;

Phase 1 Write the new routine as a separate program.

```
 MODULE     NewWriteCard -XaMonV4B
 EQUates-WriteCard
                  NOP --your new code-
                  RTS
              END NewWriteCard
```

Phase 2 Download the NewWriteCard object code into SBC ram @ 500 hex
Phase 3 Write a test program using the previously downloaded address of the procedure in ram as the reference. You can get the address of WriteCard from the NewWriteCard SYMbol table or the LST file.

```
 MODULE     WriteCardTest -XaMonV4B
EQUates-WriteCard ADR 0500 (instead of XaMonV4B's 0F806)
            --(your test code)-
           RTS
           END    WriteCardTest
```

Phase 4 Download WriteCardTest at say, 1000H and then run it. Your test program can now use your version of WriteCard that you have downloaded into another location of ram, rather than the version of WriteCard in the monitor EPROM.
It is possible to combine the two program technique into one program.

If you don't understand this discussion, continue using the **XaMonV4B** routines, continue to write your own "ramware," understanding will follow!

SBC65V1B XaMonV4B.ASM Listing

```
|*
|*     JComm LAB Sat 20 Apr 1992 10:39 AM
|*
|***----------- --Begin Information Block----------------***
|*Description:Single Board Computer Monitor Eprom code for
|*            SBC65V1B ECB-23079101 MN4301-I
|*FileName :XaMonV4B.ASM  Created FROM XaMonV4A.ASM
|*Filesize    :Source = 63,304 Object Code = 3,618 (E22 Hex)
|*System    :XaMonV4B-1 on SBC65V1B
|*Last Mod :Mon 14 Mar 1992 Assemblies:
|*Statistics :Last Hours :00.0 Hrs/ 0000 Lines/000 Assemblies
|*            Total Hours:99.9 Hrs/ 2199 Lines/218 Assemblies
|*Start Date :6 Jul 1988 End Date: 1 Jun 1992
|***-------------- End Information Block-----------------***
|*
|*History:
|* Version V3.0C  6 Jul 1988 1st Assembled on Lilith as SBCMon
|*12 Feb 1989 Started New version for SBC65V1A
|* Version V4.0A 14 Jul 1989 1st working ROM of new version
|* Version V4.0B 19 Jul 1989
|* Version V4.0C 25 Jul 1989
|* Version V4.0D 25 Nov 1989 had bugs
|* Version V4.0D  2 Dec 1989 DownLoad,UpLoad
|* Version V4.0D  3 Dec 1989 With Eprom Burner stuff
|* Version V4.0E 21 Dec 1989 Push,Pop,BitOn/Off
|* Version V4.0F 29 Sep 1990 New PrintMem,I/O redirection
|*                           Interrupts,LwrCase2Upr (2Wks)
|* Version V4.0G 14 Oct 1990 New XOn handshake 2,402Bytes
|* XaMonV4A3017  23 Oct 1990 WriteCard,XaRom SBR's imported
|*         3022  24 Oct 1990 Improved Interrupts
|*         3195  30 Oct 1990 Debug,Trace,NMIRoutine
|*         3200  31 Oct 1990 Add Uart I/O bit
|* XaMonV4B1     16 Aug 1991 Remove Parallel I/O,add comments
|*                7 Oct 1991 Add BusyReadRS232
|*         3312   9 Oct 1991 New Single Bit Input/Output
|*         3636  16 Oct 1991 Add ParIn & ParOut Drivers
|*         3388  21 Oct 1991 Uart pin test,tweaked delays
|*         3399  28 Jan 1992 SBC65V1B I/O NewMenu
|*         3463  09 Feb 1992 SInputByte,Reset,RemoveParOut/In
|*         3555  20 Feb 1992 Re-do reset, Add CkSum
|*         3554  22 Feb 1992 Routine1,routine2 load vector
|*         3592   9 Mar 1992 Fix bug in checksum,reset
|*               20 Apr 1992 Add SmartWatch Loop-up Table
|*         3618   1 Jun 1992 Move start of mon to F1DE
|*
|* Equate Definitions: Must begin with an upper case letter
|*                    to show up in SYMbol table
|* EQU = Equate = CONSTant
|* VAR  = Variable (1 byte)
|* PTR  = POINTER var,(bytes that hold an address
|*             of where data is, rather than the data itself)
|*             Must be a Page Zero VARiable
|* ADR  = ADDRESS constant ('fixed' location in mem)
|* String Equates:Hex 0 must end any WriteString
|* ASC  = data within quotes ' ' are Ascii chars
|* ASZ  = ascii chars followed by 0 marking end-of-string
|* RASC = return char, followed by ascii chars
|* RASZ = ret char, followed by ascii, then zero
```

```
              |*
              |*CONSTants
              |*
              |*Ascii CONSTants
              |*
define        |NULL     EQU  00
define        |ControlC EQU  03
define        |LineFeed EQU  0A
define        |CReturn  EQU  0D
define        |XOff     EQU  13
define        |XOn      EQU  14
define        |ControlZ EQU  1A
define        |Escape   EQU  1B
define        |Space    EQU  20
              |*
              |*    BOOLEAN CONSTants
              |*
define        |FALSE    EQU  00   = 'OFF'
define        |TRUE     EQU  01   = 'ON'
              |*
              |*     MASK CONSTants
              |*
define        |BIT0     EQU  01    %0000 0001
define        |BIT1     EQU  02    %0000 0010
define        |BIT4     EQU  10    %0001 0000
define        |BIT5     EQU  20    %0010 0000
              |*
              |*    I/O Redirection CONSTants
              |*
              |* The following constants are used w/the Procedures GetInput
              |* & SendOutput for I/O redirection. SetInput SetOutput &
              |* SetInOut allow the user to declare which routine is
              |* used for input or output.
              |* TYPE I/O=
define        |MEMORY1        EQU  0     Source
define        |MEMORY2        EQU  1     Destination
define        |SINGLEBYTE     EQU  2
define        |USER1          EQU  3
define        |USER2          EQU  4
define        |SCREEN         EQU  5
define        |KEYBOARD       EQU  6
define        |RS232          EQU  7
              |*
              |* I/O Redirection CONSTants for loading into 'IOState'
              |*
define        |InMem1OutMem2  EQU  01               0     1
define        |InRSOutRS      EQU  77      input^     ^output
define        |InRSOutUser1   EQU  73
              |*
              |* Timing CONSTant
              |*
define        |MSecCount      EQU  0C0 was 198Z (Z=Decimal)
              |*
              |* SBC65V1B Serial CONSTants
              |*
define        |ENABLE    EQU  05   2 Stop Bits,Odd Parity,Par Disabl,8 Data
define        |MODE1     EQU  0CE  ASYNC 16X RATE= CE
define        |MODE2     EQU  3F   19200 Baud INT CLK 1X RATE, via Table 1
              |*
```

```
               |*    Page Zero VARiables (Mem locs 00-FF) 0119              |*
               |*Eprom Burner Vars (If you use XaRom SBR's, don't use B8-BF)
               |*
define         |ResetPointr PTR  0B8 0B9
define         |RomAdr      ADR  0BA 0BB
define         |TotalCount  VAR  0BC 0BD
define         |Checksum    VAR  0BE 0BF
               |*
               |*   User Stack
               |*
define         |StackSpace  ADR  0C0  0DE and 0DF used by Push & Pop.
               |* You have 30Z locations available to Push onto.
               |* Past that, your whole darn program blows up!!
               |* 0133               |*   Global POINTER s
               |*
define         |GPOINTER       ADR  0E0
define         |BufferPointer1 PTR  GPOINTER
define         |BufferPointer2 PTR  GPOINTER+2
define         |StackAddress   PTR  GPOINTER+4
define         |StackPointer   PTR  GPOINTER+6
define         |TempPointer    PTR  GPOINTER+8Z   ('Z' = Decimal)
define         |TempBufferPtr  PTR  GPOINTER+10Z
define         |UserPC         PTR  GPOINTER+12Z User's Program Counter
define         |Debug          PTR  GPOINTER+14Z User's SWI break routine
               |*
define         |IOPointer   ADR  0F0          Pointers to I/O routines
define         |UserInput1  PTR  IOPointer    User's input routine address
define         |UserInput2  PTR  IOPointer+2
define         |KeyboardIn  PTR  IOPointer+4 User's Keyboard routine address
define         |UserOutput1 PTR  IOPointer+6 User's output routine address
define         |UserOutput2 PTR  IOPointer+8
define         |Screen      PTR  IOPointer+10Z User's CRT routine address
define         |HoldAByte   VAR  IOPointer+12Z Single value par used for I/O
define         |TempLoc     VAR  IOPointer+13Z
define         |UserIOVars  VAR  IOPointer+14Z User I/O variables
               |*
               |*    PAGE 2 VARiables
               |*
               |* Interrupt Vectors Summary
               |*
               |* Int  |Eprom Loc|Points To|Initialized To|MonitorInstalls
               |* NMI  | 0FFFA/B |  0200   |JMP NMIService|NMIRoutine
               |* Reset| 0FFFC/D |  0F000  |JMP Begin     |
               |* SWI  | 0FFFE/F |  0209   |JMP IntService|IRQRoutine
               |* IRQ  | 0FFFE/F |
               |* Example:
               |*When a NMI interrupt signal is encountered, the CPU gets
               |*the vector it is going to JMP to from 0FFFA-0FFFB. In this
               |*example, 0FFFA/B contain 0200. 0200 is a loc in RAM that
               |*has been initialized by the monitor with 4C, the 'JMP'
               |*instruction. Ram locs 0201-0202 have been initialized by
               |*the monitor with the address of the NMIRoutine. Interrupt
               |*service then becomes a series of jumps, as follows:
               |*CPU NMI>then JMP to 0200>then JMP to NMIRoutine. By having
               |*the Eprom vector pointing to a previously initialized
               |*location in RAM, you can install your OWN interrupt
               |*service routines! (ISR's)
               |*
define         |Page2          ADR  0200  Monitor Installs
```

```
0179 define         |NMIService     ADR  Page2     0200
0180 define         |NMIPointer     VAR  Page2+1Z  0201/202      NMIRoutine
0181 define         |SWIService     ADR  Page2+3Z  0203
0182 define         |SWIPointer     VAR  Page2+4Z  0204/205      SWIRoutine
0183 define         |IRQService     ADR  Page2+6Z  0206
0184 define         |IRQPointer     VAR  Page2+7Z  0207/208      IRQRoutine
0185 define         |IntService     ADR  Page2+9Z  0209
0186 define         |IntServPointer VAR  Page2+10Z 020A/20B      SelectInterrupt
0187 define         |Prog1Service   ADR  Page2+12Z 020C
0188 define         |Program1Ptr    VAR  Page2+13Z 020D/20E      cmdI address
0189 define         |Prog2Service   ADR  Page2+15Z 020F
0190 define         |Program2Ptr    VAR  Page2+16Z 0210/211      TestProgram Adr
0191 define         |IRQTable       ADR  Page2+18Z 0212 to 214 (3 locations)
0192                |*
0193                |* Page 2 VARiables
0194                |*                                Hex Address
0195 define         |BufferEnd        VAR  Page2+21Z  0215/0216
0196 define         |StartOfBuffer1   VAR  Page2+23Z  0217/0218
0197 define         |StartOfBuffer2   VAR  Page2+25Z  0219/021A
0198 define         |ByteCount        VAR  Page2+27Z  021B
0199 define         |PageCount        VAR  Page2+28Z  021C
0200                |*
0201 define         |ExitFlag         VAR  Page2+29Z  021D
0202 define         |DebugFlag        VAR  Page2+30Z  021E
0203 define         |UserFlag         VAR  Page2+31Z  021F
0204 define         |ResetFlag        VAR  Page2+32Z  0220
0205 define         |TraceFlag        VAR  Page2+33Z  0221
0206 define         |WriteTilSent     VAR  Page2+34Z  0222
0207 define         |XOnFlag          VAR  Page2+35Z  0223
0208                |*
0209 define         |BaudRate         VAR  Page2+36Z  0224
0210 define         |Count            VAR  Page2+37Z  0225 0226
0211 define         |CPUXReg          VAR  Page2+39Z  0227
0212 define         |CPUYReg          VAR  Page2+40Z  0228
0213 define         |CPUAccumulator   VAR  Page2+41Z  0229
0214 define         |CPUStatusReg     VAR  Page2+42Z  022A
0215 define         |CPUPgmCounterLo VAR Page2+43Z   022B
0216 define         |CPUPgmCounterHi VAR  Page2+44Z  022C
0217 define         |UserDelayTime    VAR  Page2+45Z  022D 022E
0218 define         |ErrorMsg         VAR  Page2+47Z  022F
0219 define         |FillChar         VAR  Page2+48Z  0230
0220 define         |InputDest        VAR  Page2+49Z
0231 0221 define    |IOState          VAR  Page2+50Z  0232
0222 define         |Key              VAR  Page2+51Z  0233
0223 define         |OutputDest       VAR  Page2+52Z
0234 0224 define    |RomStart         VAR  Page2+53Z  0235
0236
0225 define         |TraceVar         VAR  Page2+55Z 0237
0226 define         |WarmStart        VAR  Page2+56Z 0238
0227 define         |SmartWatchPad    VAR  Page2+57Z 0239
0228 define         |RomVars          EQU  023A to 0253 Locs used by Romburner
0229 define         |OpenRomVar       EQU  0254 to 0260 Romburner expansion
0230 define         |UserVars         EQU  0261 to 02FF 159 User locations
0231 define         |TempBuffer       ADR  0300 to 030F Buffer for line dump routine
0232 define         |WatchBuffer      ADR  0310 to 0318
0233                |*                ADR  0319 to 03FF Available to User.
0234                |*
0235                |*    SBC65V1B Hardware Equates
0236                |*
```

```
define        |ClearSingleOut  ADR  0406 0407
define        |SingleOutput    ADR  0410
define        |SingleOutputOff ADR  0410
define        |SingleOutputOn  ADR  0418
define        |SingleInput     ADR  0420
              |*
              |*    Uart
              |*
define        |RcvrReg   ADR  0430
define        |XmtReg    ADR  RcvrReg
define        |StatusReg ADR  RcvrReg+1
define        |ModeReg1  ADR  RcvrReg+2
define        |ModeReg2  ADR  ModeReg1
define        |CmdReg    ADR  RcvrReg+3
              |*
              |*    Single Board Computer SYSTEM EQUates
              |*
define        |NonMaskInt  ADR  0FFFA 0FFFB
define        |ResetInt    ADR  0FFFC 0FFFD
define        |SoftwareInt ADR  0FFFE 0FFFF
define        |MaskableInt ADR  0FFFE 0FFFF
              |*
              |*           CODE SECTION
define        |XaMonV4B    ADR  0FD78
define        |            ORG  0F1DE = Start Loc of OBJ Code in SBC Memory
              |*
              |* Note: Labels beginning with lower case letters will NOT
              |* show up in the SYMbol table file!!!
              |*
              |**------------------ PROCEDURES -----------------**
              |*
              |* Invert tense of FLAG
F1DE 49 01    |ToggleFlag   EOR  #01
F1E0 60       |             RTS
              |*
              |* Maximum 30 levels deep
F1E1 85 FC    |Push         STA  HoldAByte  PROCEDURE Push(Acc:BYTE);
F1E3 5A       |             PHY
F1E4 A4 E6    |             LDY  StackPointer
F1E6 A5 FC    |             LDA  HoldAByte
F1E8 91 E4    |             STA  (StackAddress),Y
F1EA E6 E6    |             INC  StackPointer
F1EC 7A       |             PLY
F1ED A5 FC    |             LDA  HoldAByte
F1EF 60       |             RTS
              |*
F1F0 5A       |Pop          PHY  PROCEDURE Pop(VAR Acc:BYTE);
F1F1 A4 E6    |             LDY  StackPointer
F1F3 F0 04    |             BEQ  pSkip
F1F5 C6 E6    |             DEC  StackPointer
F1F7 A4 E6    |             LDY  StackPointer
F1F9 B1 E4    |pSkip        LDA  (StackAddress),Y
F1FB 85 FC    |             STA  HoldAByte
F1FD 7A       |             PLY
F1FE A5 FC    |             LDA  HoldAByte
F200 60       |             RTS
              |*
F201 48       |FastSwap    PHA  71 cycles  swap top 2 stack entries
F202 5A       |            PHY
```

```
F203 A4 E6    |            LDY  StackPointer
F205 88       |            DEY
F206 B1 E4    |            LDA  (StackAddress),Y
F208 48       |            PHA
F209 88       |            DEY
F20A B1 E4    |            LDA  (StackAddress),Y
F20C 85 FD    |            STA  TempLoc
F20E 68       |            PLA
F20F 91 E4    |            STA  (StackAddress),Y
F211 C8       |            INY
F212 A5 FD    |            LDA  TempLoc
F214 91 E4    |            STA  (StackAddress),Y
F216 C8       |            INY
F217 84 E6    |            STY  StackPointer
F219 7A       |            PLY
F21A 68       |            PLA
F21B 60       |            RTS
              |*
F21C 20 E1 F1 |SaveRegs     JSR  Push   save Accum & all other CPU regs
F21F 8A       |             TXA
F220 20 E1 F1 |             JSR  Push   save X
F223 98       |             TYA
F224 20 E1 F1 |             JSR  Push   save Y
F227 08       |             PHP
F228 68       |             PLA
F229 20 E1 F1 |             JSR  Push   save status
F22C 60       |             RTS
              |*
              |* Note:
              |*Equal number of pushes and pops must be executed between
              |*calls to SaveRegs and RestoreRegs,otherwise strange results
              |*may occur.
F22D 20 F0 F1 |RestoreRegs  JSR  Pop
F230 48       |             PHA
F231 20 F0 F1 |             JSR  Pop    restore Y
F234 A8       |             TAY
F235 20 F0 F1 |             JSR  Pop    restore X
F238 AA       |             TAX
F239 20 F0 F1 |             JSR  Pop
F23C 48       |             PHA
F23D 68       |             PLA         restore Accum
F23E 28       |             PLP         restore status
F23F 60       |             RTS
              |*
              |* "Hard Save" registers to Global Vars rather than Userstack
              |*
F240 8D 29 02 |SaveSXYAP    STA  CPUAccumulator
F243 08       |             PHP
F244 68       |             PLA
F245 8D 2A 02 |             STA  CPUStatusReg
F248 8C 28 02 |             STY  CPUYReg
F24B 8E 27 02 |             STX  CPUXReg
F24E 68       |             PLA
F24F 8D 2B 02 |             STA  CPUPgmCounterLo
F252 68       |             PLA
F253 8D 2C 02 |             STA  CPUPgmCounterHi
F256 48       |             PHA
F257 AD 2B 02 |             LDA  CPUPgmCounterLo
F25A 48       |             PHA
```

```
F25B 60       |             RTS
              |*
F25C AD 2A 02 |RestoreSXYA  LDA  CPUStatusReg
F25F 48       |             PHA
F260 AC 28 02 |             LDY  CPUYReg
F263 AE 27 02 |             LDX  CPUXReg
F266 AD 29 02 |             LDA  CPUAccumulator
F269 48       |             PHA
F26A 68       |             PLA
F26B 28       |             PLP
F26C 60       |             RTS
              |*Software timing routines tweeked using a frequency counter
F26D EA       |FastWait      NOP      PROCEDURE FastWait;
F26E EA       |              NOP      BEGIN
F26F EA       |              NOP      END FastWait;
F270 60       |              RTS
              |*
F271 20 6D F2 |TenthMilliSec JSR  FastWait  .0001 Secs
F274 20 6D F2 |              JSR  FastWait
F277 20 6D F2 |              JSR  FastWait
F27A 20 6D F2 |              JSR  FastWait
F27D EA       |              NOP
F27E EA       |              NOP
F27F 60       |              RTS
              |*
F280 20 71 F2 |HalfMilliSec  JSR  TenthMilliSec  .0005 Secs
F283 20 71 F2 |              JSR  TenthMilliSec
F286 20 71 F2 |              JSR  TenthMilliSec
F289 20 71 F2 |              JSR  TenthMilliSec
F28C 20 71 F2 |              JSR  TenthMilliSec
F28F 20 6D F2 |              JSR  FastWait
F292 20 6D F2 |              JSR  FastWait
F295 EA       |              NOP
F296 EA       |              NOP
F297 60       |              RTS
              |*
F298 20 80 F2 |MilliSecDelay JSR  HalfMilliSec   .001 Secs
F29B 20 80 F2 |              JSR  HalfMilliSec
F29E 60       |              RTS
              |*
              |* Enter w/# of mSecs in Y reg  0 to FF Hex
              |*
F29F DA       |MSecDelay     PHX     PROCEDURE MSecDelay(YReg:BYTE);
F2A0 C0 00    |              CPY  #0
F2A2 F0 18    |              BEQ  mSecOut
F2A4 EA       |              NOP
F2A5 C0 01    |              CPY  #1
F2A7 D0 03    |              BNE  delay0
F2A9 4C B7 F2 |              JMP  lastdelay
F2AC 88       |delay0        DEY
F2AD A2 C0    |delay1        LDX  #MSecCount
F2AF CA       |delay2        DEX
F2B0 D0 FD    |              BNE  delay2
F2B2 EA       |              NOP
F2B3 EA       |              NOP
F2B4 88       |              DEY
F2B5 D0 F6    |              BNE  delay1
F2B7 A2 C0    |lastdelay     LDX  #MSecCount
F2B9 CA       |delay3        DEX
```

```
F2BA D0 FD    |              BNE  delay3
F2BC FA       |mSecOut       PLX
F2BD 60       |              RTS
              |*
              |* delay for 1 tenth sec.
F2BE 5A       |TenthSec     PHY           .1Secs
F2BF A0 64    |             LDY  #100Z
F2C1 20 9F F2 |             JSR  MSecDelay          MSecDelay(100);
F2C4 20 98 F2 |             JSR  MilliSecDelay
F2C7 20 98 F2 |             JSR  MilliSecDelay
F2CA 20 80 F2 |             JSR  HalfMilliSec
F2CD 20 71 F2 |             JSR  TenthMilliSec
F2D0 20 71 F2 |             JSR  TenthMilliSec
F2D3 20 71 F2 |             JSR  TenthMilliSec
F2D6 20 71 F2 |             JSR  TenthMilliSec
F2D9 20 71 F2 |             JSR  TenthMilliSec
F2DC 20 6D F2 |             JSR  FastWait
F2DF 20 6D F2 |             JSR  FastWait
F2E2 7A       |             PLY
F2E3 60       |             RTS
              |*
F2E4 DA       |Delay1Second PHX          1 Sec
F2E5 A2 0A    |             LDX  #10Z
F2E7 20 BE F2 |d1SLup       JSR  TenthSec
F2EA CA       |             DEX
F2EB D0 FA    |             BNE  d1SLup
F2ED FA       |             PLX
F2EE 60       |             RTS
              |*
              |*PROCEDURE DelaySeconds(XReg:BYTE );
              |*Enter w/ X Reg:=NumOfSeconds
F2EF 20 E4 F2 |DelaySeconds JSR  Delay1Second
F2F2 CA       |             DEX
F2F3 D0 FA    |             BNE  DelaySeconds
F2F5 60       |             RTS
              |*
              |* Pin 22. Gen Purpose Input.
F2F6 2C 31 04 |ReadDSRBit   BIT  StatusReg  PROCEDURE ReadDSRBit():BOOLEAN ;
F2F9 10 03    |             BPL  dsrIsHi
F2FB A9 00    |             LDA  #FALSE dsr is Lo (*Acc:=Result*)
F2FD 60       |             RTS
F2FE A9 01    |dsrIsHi      LDA  #TRUE  dsr is Hi
F300 60       |             RTS
              |*
              |*U18 Pin 23 Gen Purpose output. Can be used for bank
              |*switching 64K Eprom.
              |*PROCEDURE SetRTSBit(Acc:BOOLEA N); (*Acc:=Level*)
              |*
F301 D0 09    |SetRTSBit    BNE  SetRTSBitHi
F303 AD 33 04 |SetRTSBitLo  LDA  CmdReg
F306 09 20    |             ORA  #BIT5
F308 8D 33 04 |             STA  CmdReg
F30B 60       |             RTS
F30C AD 33 04 |SetRTSBitHi  LDA  CmdReg
F30F 29 DF    |             AND  #0DF
F311 8D 33 04 |             STA  CmdReg
F314 60       |             RTS
              |*
              |* U18 Pin 24 of Gen Purpose output.
```

```
F315 D0 09    |SetDTRBit    BNE  SetDTRBitHi
F317 AD 33 04 |SetDTRBitLo  LDA  CmdReg
F31A 09 02    |             ORA  #BIT1
F31C 8D 33 04 |             STA  CmdReg
F31F 60       |             RTS
F320 AD 33 04 |SetDTRBitHi  LDA  CmdReg
F323 29 FD    |             AND  #0FD
F325 8D 33 04 |             STA  CmdReg
F328 60       |             RTS
              |*
              |* Ck RS232 Port for an incoming CHAR once
              |* RETURN '1' in Accum IF a char present,'0' if not
              |* Pin 16 DCD.L of Uart must be low for receiver to operate,
              |* CTS.L for xmtr. Char is passed in HoldAByte.
              |* PROCEDURE BusyReadRS232(VAR HoldAByte:CHAR;Acc:BO OLEAN);
              |*
F329 AD 31 04 |BusyReadRS232 LDA  StatusReg
F32C 29 02    |              AND  #BIT1
F32E F0 08    |              BEQ  busyout
F330 AD 30 04 |              LDA  RcvrReg
F333 85 FC    |              STA  HoldAByte is used by many I/O Subroutines
F335 A9 01    |              LDA  #TRUE
F337 60       |              RTS
F338 A9 00    |busyout       LDA  #FALSE
F33A 60       |              RTS
              |*
              |*PROCEDURE ReadRS232(VAR HoldAByte:CHAR);
              |*
F33B 20 29 F3 |ReadRS232     JSR  BusyReadRS232
F33E F0 FB    |              BEQ  ReadRS232
F340 A5 FC    |              LDA  HoldAByte
F342 60       |              RTS

              |*Send a char out RS232 port via 'HoldAByte'
              |*PROCEDURE WriteRS232(HoldAByte:C HAR);
              |*
F343 AD 31 04 |WriteRS232 LDA  StatusReg
F346 29 01    |           AND  #BIT0
F348 F0 06    |           BEQ  ckRsOut
F34A A5 FC    |           LDA  HoldAByte
F34C 8D 30 04 |           STA  XmtReg
F34F 60       |           RTS
F350 AD 22 02 |ckRsOut    LDA  WriteTilSent
F353 D0 EE    |           BNE  WriteRS232
F355 60       |           RTS
              |*
F356 A9 14    |UnLockTransmit  LDA  #XOn
F358 85 FC    |                STA  HoldAByte
F35A 20 43 F3 |                JSR  WriteRS232
F35D 60       |                RTS
              |*
              |* BusyRead RS232 port,RETURN TRUE if Esc char detected.
              |*
F35E DA       |QuickEscTest    PHX
F35F A2 20    |                LDX  #020
F361 20 29 F3 |rdLup           JSR  BusyReadRS232
F364 D0 03    |                BNE  qcontCk
F366 CA       |                DEX
F367 D0 F8    |                BNE  rdLup
```

```
F369 A5 FC    |qcontCk         LDA  HoldAByte
F36B C9 9B    |                CMP  #9B
F36D F0 0B    |                BEQ  yesGo
F36F C9 1B    |                CMP  #1B
F371 F0 07    |                BEQ  yesGo
F373 20 56 F3 |                JSR  UnLockTransmit
F376 FA       |noGo            PLX
F377 A9 00    |                LDA  #FALSE
F379 60       |                RTS
F37A FA       |yesGo           PLX
F37B A9 01    |                LDA  #TRUE
F37D 60       |                RTS
              |*
              |* Set Baud Rate of 2651 Uart chip
              |*
F37E A9 CE    |SetBaud  LDA  #MODE1
F380 8D 32 04 |         STA  ModeReg1
F383 AD 24 02 |         LDA  BaudRate
F386 8D 32 04 |         STA  ModeReg2
F389 60       |         RTS 0552               |*
F38A A9 3F    |InitUart  LDA  #3F       19,200 Baud  Regular
F38C 8D 24 02 |          STA  BaudRate
F38F 20 7E F3 |          JSR  SetBaud
F392 A9 05    |          LDA  #ENABLE
F394 8D 33 04 |          STA  CmdReg
F397 AD 30 04 |          LDA  RcvrReg
F39A 60       |          RTS
              |*
F39B A9 00    |InitCkSum LDA  #0          PROCEDURE InitCkSum;
F39D 85 BE    |          STA  Checksum
F39F 85 BF    |          STA  Checksum+1
F3A1 60       |          RTS
              |*  Enter with num to be added in Accum
F3A2 18       |CheckSumAdd CLC
F3A3 65 BE    |            ADC  Checksum
F3A5 85 BE    |            STA  Checksum
F3A7 90 02    |            BCC  skipRdInc
F3A9 E6 BF    |            INC  Checksum+1
F3AB 60       |skipRdInc   RTS
              |*
              |* Calculate Checksum of buffer
              |*
F3AC 20 9B F3 |CheckSum   JSR  InitCkSum
F3AF 20 8B FA |           JSR  ResetBuffer Pointers Forward Reference FW
F3B2 20 1E FA |           JSR  GenEndAdr
F3B5 B1 E0    |csLup      LDA  (BufferPointer1),Y
F3B7 20 A2 F3 |           JSR  CheckSumAdd
F3BA 20 EC FA |           JSR  CkEndOfBuffer
F3BD F0 F6    |           BEQ  csLup
F3BF 20 AA F5 |           JSR  WriteString  FW
F3C2 8D 43 68 |           RASZ 'Checksum= '
F3C5 65 63 6B |
F3C8 73 75 6D |
F3CB 3D 20 00 |
F3CE A9 BE    |           LDA  #Checksum
F3D0 20 4C F7 |           JSR  PrintWord in Hex, JSR WriteCard for Dec
F3D3 60       |           RTS
              |*
F3D4 29 0F    |SetOutput  AND  #0F
```

```
F3D6 8D 34 02 |           STA  OutputDest
F3D9 60       |           RTS
              |*
F3DA 29 0F    |SetInput   AND  #0F
F3DC 8D 31 02 |           STA  InputDest
F3DF 60       |           RTS
              |*
              |* Set Input/Output Devices for GetInput & SendOutput
              |* PROCEDURES
F3E0 8D 32 02 |SetInOut   STA  IOState
F3E3 20 D4 F3 |           JSR  SetOutput
F3E6 AD 32 02 |           LDA  IOState
F3E9 4A       |           LSR
F3EA 4A       |           LSR
F3EB 4A       |           LSR
F3EC 4A       |           LSR
F3ED 20 DA F3 |           JSR  SetInput
F3F0 60       |           RTS
              |*
              |* Example:
              |*
F3F1 A9 77    |SetIOToRS232 LDA #InRSOutRS
F3F3 20 E0 F3 |             JSR  SetInOut
F3F6 60       |             RTS
              |*
F3F7 A9 73    |SetIOToRSInUser1Out LDA  #InRSOutUser1
F3F9 20 E0 F3 |             JSR  SetInOut
F3FC 60       |             RTS
              |*
              |* Single input routine,enter w/bit in Accum,
              |* RETURN BOOLEAN in Accum
              |*
F3FD 5A       |GetSingleInput PHY
F3FE A8       |               TAY
F3FF B9 20 04 |               LDA SingleInput,Y
F402 30 04    |               BMI singleInTrue
F404 A9 00    |               LDA #FALSE BOOLEAN value if bit is 0
F406 7A       |               PLY
F407 60       |               RTS
F408 A9 01    |singleInTrue   LDA #TRUE  BOOLEAN value if bit is 1
F40A 7A       |               PLY
F40B 60       |               RTS
              |*
F40C A2 08    |SInputByte     LDX  #8
F40E A0 07    |               LDY  #7
F410 64 FD    |               STZ  TempLoc
F412 B9 20 04 |inByteLup      LDA  SingleInput,Y
F415 2A       |               ROL
F416 26 FD    |               ROL  TempLoc
F418 88       |sibCkOut       DEY
F419 CA       |               DEX
F41A D0 F6    |               BNE  inByteLup
F41C A5 FD    |               LDA  TempLoc
F41E 60       |               RTS
F41F 38       |rolInPos       SEC
F420 26 FD    |               ROL  TempLoc
F422 80 F4    |               BRA  sibCkOut
              |*
              |* Redirectable Input Routines
```

```
               |*
F424 6C F0 00 |UserIn1   JMP (UserInput1)
F427 6C F2 00 |UserIn2   JMP (UserInput2)
F42A 6C F4 00 |KeyIn     JMP (KeyboardIn)
               |*
               |*Generalized Byte-wide input with Re-direction
               |*PROCEDURE GetInput(VAR HoldAByte:CHAR;Input Dest:I/O);
               |*
F42D AD 31 02 |GetInput  LDA  InputDest
F430 C9 00    |          CMP  #MEMORY1
F432 F0 1E    |          BEQ  GetMem1
F434 C9 01    |          CMP  #MEMORY2
F436 F0 1F    |          BEQ  GetMem2
F438 C9 02    |          CMP  #SINGLEBYTE
F43A F0 20    |          BEQ  GetByteIn
F43C C9 03    |          CMP  #USER1
F43E F0 22    |          BEQ  GetUser1
F440 C9 04    |          CMP  #USER2
F442 F0 22    |          BEQ  GetUser2
F444 C9 06    |          CMP  #KEYBOARD
F446 F0 22    |          BEQ  GetKeyIn
F448 C9 07    |          CMP  #RS232
F44A F0 22    |          BEQ  GetRS232
F44C A9 49    |          LDA  #'I'
F44E 8D 2F 02 |          STA  ErrorMsg
F451 60       |          RTS
F452 B1 E0    |GetMem1   LDA  (BufferPointer1),Y
F454 85 FC    |          STA  HoldAByte
F456 60       |          RTS
F457 B1 E2    |GetMem2   LDA  (BufferPointer2),Y
F459 85 FC    |          STA  HoldAByte
F45B 60       |          RTS
F45C 20 0C F4 |GetByteIn JSR  SInputByte
F45F 85 FC    |          STA  HoldAByte
F461 60       |          RTS
F462 20 24 F4 |GetUser1  JSR  UserIn1
F465 60       |          RTS
F466 20 27 F4 |GetUser2  JSR  UserIn2
F469 60       |          RTS
F46A 20 2A F4 |GetKeyIn  JSR  KeyIn
F46D 60       |          RTS
F46E 20 3B F3 |GetRS232  JSR  ReadRS232
F471 60       |          RTS
               |*
               |* Single Output
               |*
F472 5A       |AssertSingleOut  PHY
F473 A8       |                 TAY
F474 B9 10 04 |                 LDA  SingleOutput,Y
F477 7A       |                 PLY
F478 60       |                 RTS
               |*
F479 8A       |TurnBitOff      TXA
F47A 20 72 F4 |SOutOff         JSR  AssertSingleOut
F47D 60       |                RTS
               |*
F47E 8A       |TurnBitOn       TXA
F47F 18       |SOutOn          CLC
F480 69 08    |                ADC  #8
```

```
F482 20 72 F4 |               JSR  AssertSingleOut
F485 60       |               RTS
              |*
F486 85 FC    |SOutputByte    STA  HoldAByte
F488 DA       |               PHX
F489 5A       |               PHY
F48A A2 07    |               LDX  #7
F48C A0 08    |               LDY  #8
F48E A5 FC    |byteLup        LDA  HoldAByte
F490 2A       |               ROL
F491 85 FC    |               STA  HoldAByte
F493 B0 0B    |               BCS  turnBitOn
F495 20 79 F4 |               JSR  TurnBitOff
F498 CA       |innrLup        DEX
F499 88       |               DEY
F49A D0 F2    |               BNE  byteLup
F49C 2A       |               ROL
F49D 7A       |               PLY
F49E FA       |               PLX
F49F 60       |               RTS
F4A0 20 7E F4 |turnBitOn      JSR TurnBitOn
F4A3 80 F3    |               BRA  innrLup
              |*
              |* Redirectable Output Routines
              |*
F4A5 6C F6 00 |UserOut1    JMP  (UserOutput1)
F4A8 6C F8 00 |UserOut2    JMP  (UserOutput2)
F4AB 6C FA 00 |ScreenOut   JMP  (Screen)
              |*
              |* Generalized Output with Re-direction
              |* PROCEDURE SendOutput(HoldAByte:C HAR;Output Dest:I/O);
              |*
F4AE AD 34 02 |SendOutput LDA  OutputDest
F4B1 C9 00    |           CMP  #MEMORY1
F4B3 F0 1E    |           BEQ  SendMem1
F4B5 C9 01    |           CMP  #MEMORY2
F4B7 F0 1F    |           BEQ  SendMem2
F4B9 C9 02    |           CMP  #SINGLEBYTE
F4BB F0 20    |           BEQ  SendByte
F4BD C9 03    |           CMP  #USER1
F4BF F0 22    |           BEQ  SendUser1
F4C1 C9 04    |           CMP  #USER2
F4C3 F0 22    |           BEQ  SendUser2
F4C5 C9 05    |           CMP  #SCREEN
F4C7 F0 22    |           BEQ  SendScreen
F4C9 C9 07    |           CMP  #RS232
F4CB F0 22    |           BEQ  SendRS232
F4CD A9 4F    |           LDA  #'O'
F4CF 8D 2F 02 |           STA  ErrorMsg
F4D2 60       |           RTS
F4D3 A5 FC    |SendMem1   LDA  HoldAByte
F4D5 91 E0    |           STA  (BufferPointer1),Y
F4D7 60       |           RTS
F4D8 A5 FC    |SendMem2   LDA  HoldAByte
F4DA 91 E2    |           STA  (BufferPointer2),Y
F4DC 60       |           RTS
F4DD A5 FC    |SendByte   LDA  HoldAByte
F4DF 20 86 F4 |           JSR  SOutputByte
F4E2 60       |           RTS
```

```
F4E3 20 A5 F4 |SendUser1  JSR  UserOut1
F4E6 60       |           RTS
F4E7 20 A8 F4 |SendUser2  JSR  UserOut2
F4EA 60       |           RTS
F4EB 20 AB F4 |SendScreen JSR  ScreenOut
F4EE 60       |           RTS
F4EF 20 43 F3 |SendRS232  JSR  WriteRS232
F4F2 60       |           RTS
              |*
F4F3 20 2D F4 |Wait4Char  JSR  GetInput
F4F6 A5 FC    |           LDA  HoldAByte
F4F8 29 7F    |           AND  #7F
F4FA CD 33 02 |           CMP  Key
F4FD F0 02    |           BEQ  wOut
F4FF D0 F2    |           BNE  Wait4Char
F501 60       |wOut       RTS
              |*
F502 AD 33 02 |Wait4Space LDA  Key
F505 48       |           PHA
F506 A9 20    |           LDA  #' '
F508 8D 33 02 |           STA  Key
F50B 20 F3 F4 |           JSR  Wait4Char
F50E 68       |           PLA
F50F 8D 33 02 |           STA  Key
F512 60       |           RTS
              |*
              |* Pass Ascii byte in ACCumulator.
              |* RETURN TRUE IF =Space,CReturn,OR Y
              |*
F513 29 7F    |SCROrYesCk AND  #7F
F515 C9 59    |           CMP  #'Y'
F517 F0 0F    |           BEQ  scr1
F519 C9 79    |           CMP  #'y'
F51B F0 0B    |           BEQ  scr1
F51D C9 0D    |           CMP  #CReturn
F51F F0 07    |           BEQ  scr1
F521 C9 20    |           CMP  #Space
F523 F0 03    |           BEQ  scr1
F525 A9 00    |           LDA  #FALSE
F527 60       |           RTS
F528 A9 01    |scr1       LDA  #TRUE
F52A 60       |           RTS
              |*
F52B A9 00    |DisableXOnHandShake LDA #FALSE
F52D 8D 23 02 |           STA  XOnFlag
F530 60       |           RTS
              |*
F531 A9 01    |EnableXOnHandShake LDA #TRUE
F533 8D 23 02 |           STA  XOnFlag
F536 60       |           RTS
              |*
              |* Use "GetInput" to Rcv an 'XOn'
              |* PROCEDURE ReadChar(VAR HoldAByte:CHAR);
              |*
F537 20 2D F4 |ReadChar   JSR  GetInput
F53A A5 FC    |           LDA  HoldAByte
F53C 29 7F    |           AND  #7F
F53E C9 14    |           CMP  #XOn
F540 F0 F5    |           BEQ  ReadChar ignore XOn chars from JTerm-inal
```

```
F542 85 FC    |           STA  HoldAByte
F544 60       |           RTS
              |*
              |*Send one char to output device. May create erratic behavior
              |*on non Ascii chars, when sending to a terminal.
              |* PROCEDURE WriteChar(Acc:CHAR;XO nFlag:BOOLEAN);
              |*
F545 48       |WriteChar PHA
F546 AD 23 02 |          LDA  XOnFlag
F549 F0 03    |          BEQ  wchSkip     If handshake on
F54B 20 F3 F4 |          JSR  Wait4Char    THEN wait for XOn char
F54E 68       |wchSkip   PLA               ELSE don't
F54F 85 FC    |          STA  HoldAByte   END;
F551 20 AE F4 |          JSR  SendOutput
F554 A5 FC    |          LDA  HoldAByte
F556 60       |          RTS
              |*
              |* Wait for an Esc or a SpaceBar
              |*
F557 20 37 F5 |SpaceEscWait JSR  ReadChar
F55A C9 1B    |             CMP  #Escape
F55C F0 04    |             BEQ  tSkp
F55E C9 20    |             CMP  #Space
F560 D0 F5    |             BNE  SpaceEscWait
F562 60       |tSkp         RTS
              |*
              |* Enter w/char in 'Acc' & # of chars in 'X'
              |*
F563 20 45 F5 |SendChars    JSR  WriteChar
F566 CA       |             DEX
F567 D0 FA    |             BNE  SendChars
F569 60       |             RTS
              |*
              |* Send Carriage Return
F56A A9 0D    |WriteLn      LDA  #CReturn
F56C 20 45 F5 |             JSR  WriteChar
F56F 60       |             RTS
              |*
              |* Send CR LineFeed
              |*
F570 20 6A F5 |SendCRLF     JSR  WriteLn
F573 A9 0A    |SendLNFD     LDA  #LineFeed
F575 20 45 F5 |             JSR  WriteChar
F578 60       |             RTS
              |*
F579 A9 03    |ReleaseHost  LDA  #ControlC
F57B 20 45 F5 |             JSR  WriteChar
F57E 60       |             RTS
              |*
              |* Clear dumb terminal screen
              |*
F57F A9 1A    |ClearScreen  LDA  #ControlZ
F581 85 FC    |             STA  HoldAByte
F583 20 43 F3 |             JSR  WriteRS232
F586 60       |             RTS
              |*
F587 48       |SendABarSave PHA
F588 A9 7C    |             LDA  #'|'
F58A 20 45 F5 |             JSR  WriteChar
```

```
F58D 68       |             PLA
F58E 60       |             RTS
              |*
F58F A9 2D    |SendADash    LDA  #'-'
F591 20 45 F5 |             JSR  WriteChar
F594 60       |             RTS
              |*
F595 A9 20    |SendASpace   LDA  #' '
F597 20 45 F5 |             JSR  WriteChar
F59A 60       |             RTS
              |*
F59B A2 03    |Send3Spaces  LDX #3
F59D A9 20    |             LDA #Space
F59F 20 63 F5 |             JSR  SendChars
F5A2 60       |             RTS
              |*
              |* Initialize dumb terminal
              |*
F5A3 20 8A F3 |InitTerminal JSR  InitUart
F5A6 20 7F F5 |             JSR  ClearScreen
F5A9 60       |             RTS
              |*
              |* Send message string to Currently Installed Output Device
              |*                       CIOD. Weird results if you forget 0
              |*                       at end of your string! See PrintID
              |* PROCEDURE WriteString(str:ARRAY OF CHAR);
              |*
F5AA 68       |WriteString PLA         pull Pgm Counter Lo off stack
F5AB 85 EA    |            STA  TempBufferPtr
F5AD 68       |            PLA         pull Pgm Counter Hi off stack
F5AE 85 EB    |            STA  TempBufferPtr+1
F5B0 5A       |            PHY                     save Y reg
F5B1 A0 00    |            LDY  #0
F5B3 E6 EA    |loopR       INC  TempBufferPtr     Inc PC lo
F5B5 D0 02    |            BNE  skipAdr
F5B7 E6 EB    |            INC  TempBufferPtr+1   Inc PC hi
F5B9 B1 EA    |skipAdr     LDA  (TempBufferPtr),Y
F5BB F0 07    |            BEQ  msgRTS            hex 0 ends procedure
F5BD 20 45 F5 |            JSR  WriteChar         send char out
F5C0 A9 01    |            LDA  #1
F5C2 D0 EF    |            BNE  loopR
F5C4 7A       |msgRTS      PLY                     restore Y reg
F5C5 A5 EB    |            LDA  TempBufferPtr+1   restore altered Program
F5C7 48       |            PHA                     Counter to CPU Stack
F5C8 A5 EA    |            LDA  TempBufferPtr
F5CA 48       |            PHA
F5CB 60       |            RTS
              |*
F5CC 20 AA F5 |PrintId   JSR  WriteString
F5CF 8D 4A 43 |          RASZ  'JComm LAB uCEL(TM) XaMonV4B-1'
F5D2 6F 6D 6D |
F5D5 20 4C 41 |
F5D8 42 20 75 |
F5DB 43 45 4C |
F5DE 28 54 4D |
F5E1 29 20 58 | 0933 F5E4 61 4D 6F |
F5E7 6E 56 34 | 0933 F5EA 42 2D 31 |
F5ED 00       | 0934 F5EE 60       |          RTS
              |*
```

```
F5EF 20 AA F5 |DoneMessage  JSR  WriteString
F5F2 8D 44 6F |             RASZ 'Done'
F5F5 6E 65 00 |
F5F8 60       |             RTS
              |*
F5F9 20 AA F5 |PressSpaceBarMsg JSR  WriteString
F5FC 8D 50 72 |                 RASZ 'Press Space Key'
F5FF 65 73 73 |
F602 20 53 70 |
F605 61 63 65 |
F608 20 4B 65 |
F60B 79 00    |
F60D 60       |                 RTS
              |*
F60E 20 AA F5 |GetYResponse     JSR  WriteString
F611 28 79 65 |                 ASZ  '(yes)'
F614 73 29 00 |
F617 20 37 F5 |                 JSR  ReadChar
F61A 20 13 F5 |                 JSR  SCROrYesCk
F61D 20 E1 F1 |                 JSR  Push
F620 D0 09    |                 BNE  yesMsg
F622 A9 6E    |noMg             LDA  #'n'
F624 20 45 F5 |                 JSR  WriteChar
F627 20 F0 F1 |                 JSR  Pop
F62A 60       |                 RTS
F62B A9 79    |yesMsg           LDA  #'y'
F62D 20 45 F5 |                 JSR  WriteChar
F630 20 F0 F1 |                 JSR  Pop
F633 60       |                 RTS
              |*
F634 20 AA F5 |GetNoResponse JSR  WriteString
F637 28 6E 6F |              ASZ  '(no)'
F63A 29 00    |
F63C 20 37 F5 |              JSR  ReadChar
F63F 20 A3 F6 |              JSR  LwrCase2UprCAsc
F642 C9 59    |              CMP  #'Y'
F644 F0 08    |              BEQ  neg
F646 A9 6E    |pos           LDA  #'n'
F648 20 45 F5 |              JSR  WriteChar
F64B A9 01    |              LDA  #TRUE
F64D 60       |              RTS
F64E A9 79    |neg           LDA  #'y'
F650 20 45 F5 |              JSR  WriteChar
F653 A9 00    |              LDA  #FALSE
F655 60       |              RTS
              |*
              |* Definitions:
              |*Ascii char :A 1-byte code that represents a CHARacter
              |*AlphaAscii :Printable alphabetic Ascii character 'A'..'Z'
              |*NumAscii   :Printable numeric Ascii character '0'..'9'
              |*AlphaNumAsc:An AlphaAscii or NumAscii character
              |*CntrlAscii :Non-printable ascii chars EX: 'escape' = 1BH
              |*OtherAscii :A printable nonAlphaNumeric ascii character
              |*HexAscii  :AlphaNumAsc within range '0'..'9' or 'A'..'F'
              |*Hex Nibble :Representation of a 4-bit binary value, range
              |*            0 - F. EX: 0111B(binary) = 7H(hex)
              |*Hex Byte  :Representation of an 8-bit binary value, range
              |*            00 - FF. EX: 00000111B = 07H (BYTE=CHAR)
              |*
```

```
                |*Enter w/ Acc=char, RETURN TRUE in Acc IF NumAscii
                |*
F656 29 7F      |IsItAscii09 AND  #7F
F658 C9 30      |            CMP  #'0'    Is data < '0' ?
F65A 90 04      |            BCC  not09   Yes, so RETURN FALSE & quit
F65C C9 3A      |            CMP  #':'    Is data < ':' ?
F65E 90 03      |            BCC  is09    Yes,so it is within Ascii '0'..'9'
F660 A9 00      |not09       LDA  #FALSE  RETURN Acc:= 0
F662 60         |            RTS
F663 A9 01      |is09        LDA  #TRUE   RETURN Acc:= 1
F665 60         |            RTS
                |*
                |*Enter w/ Acc=char,RETURN TRUE in Acc IF AlphaAscii 'A'..'F'
                |*
F666 29 7F      |IsItAsciiAF AND  #7F
F668 C9 41      |         CMP  #'A'
F66A 90 04      |         BCC  notAF PROCEDURE IsItAsciiAF(Acc:CHAR):BOOLEAN;
F66C C9 47      |         CMP  #'G'  BEGIN
F66E 90 03      |         BCC  isAF   ForceBit8ToZero;
F670 A9 00      |notAF    LDA  #FALSE IF char < 'A' THEN RETURN FALSE; END;
F672 60         |         RTS         IF char < 'G' THEN RETURN TRUE;  END;
F673 A9 01      |isAF     LDA  #TRUE END IsItAsciiAF;
F675 60         |         RTS
                |*
                |*Enter w/ Acc=char, RETURN TRUE in Acc IF HexAscii
                |*
F676 48         |IsItAscii0F PHA
F677 20 56 F6   |            JSR  IsItAscii09
F67A D0 09      |            BNE  itIs
F67C 68         |            PLA
F67D 20 66 F6   |            JSR  IsItAsciiAF
F680 D0 04      |            BNE  itIsB
F682 A9 00      |            LDA  #FALSE
F684 60         |            RTS
F685 68         |itIs        PLA
F686 A9 01      |itIsB       LDA  #TRUE  RETURN A '1' IF Ascii
F688 60         |            RTS         between '0'..'F'
                |*
                |*Enter w/ Acc=char, RETURN TRUE IF printable Ascii CHAR
                |*
F689 C9 1F      |IsItAscii    CMP  #1F
F68B F0 13      |             BEQ  notPrintable
F68D 90 11      |             BCC  notPrintable  <SPACE?
F68F C9 7F      |             CMP  #7F
F691 90 0A      |             BCC  yesPrintable  <DEL?
F693 C9 9F      |             CMP  #9F
F695 F0 09      |             BEQ  notPrintable
F697 90 07      |             BCC  notPrintable  <SPACE?
F699 C9 FF      |             CMP  #0FF
F69B F0 03      |             BEQ  notPrintable
F69D A9 01      |yesPrintable LDA  #TRUE
F69F 60         |             RTS
F6A0 A9 00      |notPrintable LDA  #FALSE
F6A2 60         |             RTS
                |*
                |*Convert lower case AlphaAscii to upper case
                |*
F6A3 C9 61      |LwrCase2UprCAsc CMP  #'a'  IF ACC <Ascii 'a'?
F6A5 90 07      |                BCC  lcOut THEN do nothing
```

```
F6A7 C9 7B    |                CMP  #'z'+1
F6A9 B0 03    |                BCS  lcOut IF ACC < 'z'
F6AB 38       |                SEC
F6AC E9 20    |                SBC  #20   THEN convert to Upr case
F6AE 60       |lcOut           RTS
              |*
              |*Enter w/ Acc=HexAscii, RETURN lower HEX Nibble IN Acc
              |*
F6AF 48       |HexAsciiToHexNibble PHA
F6B0 20 56 F6 |              JSR  IsItAscii09
F6B3 D0 09    |              BNE  ascMask1
F6B5 68       |              PLA
F6B6 48       |              PHA
F6B7 20 66 F6 |              JSR  IsItAsciiAF
F6BA D0 06    |              BNE  ascMask2
F6BC 68       |              PLA
F6BD 60       |              RTS
F6BE 68       |ascMask1      PLA
F6BF 29 0F    |              AND  #0F
F6C1 60       |              RTS
F6C2 68       |ascMask2      PLA
F6C3 29 0F    |              AND  #0F
F6C5 18       |              CLC
F6C6 69 09    |              ADC  #09
F6C8 60       |              RTS
              |*
              |*Enter w/Acc=lower Hex Nibble, RETURN HexAscii in Acc
              |*
F6C9 29 0F    |HexNibbleToHexAscii AND  #0F
F6CB C9 0A    |              CMP  #0A
F6CD 90 06    |              BCC  hexUP  YES,0-9
F6CF E9 09    |              SBC  #09    NO,IT'S A-F
F6D1 18       |              CLC
F6D2 69 40    |              ADC  #'@'
F6D4 60       |              RTS
F6D5 18       |hexUP         CLC
F6D6 69 30    |              ADC  #'0'
F6D8 60       |              RTS
              |*
              |* RETURN a NumAscii char '0'..'9' via HoldAByte and Acc
              |* filter out all other chars
              |*
F6D9 20 37 F5 |GetAsciiNumber JSR  ReadChar
F6DC 20 56 F6 |               JSR  IsItAscii09
F6DF F0 F8    |               BEQ  GetAsciiNumber
F6E1 A5 FC    |               LDA  HoldAByte
F6E3 60       |               RTS
              |*
              |* RETURN a HexAscii char via HoldAByte and Acc,
              |* filter out all other chars
              |*
F6E4 20 37 F5 |GetHexAscii JSR  ReadChar
F6E7 20 76 F6 |            JSR  IsItAscii0F
F6EA F0 F8    |            BEQ  GetHexAscii
F6EC A5 FC    |            LDA  HoldAByte
F6EE 60       |            RTS
              |*
F6EF 20 D9 F6 |GetSingleHexNum JSR GetAsciiNumber
F6F2 20 1D F7 |                JSR PrintByte
```

```
F6F5 20 AF F6 |                 JSR HexAsciiToHexNibble    FR
F6F8 48       |                 PHA
F6F9 20 95 F5 |                 JSR SendASpace
F6FC 68       |                 PLA
F6FD 60       |                 RTS
              |*
              |* get 2 HexAscii chars, RETURN w/Acc=Hex Byte
              |*
F6FE 20 E4 F6 |GetHexByte  JSR  GetHexAscii
F701 20 45 F5 |            JSR  WriteChar
F704 20 AF F6 |            JSR  HexAsciiToHexNibble
F707 48       |            PHA                 Hi hex nibble
F708 20 E4 F6 |            JSR  GetHexAscii
F70B 20 45 F5 |            JSR  WriteChar
F70E 20 AF F6 |            JSR  HexAsciiToHexNibble
F711 85 FC    |            STA  HoldAByte     Lo hex nibble
F713 68       |            PLA
F714 0A       |            ASL
F715 0A       |            ASL
F716 0A       |            ASL
F717 0A       |            ASL
F718 05 FC    |            ORA  HoldAByte
F71A 85 FC    |            STA  HoldAByte
F71C 60       |            RTS
              |*
              |*Enter w/Acc=Hex Byte, send out 2 HexAscii chars to CIOD
              |*
F71D 48       |PrintByte   PHA
F71E 29 F0    |            AND  #0F0
F720 4A       |            LSR
F721 4A       |            LSR
F722 4A       |            LSR
F723 4A       |            LSR
F724 20 C9 F6 |            JSR  HexNibbleToHexAscii
F727 20 45 F5 |            JSR  WriteChar
F72A 68       |            PLA
F72B 20 C9 F6 |PrintNibl   JSR  HexNibbleToHexAscii Enter w/Acc=Hex Nibble
F72E 20 45 F5 |            JSR  WriteChar      Send out ascii char, Acc OK
F731 60       |            RTS
              |*
              |*Enter w/Acc=Hex Byte, send out 2 HexAscii, restore Acc
              |*
F732 48       |PrintByteSave PHA
F733 20 1D F7 |              JSR  PrintByte
F736 68       |              PLA
F737 60       |              RTS
              |*
F738 85 E8    |GetZPWord STA  TempPointer
F73A 5A       |          PHY                     Save Y Reg
F73B A0 00    |          LDY  #0                 Y offset:=0
F73D 84 E9    |          STY  TempPointer+1      High byte tempointer:=0
F73F B1 E8    |          LDA  (TempPointer),Y
F741 20 E1 F1 |          JSR  Push               Save Low-byte 1
F744 C8       |          INY
F745 B1 E8    |          LDA  (TempPointer),Y
F747 20 E1 F1 |          JSR  Push               Save High-byte
F74A 7A       |          PLY                     Restore Y Reg
F74B 60       |          RTS
              |*
```

```
               |*Enter w/Acc=#low byte of zero page var, send out 4 HexAscii
               |*chars to CIOD
F74C 20 38 F7 |PrintWord JSR  GetZPWord
F74F 20 F0 F1 |          JSR  Pop
F752 20 1D F7 |          JSR  PrintByte          Print High-byte
F755 20 F0 F1 |          JSR  Pop
F758 20 1D F7 |          JSR  PrintByte          Print Low-byte
F75B 60       |          RTS
              |* Print current buffer address in Hex TCIOD
F75C A9 E0    |PrintSBCAdrs  LDA  #BufferPointer1
F75E 20 4C F7 |              JSR  PrintWord
F761 60       |              RTS
              |*
              |* This proc can be improved
              |*
F762 EE 00 03 |IncCardNumber INC  TempBuffer
F765 AD 00 03 |              LDA  TempBuffer
F768 C9 0A    |              CMP  #0A
F76A F0 01    |              BEQ  IncTens
F76C 60       |              RTS
F76D A9 00    |IncTens       LDA  #0
F76F 8D 00 03 |              STA  TempBuffer
F772 EE 01 03 |              INC  TempBuffer+1
F775 AD 01 03 |              LDA  TempBuffer+1
F778 C9 0A    |              CMP  #0A
F77A F0 01    |              BEQ  IncHundreds
F77C 60       |              RTS
F77D A9 00    |IncHundreds   LDA  #0
F77F 8D 01 03 |              STA  TempBuffer+1
F782 EE 02 03 |              INC  TempBuffer+2
F785 AD 02 03 |              LDA  TempBuffer+2
F788 C9 0A    |              CMP  #0A
F78A F0 01    |              BEQ  IncThousands
F78C 60       |              RTS
F78D A9 00    |IncThousands  LDA  #0
F78F 8D 02 03 |              STA  TempBuffer+2
F792 EE 03 03 |              INC  TempBuffer+3
F795 AD 03 03 |              LDA  TempBuffer+3
F798 C9 0A    |              CMP  #0A
F79A F0 01    |              BEQ  IncTenThou
F79C 60       |              RTS
F79D A9 00    |IncTenThou    LDA  #0
F79F 8D 03 03 |              STA  TempBuffer+3
F7A2 EE 04 03 |              INC  TempBuffer+4
F7A5 AD 04 03 |              LDA  TempBuffer+4
F7A8 C9 0A    |              CMP  #0A
F7AA F0 01    |              BEQ  incErr
F7AC 60       |              RTS
F7AD A9 00    |incErr        LDA  #0
F7AF 8D 04 03 |              STA  TempBuffer+4
F7B2 A9 4F    |              LDA  #'O'
F7B4 8D 05 03 |              STA  TempBuffer+5
F7B7 60       |              RTS
              |*
F7B8 A0 05    |SkipLeadingZeroes LDY  #5
F7BA B1 EA    |skipLZLup         LDA  (TempBufferPtr),Y
F7BC F0 01    |                  BEQ  decAndDo
F7BE 60       |                  RTS
F7BF 20 95 F5 |decAndDo          JSR  SendASpace
```

```
F7C2 88       |                  DEY
F7C3 D0 F5    |                  BNE  skipLZLup
F7C5 60       |                  RTS
              |*
F7C6 20 F1 FF |PrintCardArray    JSR  InitTempBuffer
F7C9 20 B8 F7 |                  JSR  SkipLeadingZeroes
F7CC B1 EA    |pcNumLup       LDA  (TempBufferPtr),Y
F7CE 20 C9 F6 |               JSR  HexNibbleToHexAscii
F7D1 20 45 F5 |               JSR  WriteChar
F7D4 88       |               DEY
F7D5 10 F5    |               BPL  pcNumLup
F7D7 60       |               RTS
              |* Clear Temporary Buffer Locations
F7D8 20 F1 FF |ClearTempBufLocs JSR  InitTempBuffer
F7DB A2 10    |                 LDX  #10
F7DD A0 00    |                 LDY  #0
F7DF 91 EA    |clrTLup          STA  (TempBufferPtr),Y
F7E1 C8       |                 INY
F7E2 CA       |                 DEX
F7E3 D0 FA    |                 BNE  clrTLup
F7E5 A0 00    |                 LDY  #0
F7E7 60       |                 RTS
              |*
              |*Enter w/ Acc=HEX Byte, output is ascii rep in Tempbuffer
              |*
F7E8 F0 0B    |CountByteDec BEQ  cBOut
F7EA 8D 25 02 |             STA  Count
F7ED 20 62 F7 |pDecBLup     JSR  IncCardNumber
F7F0 CE 25 02 |             DEC  Count
F7F3 D0 F8    |             BNE  pDecBLup
F7F5 60       |cBOut       RTS
              |*
F7F6 DA       |WriteByteDec PHX
F7F7 5A       |             PHY
F7F8 48       |             PHA
F7F9 20 D8 F7 |             JSR  ClearTempBufLocs
F7FC 68       |             PLA
F7FD 20 E8 F7 |             JSR  CountByteDec
F800 20 C6 F7 |             JSR  PrintCardArray
F803 7A       |             PLY
F804 FA       |             PLX
F805 60       |             RTS
              |*
              |*Enter w/Acc= #zero page low Hex Byte, send out 16 bit card
              |*to CIOD
F806 DA       |WriteCard    PHX       PROCEDURE WriteCard(#ZeroPageVar) ;
F807 5A       |             PHY
F808 20 38 F7 |             JSR  GetZPWord
F80B 20 F0 F1 |             JSR  Pop
F80E 85 BD    |             STA  TotalCount+1
F810 20 F0 F1 |             JSR  Pop
F813 85 BC    |             STA  TotalCount
F815 20 D8 F7 |             JSR  ClearTempBufLocs
F818 A5 BD    |             LDA  TotalCount+1
F81A F0 29    |             BEQ  wcOutLo
F81C A5 BC    |wcLup        LDA  TotalCount
F81E 38       |             SEC
F81F E9 0A    |             SBC  #0A
F821 85 BC    |             STA  TotalCount
```

```
F823 08       |             PHP
F824 20 6D F7 |             JSR  IncTens
F827 28       |             PLP
F828 B0 F2    |             BCS  wcLup
F82A A5 BD    |             LDA  TotalCount+1
F82C 38       |             SEC
F82D E9 01    |             SBC  #1
F82F 85 BD    |             STA  TotalCount+1
F831 08       |             PHP
F832 A5 BD    |             LDA  TotalCount+1
F834 F0 03    |             BEQ  wcLupOut
F836 28       |             PLP
F837 B0 E3    |             BCS  wcLup
F839 28       |wcLupOut     PLP
F83A A5 BC    |             LDA  TotalCount
F83C 20 E8 F7 |             JSR  CountByteDec
F83F 20 C6 F7 |             JSR  PrintCardArray
F842 7A       |             PLY
F843 FA       |             PLX
F844 60       |             RTS
F845 A5 BC    |wcOutLo      LDA  TotalCount
F847 20 F6 F7 |             JSR  WriteByteDec
F84A 7A       |             PLY
F84B FA       |             PLX
F84C 60       |             RTS
              |*
F84D 20 AA F5 |HexToDecimal JSR  WriteString
F850 8D 48 65 |             RASZ 'Hex to Dec> '
F853 78 20 74 |
F856 6F 20 44 |
F859 65 63 3E |
F85C 20 00    |
F85E 20 FE F6 |             JSR  GetHexByte
F861 85 BD    |             STA  TotalCount+1
F863 20 FE F6 |             JSR  GetHexByte
F866 85 BC    |             STA  TotalCount
F868 A9 BC    |             LDA  #TotalCount
F86A 20 06 F8 |             JSR  WriteCard
F86D 60       |             RTS
              |*
              |* Send out last program counter reading
              |*
F86E 20 AA F5 |PrintPC   JSR  WriteString
F871 8D 50 43 |          RASZ 'PC= '
F874 3D 20 00 |
F877 AD 2C 02 |          LDA  CPUPgmCounterHi
F87A 20 1D F7 |          JSR  PrintByte  PC hi
F87D AD 2B 02 |          LDA  CPUPgmCounterLo
F880 20 1D F7 |          JSR  PrintByte  PC lo
F883 60       |          RTS
              |*
              |* Send out CPU's Registers
              |*
F884 20 40 F2 |PrintRegs JSR  SaveSXYAP
F887 AD 21 02 |          LDA  TraceFlag
F88A D0 04    |          BNE  pRegCont
F88C 20 5C F2 |          JSR  RestoreSXYA
F88F 60       |          RTS
F890 20 6E F8 |pRegCont  JSR  PrintPC
```

```
F893 20 AA F5 |          JSR  WriteString
F896 20 20 53 |          ASZ  '  Stat= '
F899 74 61 74 |
F89C 3D 20 00 |
F89F AD 2A 02 |          LDA  CPUStatusReg
F8A2 20 1D F7 |          JSR  PrintByte
F8A5 20 AA F5 |          JSR  WriteString
F8A8 20 20 58 |          ASZ  '  X= '
F8AB 3D 20 00 |
F8AE AD 27 02 |          LDA  CPUXReg
F8B1 20 1D F7 |          JSR  PrintByte
F8B4 20 AA F5 |          JSR  WriteString
F8B7 20 20 59 |          ASZ  '  Y= '
F8BA 3D 20 00 |
F8BD AD 28 02 |          LDA  CPUYReg
F8C0 20 1D F7 |          JSR  PrintByte
F8C3 20 AA F5 |          JSR  WriteString
F8C6 20 20 41 |          ASZ  '  A= '
F8C9 3D 20 00 |
F8CC AD 29 02 |          LDA  CPUAccumulator
F8CF 20 1D F7 |          JSR  PrintByte
F8D2 20 AA F5 |          JSR  WriteString
F8D5 20 20 54 |          ASZ  '  Trace = '
F8D8 72 61 63 |
F8DB 65 20 3D |
F8DE 20 00    |
F8E0 AD 37 02 |          LDA  TraceVar
F8E3 20 1D F7 |          JSR  PrintByte
F8E6 20 6A F5 |          JSR  WriteLn
F8E9 20 5C F2 |          JSR  RestoreSXYA
F8EC 60       |          RTS
              |*
F8ED AD 2A 02 |XaDebugRoutine LDA  CPUStatusReg
F8F0 48       |            PHA
F8F1 28       |            PLP
F8F2 20 5C F2 |            JSR  RestoreSXYA
F8F5 20 84 F8 |            JSR  PrintRegs
F8F8 20 40 F2 |            JSR  SaveSXYAP
F8FB 20 AA F5 |            JSR  WriteString
F8FE 8D 47 6F |            RASZ 'Go to Monitor? '
F901 20 74 6F |
F904 20 4D 6F |
F907 6E 69 74 |
F90A 6F 72 3F |
F90D 20 00    |
F90F 20 34 F6 |            JSR  GetNoResponse
F912 F0 01    |            BEQ  wayOut
F914 60       |            RTS
F915 4C 8B FD |wayOut      JMP  cmdI
              |*
F918 6C EE 00 |DebugRoutine JMP  (Debug)  init'ed w/addr of XaDebugRoutine
              |*
              |* RETURN TRUE in Acc if interrupt was a BRK, FALSE if IRQ
              |*
F91B AD 2A 02 |IsItSWI     LDA  CPUStatusReg
F91E 48       |            PHA
F91F 28       |            PLP
F920 29 10    |            AND #BIT4    Brk flag set?
F922 D0 03    |            BNE itIsSWI
```

```
F924 A9 00    |            LDA #FALSE   No, it is an IRQ
F926 60       |            RTS
F927 A9 01    |itIsSWI     LDA #TRUE    Yes, it is an BRK
F929 60       |            RTS
              |*
F92A 20 40 F2 |NMIRoutine  JSR  SaveSXYAP
F92D 68       |            PLA
F92E 8D 2A 02 |            STA  CPUStatusReg
F931 68       |            PLA
F932 8D 2B 02 |            STA  CPUPgmCounterLo
F935 68       |            PLA
F936 8D 2C 02 |            STA  CPUPgmCounterHi
F939 20 F1 F3 |            JSR  SetIOToRS232
F93C 20 6E F8 |            JSR  PrintPC
F93F 20 F9 F5 |            JSR  PressSpaceBarMsg
F942 20 02 F5 |            JSR  Wait4Space
F945 4C 78 FD |            JMP  XaMonV4B
              |*
F948 20 AA F5 |SWIRoutine  JSR  WriteString
F94B 8D 53 57 |            RASZ  'SWI Interrupt @ PC= '
F94E 49 20 49 |
F951 6E 74 65 |
F954 72 72 75 |
F957 70 74 20 |
F95A 40 20 50 |
F95D 43 3D 20 |
F960 00       |
F961 A5 EC    |            LDA  UserPC  adjust pc lo-2
F963 38       |            SEC
F964 E9 02    |            SBC  #2
F966 85 EC    |            STA  UserPC
F968 B0 02    |            BCS  swiSkip
F96A C6 ED    |            DEC  UserPC+1
F96C A9 EC    |swiSkip     LDA  #UserPC

F96E 20 4C F7 |            JSR  PrintWord
F971 AD 1E 02 |            LDA  DebugFlag
F974 D0 03    |            BNE  doDebug     BEQ intOut
F976 4C 90 FF |            JMP  intOut      unstructured EXIT
F979 20 18 F9 |doDebug     JSR  DebugRoutine
F97C 18       |            CLC
F97D A5 EC    |            LDA  UserPC
F97F 69 01    |            ADC  #1
F981 85 EC    |            STA  UserPC
F983 90 02    |            BCC  swiSkip2
F985 E6 ED    |            INC  UserPC+1
F987 A5 ED    |swiSkip2    LDA  UserPC+1
F989 48       |            PHA
F98A A5 EC    |            LDA  UserPC
F98C 48       |            PHA
F98D AD 2A 02 |            LDA  CPUStatusReg push status reg
F990 48       |            PHA
F991 20 5C F2 |            JSR  RestoreSXYA
F994 40       |            RTI
              |*
              |*Directs interrupt to software SWI or Hardware IRQ routine
              |*
F995 20 40 F2 |SelectInterrupt JSR  SaveSXYAP
F998 68       |            PLA
```

```
1433 F999 8D 2A 02 |            STA  CPUStatusReg
1434 F99C 68       |            PLA
1435 F99D 85 EC    |            STA  UserPC
1436 F99F 68       |            PLA
1437 F9A0 85 ED    |            STA  UserPC+1
1438 F9A2 20 1B F9 |            JSR  IsItSWI
1439 F9A5 F0 03    |            BEQ  itsAnIRQ
1440 F9A7 4C 03 02 |            JMP  SWIService
1441 F9AA 4C 06 02 |itsAnIRQ    JMP  IRQService
1442               |*
1443 F9AD 20 AA F5 |ShowAllParameters JSR  WriteString
1444 F9B0 8D 53 74 |            RASZ 'StartAd= '
1444 F9B3 61 72 74 |
1444 F9B6 41 64 3D |
1444 F9B9 20 00    |
1445 F9BB 20 8B FA |            JSR  ResetBufferPointers     is forward ref'ed
1446 F9BE 20 5C F7 |            JSR  PrintSBCAdrs
1447 F9C1 20 AA F5 |            JSR  WriteString
1448 F9C4 8D 42 79 |            RASZ 'ByteCnt= '
1448 F9C7 74 65 43 |
1448 F9CA 6E 74 3D |
1448 F9CD 20 00    |
1449 F9CF AD 1C 02 |            LDA  ByteCount+1
1450 F9D2 20 1D F7 |            JSR  PrintByte
1451 F9D5 AD 1B 02 |            LDA  ByteCount
1452 F9D8 20 1D F7 |            JSR  PrintByte
1453 F9DB 60       |            RTS
1454               |*
1455 F9DC 20 AA F5 |GetFillWP   JSR  WriteString WP=With Prompt
1456 F9DF 8D 45 6E |            RASZ 'Enter Hex- '
1456 F9E2 74 65 72 |
1456 F9E5 20 48 65 |
1456 F9E8 78 2D 20 |
1456 F9EB 00       |
1457               |*
1458 F9EC 20 FE F6 |GetFillChr  JSR  GetHexByte
1459 F9EF 8D 30 02 |            STA  FillChar
1460 F9F2 60       |            RTS
1461               |*
1462               |* get number of bytes
1463 F9F3 20 FE F6 |GetNumberOfBytes JSR  GetHexByte
1464 F9F6 8D 1B 02 |            STA  ByteCount
1465 F9F9 60       |            RTS
1466               |*
1467 F9FA 20 FE F6 |GetNumberOfPages JSR  GetHexByte
1468 F9FD 8D 1C 02 |            STA  ByteCount+1
1469 FA00 60       |            RTS
1470               |*
1471 FA01 20 FE F6 |GetStartAdr JSR  GetHexByte
1472 FA04 8D 18 02 |            STA  StartOfBuffer1+1
1473 FA07 8D 38 02 |           STA  WarmStart
1474 FA0A 20 FE F6 |            JSR  GetHexByte
1475 FA0D 8D 17 02 |            STA  StartOfBuffer1
1476 FA10 60       |            RTS
1477               |*
1478 FA11 20 FE F6 |GetDestAdr  JSR  GetHexByte
1479 FA14 8D 1A 02 |            STA  StartOfBuffer2+1
1480 FA17 20 FE F6 |            JSR  GetHexByte
1481 FA1A 8D 19 02 |            STA  StartOfBuffer2
```

```
FA1D 60       |           RTS
              |*
              |* Given a byte count & a start ADDRESS, use
              |* current bytecount and bufferstart,to generate end Adr
              |*
FA1E 18       |GenEndAdr CLC
FA1F AD 1B 02 |          LDA  ByteCount
FA22 6D 17 02 |          ADC  StartOfBuffer1
FA25 8D 15 02 |          STA  BufferEnd
FA28 AD 1C 02 |          LDA  ByteCount+1
FA2B 6D 18 02 |          ADC  StartOfBuffer1+1
FA2E 8D 16 02 |          STA  BufferEnd+1
FA31 60       |          RTS
              |*
FA32 20 AA F5 |GetAddrWP  JSR   WriteString  With Prompt
FA35 8D 41 64 |           RASZ  'Addr: '
FA38 64 72 3A |
FA3B 20 00    |
FA3D 20 01 FA |GetAddress JSR   GetStartAdr  Without a prompt
FA40 20 1E FA |           JSR   GenEndAdr
FA43 60       |           RTS
              |*
FA44 20 AA F5 |GetDestAdWP JSR  WriteString  With Prompt
FA47 8D 44 65 |            RASZ 'Dest: '
FA4A 73 74 3A |
FA4D 20 00    |
FA4F 20 11 FA |GetDestAddr JSR  GetDestAdr   Without prompt
FA52 20 1E FA |            JSR  GenEndAdr
FA55 60       |            RTS
              |*
FA56 20 AA F5 |GetBytCntWP JSR  WriteString
FA59 8D 23 42 |            RASZ '#Bytes: '
FA5C 79 74 65 |
FA5F 73 3A 20 |
FA62 00       |
FA63 20 F3 F9 |GetByteCnt  JSR  GetNumberOfBytes
FA66 20 1E FA |            JSR  GenEndAdr

FA69 60       |            RTS
              |*
FA6A 20 AA F5 |GetPgCntWP JSR  WriteString
FA6D 8D 23 50 |           RASZ '#Pages: '
FA70 61 67 65 |
FA73 73 3A 20 |
FA76 00       |
FA77 20 FA F9 |GetPageCnt JSR  GetNumberOfPages
FA7A 20 1E FA |           JSR  GenEndAdr
FA7D 60       |           RTS
              |*
              |* Get all parameters for buffer operations
              |*
FA7E 20 32 FA |GetAllPar JSR  GetAddrWP
FA81 20 6A FA |          JSR  GetPgCntWP
FA84 20 56 FA |          JSR  GetBytCntWP
FA87 20 1E FA |          JSR  GenEndAdr
FA8A 60       |          RTS
              |*
              |* Restore buffer pointers
              |*
```

```
FA8B A0 00    |ResetBufferPointers LDY  #0
FA8D AD 1A 02 |             LDA  StartOfBuffer2+1
FA90 85 E3    |             STA  BufferPointer2+1
FA92 AD 19 02 |             LDA  StartOfBuffer2
FA95 85 E2    |             STA  BufferPointer2
FA97 AD 18 02 |             LDA  StartOfBuffer1+1
FA9A 85 E1    |             STA  BufferPointer1+1
FA9C AD 17 02 |             LDA  StartOfBuffer1
FA9F 85 E0    |             STA  BufferPointer1
FAA1 60       |             RTS
              |*
FAA2 AD 17 02 |SaveParams   LDA  StartOfBuffer1
FAA5 20 E1 F1 |             JSR  Push
FAA8 AD 18 02 |             LDA  StartOfBuffer1+1
FAAB 20 E1 F1 |             JSR  Push
FAAE AD 1B 02 |             LDA  ByteCount
FAB1 20 E1 F1 |             JSR  Push
FAB4 AD 1C 02 |             LDA  ByteCount+1
FAB7 20 E1 F1 |             JSR  Push
FABA AD 35 02 |             LDA  RomStart
FABD 20 E1 F1 |             JSR  Push
FAC0 AD 36 02 |             LDA  RomStart+1
FAC3 20 E1 F1 |             JSR  Push
FAC6 60       |             RTS
              |*
FAC7 20 F0 F1 |RestoreParams JSR  Pop
FACA 8D 36 02 |              STA  RomStart+1
FACD 20 F0 F1 |              JSR  Pop
FAD0 8D 35 02 |              STA  RomStart
FAD3 20 F0 F1 |              JSR  Pop
FAD6 8D 1C 02 |              STA  ByteCount+1
FAD9 20 F0 F1 |              JSR  Pop
FADC 8D 1B 02 |              STA  ByteCount
FADF 20 F0 F1 |              JSR  Pop
FAE2 8D 18 02 |              STA  StartOfBuffer1+1
FAE5 20 F0 F1 |              JSR  Pop
FAE8 8D 17 02 |              STA  StartOfBuffer1
FAEB 60       |              RTS
              |*
              |* Inc buffer pointers & Check for end-of-buffer.
              |* End = bufferEnd+1
              |*
FAEC E6 E0    |CkEndOfBuffer INC  BufferPointer1
FAEE D0 02    |              BNE  ckSkip
FAF0 E6 E1    |ckInc         INC  BufferPointer1+1
FAF2 E6 E2    |ckSkip        INC  BufferPointer2
FAF4 D0 02    |              BNE  Skip2
FAF6 E6 E3    |              INC  BufferPointer2+1
FAF8 A5 E0    |Skip2         LDA  BufferPointer1
FAFA CD 15 02 |              CMP  BufferEnd
FAFD A5 E1    |              LDA  BufferPointer1+1
FAFF ED 16 02 |              SBC  BufferEnd+1
FB02 B0 03    |              BCS  ckOut
FB04 A9 00    |              LDA  #FALSE  RETURN a '0' IF NOT end of buffer
FB06 60       |              RTS
FB07 A9 01    |ckOut         LDA  #TRUE   RETURN a '1' IF end of buffer
FB09 60       |              RTS
              |*
              |* Get from input & send to output
              |*
FB0A 20 2D F4 |XferBuffer JSR  GetInput
FB0D 20 AE F4 |           JSR  SendOutput
FB10 20 EC FA |           JSR  CkEndOfBuffer
FB13 F0 F5    |           BEQ  XferBuffer
FB15 60       |           RTS
              |*
              |* By doing it this way, you only have to set # of bytes,
              |* pages, & starting address once.
              |* after, use ResetBufferPointers to reset the pointers
              |*
              |* SEQUENCE:1. Get start and/or dest addresses
              |*          2. Get page and byte count (ByteCount(Hi/Lo))
              |*          3. JSR GenEndAdr
              |*          4. JSR ResetBufferPointers
              |*
              |* As long as you haven't changed buffer address or bytecount
              |* New memory operations only require a call to
              |* ResetBufferPointers before going into a loop
              |*
              |* Hi-level block mem move
              |* Example of System CALL:
              |* Get Dest addr & CALL GenEndAdr BEFORE calling MoveMemory
              |* See CmdInterpreter (toward end of listing) for examples
              |*
FB16 20 8B FA |MoveMemory JSR  ResetBufferPointers
FB19 20 0A FB |           JSR  XferBuffer
FB1C 60       |           RTS
              |*
              |*Send Hex Byte to output device; "bytecount" number of times
              |*
FB1D 20 8B FA |FillMem   JSR  ResetBufferPointers    Reset mem pointers
FB20 AD 30 02 |fillLup   LDA  FillChar
FB23 20 AE F4 |          JSR  SendOutput
FB26 20 EC FA |          JSR  CkEndOfBuffer
FB29 F0 F5    |          BEQ  fillLup
FB2B 60       |          RTS
              |*
FB2C 20 DC F9 |FillRoutine  JSR  GetFillWP
FB2F AD 34 02 |             LDA  OutputDest    Save output direction
FB32 48       |             PHA
FB33 A9 00    |             LDA  #MEMORY1
FB35 20 D4 F3 |             JSR  SetOutput     Change output to memory
FB38 20 1D FB |             JSR  FillMem
FB3B 68       |             PLA
FB3C 20 D4 F3 |             JSR  SetOutput     Restore former output
FB3F 60       |             RTS                direction
              |*
FB40 20 A2 FA |ClearSBCRam  JSR  SaveParams
FB43 20 7E FA |             JSR  GetAllPar
FB46 20 2C FB |             JSR  FillRoutine
FB49 20 C7 FA |             JSR  RestoreParams
FB4C 20 EF F5 |             JSR  DoneMessage
FB4F 60       |             RTS
              |*
              |* Download from Input device into ram
              |*
FB50 20 8B FA |DownLoad  JSR  ResetBufferPointers
FB53 20 79 F5 |          JSR  ReleaseHost
```

```
FB56 20 2D F4 |downLup   JSR  GetInput
FB59 A5 FC    |          LDA  HoldAByte
FB5B 91 E0    |          STA  (BufferPointer1),Y
FB5D A9 14    |          LDA  #XOn     14h
FB5F 20 45 F5 |          JSR  WriteChar
FB62 20 EC FA |          JSR  CkEndOfBuffer
FB65 F0 EF    |          BEQ  downLup
FB67 60       |          RTS
              |*
              |* UpLoad FROM RAM TO output device
              |*
FB68 20 8B FA |UpLoad    JSR  ResetBufferPointers
FB6B 20 F3 F4 |          JSR  Wait4Char
FB6E B1 E0    |upLup     LDA  (BufferPointer1),Y
FB70 20 45 F5 |          JSR  WriteChar
FB73 20 F3 F4 |          JSR  Wait4Char    Wait for an XOn
FB76 20 EC FA |          JSR  CkEndOfBuffer
FB79 F0 F3    |          BEQ  upLup
FB7B 60       |          RTS
              |*
              |* Load HEX data from Input device into RAM
              |*
FB7C 20 8B FA |LoadHexData JSR  ResetBufferPointers
FB7F 20 6A F5 |            JSR  WriteLn
FB82 20 5C F7 |            JSR  PrintSBCAdrs
FB85 20 8F F5 |            JSR  SendADash
FB88 20 FE F6 |getLup      JSR  GetHexByte
FB8B 20 D3 F4 |            JSR  SendMem1    store into mem @ BufferPointer1
FB8E 20 95 F5 |            JSR  SendASpace
FB91 20 EC FA |            JSR  CkEndOfBuffer
FB94 F0 F2    |            BEQ  getLup
FB96 60       |            RTS
              |*
FB97 29 0F    |CkPtrForHex0  AND  #0F
FB99 C9 00    |              CMP  #0
FB9B F0 03    |              BEQ  ckStartOn
FB9D A9 00    |              LDA  #FALSE
FB9F 60       |              RTS
FBA0 A9 01    |ckStartOn     LDA  #TRUE       TRUE IF lwr nibl = hex 00
FBA2 60       |              RTS
              |*
FBA3 48       |IncPointersSave PHA
FBA4 E6 EA    |                INC  TempBufferPtr
FBA6 E6 E0    |                INC  BufferPointer1
FBA8 D0 02    |                BNE  iSkip
FBAA E6 E1    |                INC  BufferPointer1+1
FBAC A5 E0    |iSkip           LDA  BufferPointer1
FBAE CD 15 02 |                CMP  BufferEnd
FBB1 A5 E1    |                LDA  BufferPointer1+1
FBB3 ED 16 02 |                SBC  BufferEnd+1
FBB6 B0 0A    |                BCS  iCkOut
FBB8 AD 1D 02 |                LDA  ExitFlag
FBBB D0 05    |                BNE  iCkOut
FBBD 9C 1D 02 |                STZ  ExitFlag RETURN FALSE IF NOT end of buffer
FBC0 68       |                PLA
FBC1 60       |                RTS
FBC2 A9 01    |iCkOut          LDA  #TRUE
FBC4 8D 1D 02 |                STA  ExitFlag
FBC7 68       |                PLA
```

```
FBC8 60       |                RTS
              |*
FBC9 91 EA    |WriteTempBfrChar STA  (TempBufferPtr),Y
FBCB 85 FC    |                 STA  HoldAByte
FBCD A5 E0    |                 LDA  BufferPointer1
FBCF 29 0F    |                 AND  #0F
FBD1 C9 07    |                 CMP  #07
FBD3 F0 03    |                 BEQ  stBfrOut
FBD5 A9 00    |                 LDA  #FALSE
FBD7 60       |                 RTS
FBD8 A9 01    |stBfrOut         LDA  #TRUE IF lwr nibl = hex 07
FBDA 60       |                 RTS
              |*
FBDB B1 EA    |GetTempBfrChar LDA  (TempBufferPtr),Y
FBDD 85 FC    |               STA  HoldAByte
FBDF A5 EA    |               LDA  TempBufferPtr
FBE1 29 0F    |               AND  #0F
FBE3 C9 08    |               CMP  #08
FBE5 F0 03    |               BEQ  gtBfrOut
FBE7 A9 00    |               LDA  #FALSE
FBE9 60       |               RTS
FBEA A9 01    |gtBfrOut       LDA  #TRUE IF lwr nibl = hex 08
FBEC 60       |               RTS
              |*
FBED A5 E0    |GenerateSpaces LDA  BufferPointer1
FBEF 20 97 FB |               JSR  CkPtrForHex0
FBF2 D0 19    |               BNE  genOut
FBF4 A5 E0    |               LDA  BufferPointer1
FBF6 29 0F    |               AND  #0F
FBF8 8D 25 02 |               STA  Count
FBFB 20 9B F5 |spaceLup       JSR  Send3Spaces
FBFE A9 20    |               LDA  #' '
FC00 20 C9 FB |               JSR  WriteTempBfrChar
FC03 E6 EA    |               INC  TempBufferPtr
FC05 CE 25 02 |               DEC  Count
FC08 AD 25 02 |               LDA  Count
FC0B D0 EE    |               BNE  spaceLup
FC0D 60       |genOut         RTS
              |*
              |* Print 1 line for memory dump
              |*
FC0E A5 E0    |PrintLineDump  LDA  BufferPointer1
FC10 29 0F    |               AND  #0F
FC12 85 EA    |               STA  TempBufferPtr
FC14 B1 E0    |pLineLup       LDA  (BufferPointer1),Y  Get memory byte
FC16 20 32 F7 |               JSR  PrintByteSave       Print to output dev
FC19 20 C9 FB |               JSR  WriteTempBfrChar    Store Ascii @ 0300
FC1C 20 A3 FB |               JSR  IncPointersSave
FC1F D0 05    |               BNE  sendBar
FC21 20 95 F5 |               JSR  SendASpace
FC24 80 11    |               BRA  plCont
FC26 B1 E0    |sendBar        LDA  (BufferPointer1),Y
FC28 20 87 F5 |               JSR  SendABarSave
FC2B 20 32 F7 |               JSR  PrintByteSave
FC2E 20 C9 FB |               JSR  WriteTempBfrChar
FC31 20 A3 FB |               JSR  IncPointersSave
FC34 20 95 F5 |               JSR  SendASpace
FC37 A5 E0    |plCont         LDA  BufferPointer1
FC39 20 97 FB |               JSR  CkPtrForHex0
```

```
FC3C F0 D6    |               BEQ  pLineLup
FC3E 60       |               RTS
              |*
              |* Print Ascii equivalent of PrintLineDump
              |*
FC3F 20 95 F5 |PrintAscii   JSR  SendASpace
FC42 A2 10    |             LDX  #10
FC44 20 F1 FF |             JSR  InitTempBuffer
FC47 20 DB FB |pAsciiLup    JSR  GetTempBfrChar
FC4A D0 18    |             BNE  sendAsciiBar
FC4C 20 DB FB |pLupBack     JSR  GetTempBfrChar
FC4F A5 FC    |             LDA  HoldAByte
FC51 20 89 F6 |             JSR  IsItAscii
FC54 F0 13    |             BEQ  pdotIt
FC56 20 DB FB |             JSR  GetTempBfrChar
FC59 A5 FC    |             LDA  HoldAByte
FC5B 20 45 F5 |pLupBack2    JSR  WriteChar
FC5E E6 EA    |             INC  TempBufferPtr
FC60 CA       |             DEX
FC61 D0 E4    |             BNE  pAsciiLup
FC63 60       |pAscOut      RTS
FC64 20 87 F5 |sendAsciiBar JSR  SendABarSave
FC67 80 E3    |             BRA  pLupBack
FC69 A9 2E    |pdotIt       LDA  #'.'
FC6B 80 EE    |             BRA  pLupBack2
              |*
              |* High level routine. Memory dump to output device (CIOD)
              |*
FC6D 20 F1 FF |PrintMem     JSR  InitTempBuffer
FC70 A9 00    |             LDA  #FALSE
FC72 8D 1D 02 |             STA  ExitFlag
FC75 20 8B FA |             JSR  ResetBufferPointers
FC78 20 6A F5 |printMLup    JSR  WriteLn
FC7B 20 5C F7 |             JSR  PrintSBCAdrs
FC7E 20 8F F5 |             JSR  SendADash
FC81 20 ED FB |             JSR  GenerateSpaces
FC84 20 0E FC |             JSR  PrintLineDump
FC87 20 3F FC |             JSR  PrintAscii
FC8A AD 1D 02 |             LDA  ExitFlag
FC8D F0 E9    |             BEQ  printMLup
FC8F 60       |             RTS
              |*
FC90 AD 18 02 |NewResetVector LDA  StartOfBuffer1+1
FC93 8D 38 02 |           STA  WarmStart
FC96 8D 0E 02 |           STA  Program1Ptr+1
FC99 AD 17 02 |           LDA  StartOfBuffer1
FC9C 8D 0D 02 |           STA  Program1Ptr
FC9F 20 AA F5 |           JSR  WriteString
FCA2 8D 4E 65 |           RASZ 'New Reset Vector Loaded'
FCA5 77 20 52 |
FCA8 65 73 65 |
FCAB 74 20 56 |
FCAE 65 63 74 |
FCB1 6F 72 20 |
FCB4 4C 6F 61 |
FCB7 64 65 64 |
FCBA 00       |
FCBB 60       |           RTS
              |*
```

```
FCBC AD18 02  |NewTestVector LDA  StartOfBuffer1+1
FCBF 8D 11 02 |           STA  Program2Ptr+1
FCC2 AD 17 02 |           LDA  StartOfBuffer1
FCC5 8D 10 02 |           STA  Program2Ptr
FCC8 20 AA F5 |           JSR  WriteString
FCCB 8D 4E 65 |           RASZ 'New Test Vector Loaded'
FCCE 77 20 54 |
FCD1 65 73 74 |
FCD4 20 56 65 |
FCD7 63 74 6F |
FCDA 72 20 4C |
FCDD 6F 61 64 |
FCE0 65 64 00 |
FCE3 60       |           RTS
              |*
              |*-----------------  Monitor Specific Routines  ------------
              |*
FCE4 20 F1 F3 |InitProcedures  JSR  SetIOToRS232
FCE7 20 A3 F5 |                JSR  InitTerminal
FCEA 20 8B FA |                JSR  ResetBufferPointers
FCED 20 1E FA |                JSR  GenEndAdr
FCF0 60       |                RTS
              |*
FCF1 A9 4C    |InitServiceRoutines LDA  #4C         Install JMP instruction
FCF3 8D 00 02 |                STA  NMIService  ahead of each pointer.
FCF6 8D 03 02 |                STA  SWIService
FCF9 8D 06 02 |                STA  IRQService
FCFC 8D 09 02 |                STA  IntService
FCFF 8D 0C 02 |                STA  Prog1Service
FD02 8D 0F 02 |                STA  Prog2Service
FD05 60       |                RTS
              |*
FD06 20 F1 FC |InstallRoutines JSR  InitServiceRoutines
FD09 20 9E FF |                JSR  InstalNMIRoutine    Forward Reference
FD0C 20 BF FF |                JSR  InstalSWIRoutine    FR
FD0F 20 CA FF |                JSR  InstalIRQRoutine    FR
FD12 20 D5 FF |                JSR  InstalSelectRoutine FR
FD15 20 A9 FF |                JSR  InstalProgram1     FR
FD18 20 B4 FF |                JSR  InstalProgram2     FR
FD1B 20 E0 FF |                JSR  InstalDBRoutine     FR
FD1E 60       |                RTS
              |* Executed upon power-up and with '+' command
FD1F A9 00    |InitMonitor  LDA  #FALSE  = Hex 00
FD21 A8       |             TAY
FD22 8D 2F 02 |             STA  ErrorMsg     :Error = 0
FD25 8D 1B 02 |             STA  ByteCount    :Bytes = 0
FD28 8D 17 02 |             STA  StartOfBuffer1  :address = xx00
FD2B A9 01    |             LDA  #TRUE
FD2D 8D 1C 02 |             STA  ByteCount+1  :Pages:= 01
FD30 8D 38 02 |             STA  WarmStart    :for subsequent resets
FD33 A9 05    |             LDA  #05          :initial buffer address:=0500
FD35 8D 18 02 |             STA  StartOfBuffer1+1
FD38 A9 14    |             LDA  #XOn         :Used by Wait4Char for
FD3A 8D 33 02 |             STA  Key          :flow control in WriteChar;
FD3D A9 34    |             LDA  #34          :So that only 1st reset on
FD3F 8D 20 02 |             STA  ResetFlag    :power-up init's monitor
FD42 A9 C0    |             LDA  #StackSpace  :init pointer
FD44 85 E4    |             STA  StackAddress :For Push; & Pop;
FD46 20 06 FD |             JSR  InstallRoutines :Reset,NMI,IRQ,Debug,Et
```

```
FD49 A9 FF    |                LDA  #0FF
FD4B 20 86 F4 |                JSR  SOutputByte  :SingleOutput gets := all 1's
FD4E 58       |                CLI               :Enable IRQ & SWI interrupts
FD4F 60       |                RTS
              |* Executed with each push of reset button
FD50 20 E4 FC |InitVarsEaReset JSR  InitProcedures :init buff addr & I/O
FD53 A9 00    |                LDA  #FALSE
FD55 85 E5    |                STA  StackAddress+1    :zero user stack
FD57 85 E6    |                STA  StackPointer      :for Push; and Pop;
FD59 85 EB    |                STA  TempBufferPtr+1  :init for linedump
FD5B 8D 21 02 |                STA  TraceFlag         :turn tracing off
FD5E 8D 37 02 |                STA  TraceVar          :user's trace variable
FD61 60       |                RTS
              |*
              |*    Initialize PROGRAM
              |*
FD62 AD 20 02 |Initialize      LDA  ResetFlag
FD65 C9 34    |                CMP  #34         Power up reset?
FD67 D0 08    |                BNE  firstReset  yes,so init monitor
FD69 AD 38 02 |                LDA  WarmStart   no,continue checks
FD6C CD 18 02 |                CMP  StartOfBuffer1+1 Has buffer adr changed?
FD6F F0 03    |                BEQ  byPassVars  no, so don't re-init monitor
FD71 20 1F FD |firstReset      JSR  InitMonitor yes, re-init monitor on 'warm'
FD74 20 50 FD |byPassVars      JSR  InitVarsEaReset  reset
FD77 60       |                RTS
              |*
FD78 78       |Begin     SEI
FD79 A9 FF    |          LDA  #0FF Init StackPointer
FD7B AA       |          TAX
FD7C 9A       |          TXS
FD7D 20 62 FD |          JSR  Initialize   Ck for changes, init accordingly
FD80 20 F6 F2 |          JSR  ReadDSRBit
FD83 D0 03    |          BNE  prog1        Init'ed w/cmdI
FD85 4C 0F 02 |          JMP  Prog2Service Init'ed w/TestProgram
FD88 4C 0C 02 |prog1     JMP  Prog1Service
FD8B 20 CC F5 |cmdI      JSR  PrintId
              |*
FD8E 20 6A F5 |CmdInterp JSR  WriteLn  JTerm's Command Interpreter
FD91 A9 3E    |          LDA  #'>'
FD93 20 45 F5 |          JSR  WriteChar
FD96 20 37 F5 |          JSR  ReadChar capable of ~6K loops/sec w/1 MHz clk
FD99 C9 31    |          CMP  #'1'
FD9B F0 37    |          BEQ  ci1
FD9D C9 32    |          CMP  #'2'
FD9F F0 3B    |          BEQ  ci2
FDA1 C9 33    |          CMP  #'3'
FDA3 F0 3F    |          BEQ  ci3
FDA5 C9 34    |          CMP  #'4'
FDA7 F0 43    |          BEQ  ci4
FDA9 C9 35    |          CMP  #'5'
FDAB F0 44    |          BEQ  ci5
FDAD C9 36    |          CMP  #'6'
FDAF F0 48    |          BEQ  ci6
FDB1 C9 37    |          CMP  #'7'
FDB3 F0 4C    |          BEQ  ci7
FDB5 C9 38    |          CMP  #'8'
FDB7 F0 55    |          BEQ  ci8
FDB9 C9 42    |          CMP  #'B'
FDBB F0 56    |          BEQ  ciB
```

```
FDBD C9 62    |         CMP  #'b'
FDBF F0 5A    |         BEQ  cib
FDC1 C9 43    |         CMP  #'C'
FDC3 F0 5E    |         BEQ  ciC
FDC5 C9 63    |         CMP  #'c'
FDC7 F0 5F    |         BEQ  cic
FDC9 C9 44    |         CMP  #'D'
FDCB F0 62    |         BEQ  ciD
FDCD C9 64    |         CMP  #'d'
FDCF F0 65    |         BEQ  cid
FDD1 4C 92 FE |         JMP  contCk
FDD4 20 32 FA |ci1      JSR  GetAddrWP   get address of buffer
FDD7 20 AD F9 |         JSR  ShowAllParameters
FDDA 80 B2    |         BRA  CmdInterp
FDDC 20 6A FA |ci2      JSR  GetPgCntWP get number of pages for buffer
FDDF 20 AD F9 |         JSR  ShowAllParameters
FDE2 80 AA    |         BRA  CmdInterp
FDE4 20 56 FA |ci3      JSR  GetBytCntWP get number of bytes for buffer
FDE7 20 AD F9 |         JSR  ShowAllParameters
FDEA 80 A2    |         BRA  CmdInterp
FDEC 20 AD F9 |ci4      JSR  ShowAllParameters show buffer parameters
FDEF 80 1A    |         BRA  cint
FDF1 20 7E FA |ci5      JSR  GetAllPar   prompt user for all buffer params
FDF4 20 AD F9 |         JSR  ShowAllParameters
FDF7 80 12    |         BRA  cint
FDF9 20 6D FC |ci6      JSR  PrintMem    dump out buffer to screen
FDFC 20 6A F5 |         JSR  WriteLn
FDFF 80 0A    |         BRA  cint
FE01 20 2C FB |ci7      JSR  FillRoutine fill buffer with hex character
FE04 AD 1C 02 |         LDA  ByteCount+1
FE07 C9 04    |         CMP  #4
FE09 90 EE    |         BCC  ci6         Print to screen IF < 4 pages
FE0B 4C 8E FD |cint     JMP  CmdInterp
FE0E 20 7C FB |ci8      JSR  LoadHexData load data into buffer by hand
FE11 80 E6    |         BRA  ci6
FE13 20 F0 F1 |ciB      JSR  Pop         set rts bit to user-pushed value
FE16 20 01 F3 |         JSR  SetRTSBit
FE19 80 F0    |         BRA  cint
FE1B 20 F0 F1 |cib      JSR  Pop         set dtr bit to user-pushed value
FE1E 20 15 F3 |         JSR  SetDTRBit
FE21 80 E8    |         BRA  cint
FE23 20 84 F8 |ciC      JSR  PrintRegs   print to screen CPU registers
FE26 80 E3    |         BRA  cint
FE28 A9 00    |cic      LDA  #FALSE      turn off print register SBR
FE2A 8D 21 02 |         STA  TraceFlag
FE2D 80 DC    |         BRA  cint
FE2F A9 01    |ciD      LDA  #TRUE       turn debugging on
FE31 8D 1E 02 |         STA  DebugFlag
FE34 80 D5    |         BRA  cint
FE36 A9 00    |cid      LDA  #FALSE      turn debugging off
FE38 8D 1E 02 |         STA  DebugFlag
FE3B 80 CE    |         BRA  cint
FE3D 20 8B FA |ciG      JSR  ResetBufferPointers   Run program out of ram
FE40 6C E0 00 |         JMP  (BufferPointer1)
FE43 20 4D F8 |ciH      JSR  HexToDecimal convert 4-digit hex to decimal
FE46 80 C3    |         BRA  cint
FE48 20 EF F6 |ciL      JSR  GetSingleHexNum read single bit input #__
FE4B 20 FD F3 |         JSR  GetSingleInput
FE4E 20 1D F7 |         JSR  PrintByte
```

```
FE51 80 3C    |          BRA  ci
FE53 20 44 FA |ciM       JSR  GetDestAdWP  copy buffer to indicated address
FE56 AD 32 02 |          LDA  IOState
FE59 48       |          PHA
FE5A A9 01    |          LDA  #InMem1OutMem2
FE5C 20 E0 F3 |          JSR  SetInOut
FE5F 20 16 FB |          JSR  MoveMemory
FE62 68       |          PLA
FE63 20 E0 F3 |          JSR  SetInOut
FE66 80 27    |          BRA  ci
FE68 20 87 F5 |ciO       JSR  SendABarSave turn on indicated output bit
FE6B 20 EF F6 |          JSR  GetSingleHexNum
FE6E 20 7F F4 |          JSR  SOutOn
FE71 80 1C    |          BRA  ci
FE73 20 EF F6 |cio       JSR  GetSingleHexNum turn off indicated O/P bit
FE76 20 7A F4 |          JSR  SOutOff
FE79 80 14    |          BRA  ci
FE7B 20 FE F6 |ciP       JSR  GetHexByte push 2-digit hex to user stack
FE7E 20 E1 F1 |          JSR  Push
FE81 20 95 F5 |          JSR  SendASpace
FE84 80 09    |          BRA  ci
FE86 20 F0 F1 |cip       JSR  Pop        pop 2-digit hex from user stack
FE89 20 1D F7 |          JSR  PrintByte
FE8C 20 95 F5 |          JSR  SendASpace
FE8F 4C 8E FD |ci        JMP  CmdInterp
FE92 C9 47    |contCk    CMP  #'G'
FE94 F0 A7    |          BEQ  ciG
FE96 C9 48    |          CMP  #'H'
FE98 F0 A9    |          BEQ  ciH
FE9A C9 4C    |          CMP  #'L'
FE9C F0 AA    |          BEQ  ciL     Note:Branches must NOT be further
FE9E C9 4D    |          CMP  #'M'         than +127 or -128 from program
FEA0 F0 B1    |          BEQ  ciM          counter. Exceed this range on
FEA2 C9 4F    |          CMP  #'O'         ANY branch (BRA,BEQ,BCC,etc)
FEA4 F0 C2    |          BEQ  ciO          & you may have real weird
FEA6 C9 6F    |          CMP  #'o'         program operation/symptoms.
FEA8 F0 C9    |          BEQ  cio          Current version of JAsm does
FEAA C9 50    |          CMP  #'P'         not catch this type of error,
FEAC F0 CD    |          BEQ  ciP          so use your LST file to check
FEAE C9 70    |          CMP  #'p'         the branch's 'Offset' value.
FEB0 F0 D4    |          BEQ  cip
FEB2 C9 53    |          CMP  #'S'
FEB4 F0 3D    |          BEQ  ciS
FEB6 C9 73    |          CMP  #'s'
FEB8 F0 41    |          BEQ  cis
FEBA C9 52    |          CMP  #'R'
FEBC F0 2B    |          BEQ  ciR
FEBE C9 72    |          CMP  #'r'
FEC0 F0 2C    |          BEQ  cir
FEC2 C9 54    |          CMP  #'T'
FEC4 F0 3A    |          BEQ  ciT
FEC6 C9 74    |          CMP  #'t'
FEC8 F0 3D    |          BEQ  cit
FECA C9 55    |          CMP  #'U'
FECC F0 40    |          BEQ  ciU
FECE C9 75    |          CMP  #'u'
FED0 F0 41    |          BEQ  ciu
FED2 C9 58    |          CMP  #'X'
FED4 F0 42    |          BEQ  ciX
```

```
FED6 C9 59    |           CMP  #'Y'
FED8 F0 4C    |           BEQ  ciY
FEDA C9 5A    |           CMP  #'Z'
FEDC F0 56    |           BEQ  ciZ
FEDE C9 3F    |           CMP  #'?'
FEE0 F0 58    |           BEQ  ciQM
FEE2 C9 2B    |           CMP  #'+'
FEE4 F0 5C    |           BEQ  ciPM
FEE6 4C 8E FD |cmdinterp JMP  CmdInterp
FEE9 20 90 FC |ciR       JSR  NewResetVector load current buffer addr as
FEEC 80 F8    |           BRA  cmdinterp      new reset vector
FEEE 20 BC FC |cir       JSR  NewTestVector  load current buffer addr as
FEF1 80 F3    |           BRA  cmdinterp      new test vector
FEF3 20 FE F6 |ciS       JSR  GetHexByte     Output 2-digit hex to out port
FEF6 20 86 F4 |           JSR  SOutputByte
FEF9 80 EB    |           BRA  cmdinterp
FEFB 20 AC F3 |cis       JSR  CheckSum       calculate MOD FFFF ck sum of
FEFE 80 E6    |           BRA  cmdinterp      current buffer
FF00 A9 01    |ciT       LDA  #TRUE         WriteChar uses XOn flow control
FF02 8D 23 02 |           STA  XOnFlag
FF05 80 DF    |           BRA  cmdinterp
FF07 A9 00    |cit       LDA  #FALSE        WriteChar does not use XOn flow
FF09 8D 23 02 |           STA  XOnFlag       control
FF0C 80 D8    |           BRA  cmdinterp
FF0E 20 68 FB |ciU       JSR  UpLoad        Upload current buffer to temp
FF11 80 D3    |           BRA  cmdinterp     file on host PC
FF13 20 50 FB |ciu       JSR  DownLoad      Download file from host to SBC
FF16 80 CE    |           BRA  cmdinterp
FF18 20 AA F5 |ciX       JSR  WriteString   load 2-digit hex into CPU X reg
FF1B 8D 58 3A |           RASZ 'X:='
FF1E 3D 00    |
FF20 20 FE F6 |           JSR  GetHexByte
FF23 AA       |           TAX
FF24 80 C0    |           BRA  cmdinterp
FF26 20 AA F5 |ciY       JSR  WriteString   load 2-digit hex into CPU Y reg
FF29 8D 59 3A |           RASZ 'Y:='
FF2C 3D 00    |
FF2E 20 FE F6 |           JSR  GetHexByte
FF31 A8       |           TAY
FF32 80 B2    |           BRA  cmdinterp
FF34 20 0C F4 |ciZ       JSR  SInputByte    read singlebit port as a byte
FF37 20 1D F7 |           JSR  PrintByte
FF3A A9 43    |ciQM      LDA  #43
FF3C 8D 20 02 |           STA  220
FF3F 4C 8B FD |           JMP  cmdI          monitor version to O/P device
FF42 20 1F FD |ciPM      JSR  InitMonitor  Full init of monitor
FF45 20 50 FD |           JSR  InitVarsEaReset
FF48 80 9C    |           BRA  cmdinterp
              |*
FF4A A9 00    |TestProgram    LDA  #FALSE
FF4C 8D 23 02 |               STA  XOnFlag  Disable XOn handshaking
FF4F A0 04    |               LDY  #4
FF51 A2 FF    |testLup        LDX  #0FF
FF53 20 29 F3 |ledOn          JSR  BusyReadRS232
FF56 CA       |               DEX
FF57 D0 FA    |               BNE  ledOn
FF59 88       |               DEY
FF5A D0 F5    |               BNE  testLup
FF5C A9 31    |               LDA  #'1'
```

```
FF5E 8D 30 04 |                STA  XmtReg ;
FF61 20 95 F5 |                JSR  SendASpace
FF64 AD 06 04 |                LDA  ClearSingleOut
FF67 A9 FF    |                LDA  #0FF
FF69 20 86 F4 |                JSR  SOutputByte
FF6C 20 E4 F2 |                JSR  Delay1Second
FF6F 20 F6 F2 |                JSR  ReadDSRBit
FF72 F0 DD    |                BEQ  testLup
FF74 A9 01    |                LDA  #TRUE
FF76 8D 23 02 |                STA  XOnFlag  Enable XOn handshaking
FF79 4C 8B FD |                JMP  cmdI     (flow control)
              |*
              |* End Test Routine
              |*
              |* Due to an assembler bug, all uses of "#>" that refer to a
              |* label, must be implemented at the end of all 'normal' code.
              |* Failure to do so will result in some weird problems because
              |* the addresses for your labels may be off by 3 bytes!
              |* #>Label designates the high byte of the label's address
              |* #Label designates the lower byte of the label's address
              |*
FF7C 20 6A F5 |IRQRoutine    JSR  WriteLn
FF7F 20 AA F5 |              JSR  WriteString
FF82 49 52 51 |              ASZ  'IRQ Interrupt'
FF85 20 49 6E |
FF88 74 65 72 |
FF8B 72 75 70 |
FF8E 74 00    |
FF90 20 5C F2 |intOut        JSR  RestoreSXYA
FF93 A9 FD    |              LDA  #>CmdInterp  push hi label address
FF95 48       |              PHA
FF96 A9 8E    |              LDA  #CmdInterp   push lo label address
FF98 48       |              PHA
FF99 AD 2A 02 |              LDA  CPUStatusReg push status reg
FF9C 48       |              PHA
FF9D 40       |              RTI
              |*
              |* Non Maskable Interrupt.      XaMonV4B FFFA/FFFB:= 00 02
FF9E A9 F9    |InstalNMIRoutine  LDA  #>NMIRoutine
FFA0 8D 02 02 |                  STA  NMIPointer+1  0200 points to NMI
FFA3 A9 2A    |                  LDA  #NMIRoutine  NMI service routine
FFA5 8D 01 02 |                  STA  NMIPointer
FFA8 60       |                  RTS
              |*
              |*Hardware Reset.               XaMonV4B FFFC/FFFD:= 00 F0
              |* If DSR bit is HI, vector to installed address on reset
FFA9 A9 FD    |InstalProgram1    LDA  #>cmdI
FFAB 8D 0E 02 |                  STA  Program1Ptr+1 Ram vector high
FFAE A9 8B    |                  LDA  #cmdI
FFB0 8D 0D 02 |                  STA  Program1Ptr  Ram vector low
FFB3 60       |                  RTS
              |* If DSR bit is LO, vector to installed address on reset
FFB4 A9 FF    |InstalProgram2    LDA  #>TestProgram
FFB6 8D 11 02 |                  STA  Program2Ptr+1 Ram vector high
FFB9 A9 4A    |                  LDA  #TestProgram
FFBB 8D 10 02 |                  STA  Program2Ptr  Ram vector low
FFBE 60       |                  RTS
              |*
              |* SoftWare Interrupt.           XaMonV4B FFFE/FFFF:= 09 02
```

```
FFBF A9 F9    |InstalSWIRoutine LDA  #>SWIRoutine Install pointer to SWI
FFC1 8D 05 02 |                 STA  SWIPointer+1 interrupt service routine
FFC4 A9 48    |                 LDA  #SWIRoutine  The SBR installed here
FFC6 8D 04 02 |                 STA  SWIPointer   will execute every time a
FFC9 60       |                 RTS               break is encountered in
              |*                                  your code
              |* Maskable Hardware Interrupt. XaMonV4B FFFE/FFFF:=0902
FFCA A9 FF    |InstalIRQRoutine LDA  #>IRQRoutine   Install pointer to IRQ
FFCC 8D 08 02 |                 STA  IRQPointer+1  service routine
FFCF A9 7C    |                 LDA  #IRQRoutine
FFD1 8D 07 02 |                 STA  IRQPointer
FFD4 60       |                 RTS
              |*
              |* IRQ/SWI Filter. Determines whether IRQ or SWI interrupt
FFD5 A9 F9    |InstalSelectRoutine LDA  #>SelectInterrupt
FFD7 8D 0B 02 |                    STA  IntServPointer+1
FFDA A9 95    |                    LDA  #SelectInterrupt
FFDC 8D 0A 02 |                    STA  IntServPointer
FFDF 60       |                    RTS
              |*
FFE0 A9 01    |InstalDBRoutine  LDA  #1         The SBR installed here will
FFE2 8D 1E 02 |                 STA  DebugFlag         execute if DebugFlag
FFE5 8D 21 02 |                 STA  TraceFlag         is TRUE,w/ every BRK
FFE8 A9 F8    |                 LDA  #>XaDebugRoutine
FFEA 85 EF    |                 STA  Debug+1
FFEC A9 ED    |                 LDA  #XaDebugRoutine
FFEE 85 EE    |                 STA  Debug
FFF0 60       |              RTS
              |*
FFF1 A9 03    |InitTempBuffer LDA  #>TempBuffer   Addr:=0300
FFF3 85 EB    |               STA  TempBufferPtr+1
FFF5 A9 00    |               LDA  #TempBuffer
FFF7 85 EA    |               STA  TempBufferPtr
FFF9 60       |               RTS
              |*
FFFA 00       |loNMI          BYTE   00   0200 = NMI memory vector
FFFB 02       |hiNMI          BYTE   02
FFFC 78       |loReset        BYTE   78   FD78 = Begin of Cmd Interpreter
FFFD FD       |hiReset        BYTE  0FD
FFFE 09       |loSWI          BYTE   09   0209 = SWI memory vector
define        |hiSWI          BYTE   02
              |*
0001          |          END  XaMonV4B
0001 %% %% %% |.
```

Chapter 14

JAsm Cross-Assembler

JAsm is a cross-assembler for the 65C02 processor that runs on your IBM PC or compatible.

This program takes your assembler source code, created using your favorite editor, and generates a binary file that the 65C02 CPU can execute, a .ROC file.

The binary values are represented as hex codes that correspond to the "*mnemonic*" source code instructions such as LDA (load the accumulator).

JAsm also calculates the branch instruction offsets and adjusts the PC (program counter) value relative to the ORG address you establish in your source code. **JAsm** includes some "fake" codes or "*psuedoOps*" that offer you some relief to your code writing task. More on that later.

FILES

JAsm can generate several different types of files for your use. These files have unique extensions and will now be explained:

.LST file is a file containing a list combining your source code with the hex information generated by the assembler, along with the program counter values and branch offsets.

Any time you have made an error in your source code, **JAsm** will automatically generate a .LST file that you can view with your editor. This list file will show you where and what types of errors are present. Errors are flagged in the list file by asterisks ***** and pointed to by a ^ character on the following source code line.

It is also possible to have **JAsm** generate a list file every time you assemble your source code file, by using the **Xasm.CFG** file to be discussed later.

.SYM file contains the symbol table. In this table you will find a summary of all the labels and identifiers used in your source code that begin with an upper-case alpha character (letter). The table will also indicate what type (what kind) of identifiers they are. The address where the label occurs is listed to the right of each identifier. You can use these addresses in your source code import list with other programs you write. See the CapMeter.ASM file included on the enclosed floppy disk for an example of how this works.

.ASC is a ascii-file representation of the .ROC binary file described below. Some EPROM burners have provisions for accepting this type of file. If you order a "bare" pc board from **JComm LAB**, you will receive a floppy disk with both the .ASC and .ROC files for the **XaMonV4B** monitor. If you have access to an EPROM burner, you will be able to burn your own **XaMonV4B** monitor EPROM.

.ROC file is the binary object code that you will download into your **SBC65V1B** and execute.

JAsm Operation

JAsm is easy to use. From the DOS prompt type **JAsm** and then <RETURN>. JAsm will show you a menu, listing all the (source code) files with the .ASM extention. Use the arrow keys on your keyboard to highlight the file you desire to assemble. Press the <RETURN> key. The assembly process proceeds as follows:

- The **JAsm** displays the filename of the file being assembled at the top of the screen.
- **JAsm** parses through your source code and indicates the number of lines of code parsed at the window on the upper right. Any errors indicated during this pass point to problems with your use of the MODULE (name) and END psuedoOps.
- On the next two passes, **JAsm** generates the necessary CPU codes. The ORG value(s) is displayed in the ORG window on the lower right of the screen during pass two. At the end of pass two, **JAsm** displays the size of your object code in Hex (base 16) and Decimal (base 10) in the upper right window.
- The error status of the assembly and the files generated are displayed next. If errors are present, **JAsm** will pause, so that you will be sure to notice that there are errors. Press any key to continue.

Xasm.CFG

If this file is not present in the directory containing **JAsm**, all the above mentioned files will be generated. Display the **Xasm.CFG** file contents by using the DOS "type" command

(type Xasm.CFG <RETURN>)

or by viewing with your favorite editor. Notice there are four BOOLEAN values (TRUE or FALSE). The default configuration for this file is:

```
FALSE
FALSE
FALSE
FALSE
```

The outcome from using these values is that only a .ROC file will be generated each time you run **JAsm**.

For example, if each BOOLEAN value in the **Xasm.CFG** file were declared TRUE, **JAsm** would generate the following files for a source file named
Test.ASM:

Test.LST
Test.SYM
Test.ASC
Test.ROC

You can create the Xasm.CFG file using DOS by typing:

```
copy con Xasm.CFG<RETURN>
 <RETURN>
 <RETURN>
 <RETURN>
 <RETURN>
```

```
FALSE<RETURN>
FALSE<RETURN>
FALSE<RETURN>
FALSE<RETURN>
(then press the F6 function key)<RETURN>
```

The file you have just created will instruct **JAsm** to only create a .ROC file.

If you want a .LST file to be created every assembly, replace the first "FALSE" encountered in the **Xasm.CFG** file with "TRUE." If you want a .SYM file, replace the second "FALSE" with "TRUE." If you want an .ASC file, replace the third "FALSE" with "TRUE."

During the normal program development cycle, you may find that your work progresses much faster if you leave all the BOOLEAN values as "FALSE." It takes time for **JAsm** to generate the extra files. As mentioned earlier, if **JAsm** encounters an error in your program, it will automatically generate the .LST file so that you can locate your error.

PsuedoOps

PsuedoOps are mnemonic instructions that are not part of the 65C02's instruction set, but are used by **JAsm**. Some of these special codes are intended to support you in your program writing.

An example of all 65C02 instructions and all **JAsm** psuedoOps can be found in the enclosed **XaTestR.ASM** file. Also see **CapMeter.ASM**.

Here is a description of the **JAsm psuedoOps**:

System psuedoOps

- **MODULE** is followed by the name of the file. It must be the same name used for the END psuedoOps.
- **ORG** is used to declare the Program Counter value. This is the actual memory map address where you want to start your code. (typically 0500Hex for 'ramware')
- **END** declares the end of your source code file. The filename declared here must be the same name used in the MODULE declaration.

Byte and String psuedoOps

- **BYTE** is used to enter 1 hex byte.
- **BYT** can also be used to enter 1 hex byte.
- **WORD** is used to enter two hex bytes. Note: An ascii string following a psuedoOp must not be longer than 31 characters.
- **ASC** used to enter ascii data as a string.
- **ASZ** is also used to enter ascii data. After the ascii data, JAsm automatically inserts a hex 0 into the object code.
- **RASC** is also used to enter ascii data. JAsm automatically inserts a hex 0D 'RETURN' before the string.
- **RASZ** is also used to enter ascii data. JAsm automatically inserts a hex 0D 'RETURN' before the string, and a hex 0 after the string.

psuedoOp Examples

```
MODULE  Test
ORG    0500
BYTE   0FF
BYT    0AA
WORD   0C000
ASC   'Hello World'
RASC  'Hello World'
ASZ   'Hello World'
RASZ  'Hello World'
END    Test
```

You may want to make a short Test.ASM program using the above examples, and then look at the LST file generated. You don't have to modify your **Xasm.CFG** file to have **JAsm** make a .LST file. Insert a deliberate error, such as

```
BRA nowhere
```

and **JAsm** will automatically generate the .LST file.

Explanation of Equates

It is possible to declare all variables, constants, pointers and addresses by using the "EQU" psuedoOp. Nonetheless, the following psuedoOps can be used in place of "EQU" to help make your source code more readable.

Equate psuedoOps

EQU is used to declare a constant.

ADR is used to declare an address. This address is typically a subroutine (or PROCEDURE) that has been written in another MODULE. It can also be the hardware address of some interface circuitry.

VAR is used to declare a common variable.

PTR is used to declare a POINTER variable.

These equate psuedoOps are used to mimic the "flavor" of **Modula-2**, the language chosen to write both **JAsm** and **JTerm**. I used the Jensen Partners "TopSpeed." If you are not familiar with **Modula-2**, I highly recommend learning it, even though you will probably have to refer to books that are out of print. The next choice would be to learn the older, but widely adopted commercially, **Pascal**.

You can teach yourself alot about structured *Top-Down* program design. If you adopt the underlying tenants of **Modula-2** or **Pascal** you will design and implement more efficient, readable, maintainable, modifiable and elegant assembly language programs. By learning **Modula-2**, you will have the background necessary to learn Niklaus Wirth's (the author of **Pascal** and **Modula-2**) latest and most elegant language...the object oriented ***Oberon***.

I have modified **Modula-2** programs that I had not even looked at for years, by merely "skimming" over the source code (like looking for a previously-read article in a magazine), finding the PROCEDURE of interest, and then making the necessary changes.

If you are careful with the selection of your of your identifier (label,constant and variable) names, you may find your assembly language programs easier to modify and maintain. Remember that all identifiers must begin with an upper-case letter if you want them to show up in the SYMbol table. Identifiers must be less that 31 characters long.

TROUBLESHOOTING YOUR PROGRAMS

The following is a list of some of the things that can go "wrong" during your assembly language experience:

➔ Trying to branch (BRA,BNE, etc) further than 127 bytes backward or 128 bytes forward. (unfortunately, **JAsm** will not flag this bug)

➔ Using and unequal number or pushes (PHA) or pops (PLA) in your subroutines (SBRs). If you enter a procedure using say, PHX (push xreg onto stack), you better not leave the procedure without using a PLX (restore x-reg from stack) instruction.

➔ Similarly, if you use the **XaMonV4B** subroutines "Push" or "Pop", or "SaveRegs" or "RestoreRegs", do not use one without the other. If you use Push twice in one subroutine, you better use Pop the same number of times somewhere else in your program (or in the same SBR).

➔ Failing to initialize flags, pointers or other variables in your program.

➔ Incorrectly using the Input/Output re-direction feature of the monitor. If you have I/O directed to memory, you have to re-direct the I/O yet again to get output to the RS232 port.

Here's a real gotcha:

➔ Failing to save the CPU registers before calling a SBR (procedure) that will ultimately clobber them.

```
Example: MyProcedure  (you left out PHY before loading Y)
                      LDY #5
         myLup        JSR  DoSomething
                      DEY
                      BNE  myLup
                     (you left out PLY before returning)
                      RTS
```

In this example, if "DoSomething" uses the Y register, it better save Y before using it and restore Y when it is done, or weird symptoms may show up in your program.

➔ Failing to use HoldAByte as a parameter passing variable for redirected I/O. See **XaMonV4B** listing (using your editor's search command) for uses of HoldAByte.

➔ Forgetting to end a string with zero when using the WriteString procedure.

```
Example;            JSR WriteString RASC  'Hello World'
```

WriteString will continue to treat all code following "Hello World" as if it were ascii data, resulting in somewhat bizarre effects.

```
Corrected example: JSR WriteString RASZ  'Hello World'
```

JAsm puts a hex 0 at the end of the string. The hex 0 tells WriteString that it has found the end of the ascii string. See **CapMeter.ASM** for more examples.

Another problem that can occur using WriteString has to do with punctuation. The psuedoOp is expecting a " ' " to begin and end the string.

```
Ex:  RASZ   'Hello World'
```

Consequently, it would be a mistake to try to use the ' delimiter in a string

```
Ex:  RASZ 'You're funny'
              ^bug
```

Not all possible problems are due only to software errors. Hardware can fail too! For example, if you fail to set the memory jumpers correctly, you may see a very weird memory dump when using **JTerm**. Once I set the EPROM jumpers for a 32K X 8 when I was using only a 16K X 8. Needless to say, the output from the SBC to my dumb terminal looked pretty scary. It didn't hurt anything but my pride!

Chapter 15

JTerm Communication Software

JTerm is a communication program. Using **JTerm**, your IBM PC or compatible becomes a terminal that you can use to communicate with the **µCEL SBC65V1B** single board computer/ controller.

This program has built-in provisions for sending commands and data to the SBC, and provides the menu for the **XaMonV4B** monitor program residing in the **SBC65V1B**. In fact, **JTerm** is more of a *responder* to the monitor program than it is a *controller*. Consequently, **JTerm** will be described in terms of explaining the monitor commands.

Connect the serial cable from your PC to your SBC. Type **JTerm** followed by a <RETURN>. Notice the menu presented to you. We will now explore the monitor.

Many of the **XaMonV4B** monitor functions operate on a *buffer* of memory at a time. The buffer can start most anywhere in the SBC's memory map, and can be from 1 to FFFF hex bytes long. The first eight commands of the **XaMonV4B** menu deal with this buffer.

Buffer Commands

1 Prompts you for the hex buffer address.
2 Prompts you for the number of hex pages (256locs/pg) in the buffer.
3 Prompts you for the number of hex bytes
4 Shows you the current buffer settings
5 Prompts you to input all the buffer settings
6 Dumps the buffer to the screen

The best way to learn these six commands is to practice using them. You can not hurt anything as you experiment with these first six commands. If you set a buffer longer than a few pages, and then use the '**6**' command, you may be in for as long wait as the SBC dumps the memory to the screen. If you get tired of waiting, press the reset button S1 on the SBC.

Commands '7' and '8' act on the buffer itself.
7 Fill memory buffer with the hex byte you enter
8 Fills memory buffer with the hex characters you type from the keyboard

Command '**8**' is the 'hand-load command', used when you want to enter a special sequence of bytes, or hand-load a program. Short programs can be entered very quickly this way. Try the following sequence of key presses: (do not type text enclosed in brackets (), it is for explanation only)

```
 5 (get all parameters)
1000 (hex address
  00 (number of pages)
  05 (number of bytes)
```

After the **5,** SBC will feedback a summary of what you entered

```
 8 (hand-load command)    0102030405   (hex characters)
```

Notice that the monitor displays the data as you enter it, and then does a memory dump, allowing you to verify the data or program you typed in. There is no backup key. If you make a mistake, you must re-type the data.

Memory Dump

- The memory dump screen is comprised of three basic parts:
- The PC address (Program Counter)
- The 16 byte hex line of data
- The ascii representation of the hex data.

Notice that there is a vertical bar |, dividing each 16 byte line of hex information into two 8 byte "chunks." Also notice the ascii representation of the hex data to the right of each line of hex data, is also divided by a bar. These bars are used as a visual reference point, to make it easier to find the byte of interest.

Memory Move

A copy of the currently defined buffer can be moved to a different location in memory with the '**M**' command. This copy may be referred to as the "destination buffer." After the '**M**' command, the monitor will prompt you for the destination's starting address.

If after doing the previous exercise you were to use the '**M**' command, and used 2000 for the destination address, you would find the hex characters 0102030405 starting at location 2000.

There is a "gotcha" to look out for. If you specify a destination address that is at the "tail" of your currently specified source buffer, you will clobber your data when you attempt the move. In the last example, if you had specified a dest addr of 1005, you would have clobbered the byte that contained the 05. Again, experimentation will probably be your best teacher.

Downloading

Probably the most useful feature of **JTerm** is the downloader. The downloader will transfer .**ROC** files you have generated with the **JAsm** cross assembler, to the buffer you have declared within your SBC's memory map. Actually, only the starting address of the buffer is needed. The downloader will automatically set the buffer size for you.

Downloading sequence

Cntrl-D (invokes the downloader) Arrow keys (highlight the file you want to download)
RETURN (selects the file you have highlighted)
Cntrl-Z (clears the transfer)

Running the downloaded program

G (for "Go")

Uploading

You may have designed an interface that has accumulated some data in ram that you want to store in a file. The upload feature of the monitor allows you to do this.

Cntrl-U (command for upload) The buffer you have declared will be sent to a **UpLoad.ROC** file. You can exit **JTerm**, and rename **UpLoad.ROC** to a file name of your choice. If you upload twice in a row, only the data sent from the second transfer will be present.

Checksum

s (performs a checksum of the current buffer) This checksum may or may not match the checksum given to you by the downloader.

I/O Commands

The **XaMonV4B** monitor allows you to control all the available I/O from **JTerm**. From **JTerm** you can use:

Command	(Action)
O followed by bit#	(Set any of the 8 output bits High)
o followed by bit#	(Set any of the 8 output bits Low)
L followed by bit#	(Read any of the 8 input bits)
S followed by hh	(Send a byte out to the 8 output bits. hh = 2 hex characters)
Z	(Read a byte from the 8 input bits)
B	(Set the level of the uart output bit RTS, by popping last stack entry)
b	(Set the level of the uart output bit DTR, by popping last stack entry)

You can use the output bits to control small relays directly from **JTerm**. If you do your own IBM PC programming, you can write programs that make simple ascii calls to the monitor, using the **O**,**o** and **L** commands rather than writing firmware for the SBC. The uart output bits can be exercised as follows:

P hh (push a hex byte onto user stack)
B (pop the formerly pushed by to the RTS bit)
OR
b (pop the formerly pushed byte to the DTR bit) It may be helpful to you to use your logic probe to monitor these bits as you experiment with the above instructions.

The following commands perform various system functions:
C (display CPU registers)
c (don't display CPU register until next '+' command)
D (software interrupt debug on) d (software interrupt debug off)
X (load x register with hex byte hh)
Y (load y register with hex byte hh)

If you carefully look at the **XaMonV4B** interrupt routines in the monitor source code, you can spot the user-installable debug routine. This routine can be turned on or off from **JTerm** with the '**D**' and '**d**' commands respectively. A typical debug routine prints key variables onto the screen, or allows you to set the value of variables.

Utility

H (converts 4 hex digits to Decimal)
P (pushes next 2 hex digits entered onto user stack)
p (pops last 2 hex digits entered off of user stack)
? (displays current version of monitor)

- (resets the SBC65V1B monitor's variables)
-

Reset Vectors

When you first power up your SBC, the system (**XaMonV4B**) monitor program is run. If you download a program into ram and run it, you will stay in your program until you press the reset button S1 on the SBC. Program control will then revert to the monitor.

What if you want ***YOUR*** program to run every time you press reset? That is the purpose of the '**R**' and '**r**' commands. Normally, the program counter points to the monitor at reset, and the test program in the monitor if J15 is shorted (remember the blink test?).

By pressing **R**, the current buffer address will become the new reset vector when jumper J15 is open.

By pressing **r**, the current buffer address will become the new reset vector when jumper

J15 is shorted.

This feature makes it possible for you to load a program starting at say, 0500, type the **R** command, and invoke that program every time you press reset S1 on the SBC. You can load another program at say, 2000, and type the **r** command. Now you have access to either program upon reset, depending on whether you have J15 shorted or open. I normally put my main program on the **R** vector and a test program on the (lower case) **r** vector.

Make sure your main program has a *hook* back to the monitor. A simple hook that would be in your program might be

```
    Exit JMP MonitorF1DE
```

Otherwise you may have to turn off power in order to get to the monitor. It can get even trickier if you have battery backed ram, so be sure to put in that hook to the monitor!!

This discussion is by no means exhaustive. The best way to learn is by doing. There should be just enough information in this chapter to make you good and dangerous!

JTerm has a couple of help keys. Here is a summary of the control keys you get with the **Control-H** command.

Cntrl-B (Set the baud rate)
Cntrl-D (Download from PC to SBC)
Cntrl-F (Tell JTerm your clock speed)
Cntrl-I (Set MSB of ascii data HI)
Cntrl-K (Other help key, examples)
Cntrl-N (Print the XaMonV4B menu)
Cntrl-Q (Quit the program)
Cntrl-T (Toggle the XOn handshake on/off)
Cntrl-U (Upload from SBC to PC file)
Cntrl-X (Send an XOn each keypress)
Cntrl-Z (Clear the screen)

The baud rate command gives you the opportunity to use **JTerm** for your RS232 experiments, such as implementing an RS232 uart in firmware. Some call that technique bit-banging.

The default baud rate for **JTerm** and for **XaMonV4B** monitor is 19,200 baud. To change the baudrate on the **SBC65V1B** you must first download the "**HotBaud.ROC**" program and run it; then type **Cntrl-B** and set **JTerm**'s baudrate to the baudrate you set with HotBaud.

The clock speed command is a "kludge" for the download feature. **JTerm** was designed to work on a 10 MHz system. Use the **Cntrl-F** command if your pc's clock speed is different than 10 Mhz.

This command sets a "magic number" within **JTerm**. It was designed to get around a PC system "burp" that occurs just often enough to upset the XOn protocol used by the downloader. A future version will remedy this problem.

As a last-resort, **JTerm** gives you access to the magic numbers themselves with "Set Magic Numbers," if the preset options do not work for you.

Chapter 16

Capacitance Meter

Objective

Design a capacitance measurement circuit using a 555 timer as the timing element.

Theory

Refer to circuit at **Figure 15**. An output bit is used to trigger the 555 input pin 2, initiating a timing cycle. The ouput of the timer pin 3 is normally low. When the 555 is triggered, the output pin 3 goes high. This output is monitored by an input bit on the SBC. A counter implemented in firmware measures the time, 't', that the output pin 3 of the 555 is high. The time 't' is a function of the unknown capacitance, Cx and resistor R1-R4. The range of this circuit is approximately 10 to 65,000 picofarads (65,000 pFd = .065 uFd).

Host Computer Software Jterm

Host computer sends an ASCII 'space' character via its RS232 port, to the **SBC65V1B** CPU I/O module.

The SBC firmware interprets this character as a "read capacitance" command. The SBC makes a capacitance reading which is in turn sent out its RS232 port as a string of ASCII characters.

If the host computer is running a *dumb terminal* program, the capacitance measurement is displayed directly on the crt. A custom-written host program could process the data from the **SBC65V1B** as required.

Ramware

CapMeter.ASM

A trigger pulse is created by toggling an output bit from **HI**, to **LO** and back to **HI**. Output of 555 is read by a single input pin bit, in a tight loop. 16 capacitance readings are taken by the SBC, then averaged. The averaged reading is converted to its ASCII decimal representation by the PROCEDURE "**WriteCard**." Zeroing for low values of capacitance is accomplished by the PROCEDURE "**Auto Zero**." All values output from the SBC are represented in picoFarads (pFd).

Checkout

Place a 1uFd for Cx. Monitor the output pin 3 of the 555 with a logic probe. Trigger the 555 by momentarily connecting the trigger pin 2 to ground. The resting state of the 555 output is **LO**. After triggering, the output should temporarily go **Hi** and then return to **LO**. If the output does not return to **LO**, your capacitor is too large a value, or you need to re-check your wiring. Also check power and ground. Vcc should read 5V.

Calibration Assemble the file **CapMeter.ASM**, using **JAsm**. Download the CapMeter.ROC file using **Jterm**.

After downloading the firmware, and hooking up the 2 SBC I/O bits to your interface, connect your reference capacitor to Cx. Use the highest precision (1% or better) value you can find. You can "make" a reference cap by choosing a high qulality polystyreene or mylar cap and measuring its capacitance value using an LCR meter. With no capacitor present, the capacitance reading should be '0' pFd. If it doen't, use the "**AutoZero**" function of CapMeter. Now with your "reference" cap, insure that the readings are within 1-5% of expected. If readings are not accurate enough for your purposes, adjust R1 until the cap reading comes up to your spec. Adjustments can be made by inserting a 100 + pot in serires with R1, or by using different value fixed resistors in various series or parallel connections. It is probably a good idea to use a "plug board" while determining the proper resistor values before you solder them into your "final" prototype.

Mechanical

This circuit can be soldered to an inexpensive single-sided "radio shack" type of

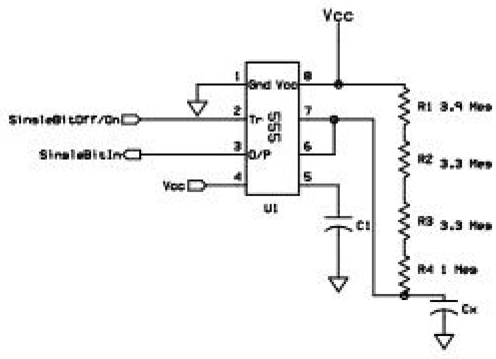


Figure 12 Simple Capacitance Meter

breadboard using point to point wiring. A common "plug board" works well too.

Jasm Assembly Listing

```
MODULE  CapMeter
(*JComm LAB 21 Apr 1992 12:30 PM -Information Block -
Description: This 'firmware' drives the simple cap meter interface from
Interface Note 1 FileName: CapMeter.ASM
(*Created FROM CAL555.ASM Filesize  :Source = 3519 Object Code = 269Z
System     :XaMonV4B-1 on SBC65V1B
Last Mod   : 20 Apr 1992
Assemblies:
Statistics : Last Hours : 1.5 Hrs/ 0000 Lines/00
Total Hours:12.5 Hrs/ 0130 Lines/40
Start Date : 27 Sep 1989
End Date: 6 Apr 1991
History:  6 Apr 1991 worked on SBC65V1A  10 Mar 1992 ported to SBC65V1B
End Info Block*)
```

```
FROM XaMonV4B IMPORT DelaySeconds,ReadChar,WriteChar,
                     SpaceEscWait,WriteLn,ClearScreen,
                     WriteString,GetNoResponse,GetHexByte,
                     PrintByte,WriteCard,CmdInterp,HoldAByte;
CONST
FALSE     = 00;
TRUE      = 01;
Escape    = 1B;
Threshold = 188Z;
VAR
Count        :ZPBYTE;
AverageCount :ZPWORD;
ReadFlag,ZeroAdjust,CapAdjust:ZPBYTE;
ADR
SingleBitOff  =  0410
SingleBitOn   =  0418
SingleBitIn   =  0420
ORG           =  0500;

                      JMP  Begin  skip over procedures
InitCount             LDA  #0
                      STA  Count
                      STA  Count+1
                      RTS

InitAveCount          LDA  #0
                      STA  AverageCount
                      STA  AverageCount+1
                      RTS DivideBy16
                      PHX
                      LDX  #4 divideLup
                      CLC
                      ROR  AverageCount+1
                      ROR  AverageCount
                      DEX
                      BNE  divideLup
                      PLX
                      RTS

AutoZero              WriteString('Remove Cx, then press SpaceBar');
                      SpaceEscWait;
                      STZ  ZeroAdjust
                      TakeOneReading;
                      LDA  Count
                      CMP  #0
                      BNE  notDone done
                      RTS notDone
                      INC  ZeroAdjust zeroLup
                      TakeOneReading;
                      LDA  Count
                      SEC
                      SBC  ZeroAdjust
                      BNE  notDone
                      WriteString('ZeroAdjust = ');
                      PrintByte(ZeroAdjust);
                      WriteLn;
                      BRA  done
GetCapAdjustment      WriteString('Current Adjustment = ');
                      PrintByte(CapAdjust);
                      WriteLn;
                      WriteString('Enter Hex Adjustment Factor>');
                      GetHexByte(CapAdjust);
                      RTS

AdjustReading         LDA  Count
                      SEC
                      SBC  ZeroAdjust
                      STA  Count
                      RTS
CheckForZero          LDA Count+1
                      BNE notZero
                      LDA Count
                      BNE notZero
                      LDA #TRUE
                      RTS notZero
                      LDA #FALSE
                      RTS

NormalizeCapacitance  CheckForZero;
                      BNE  normOut = 1 if count is zero
                      LDA  Count+1
                      BEQ  smallOut
                      LDA  Count
                      SEC
                      SBC  CapAdjust adjSmall
                      STA  Count
                      LDA  Count+1
                      SBC  #0
                      STA  Count+1 normOut
                      RTS smallOut
                      LDA  Count
                      SEC
                      SBC  #0
                      BRA  adjSmall
PrintReading          AdjustReading;
                      NormalizeCapacitance;
                      WriteString;                 ASZ  ' Capacitance= '
                      WriteCard(Count);
                      WriteString;                 ASZ  ' picoFarads'
                      WriteLn;
                      RTS

TakeOneReading        InitAveCount; (16 readings are
                      LDX  #16Z        averaged)
 countLoop            InitCount; trigger555
                      LDA  SingleBitOff
                      LDA  SingleBitOn read555
                      LDA  Count
                      CLC
                      ADC  #2
                      STA Count
                      LDA  Count+1
                      ADC  #0
                      STA  Count+1
                      LDA  SingleBitIn
                      BMI read555  *BPL (when testing w/no interface)
                      LDA  AverageCount
                      CLC
                      ADC  Count
                      STA  AverageCount
                      BCC  skipAveInc
                      INC  AverageCount+1 skipAveInc
                      WriteChar('.');
                      DEX
                      BNE  countLoop
                      DivideBy16;
                      LDA  AverageCount
                      STA  Count
                      RTS

PrintMenu             WriteString;
                      RASC 'JComm LAB Capacitance Meter',8D
                      RASC ' 1.>One Reading per Spacebar'
                      RASC ' 2.>Read continuously'
                      RASC ' 3.>Auto Zero'
                      RASC ' 4.>Adjust Cap Reading',8D
                      RASZ 'Esc to Exit'
                      WriteLn;
                      RTS
```

```
Begin                ClearScreen;
                     STZ  ReadFlag
                     LDA  #1
                     STA  Count+1
                     PrintMenu;
                     LDA  SingleBitOn cI
                     ReadChar;
                     CMP  #'1'
                     BEQ  ci1
                     CMP  #'2'
                     BEQ  ci2
                     CMP  #'3'
                     BEQ  ci3
                     CMP  #'4'
                     BEQ  ci4
                     CMP  #Escape
                     BEQ  ciEsc
                     BRA  cI ci1
                     JMP  TakeReadings
ci2                  LDA  #TRUE
                     STA  ReadFlag
                     JMP  TakeReadings
ci3                  AutoZero;
                     DelaySeconds(3);
                     BRA  Begin
ci4                  GetCapAdjustment;
                     DelaySeconds(2);
                     BRA  Begin
TakeReadings         TakeOneReading;
                     PrintReading;
                     LDA  ReadFlag
                     BNE  TakeReadings
                     SpaceEscWait;
                     LDA  HoldAByte
                     CMP  #Escape
                     BEQ  lookOut
                     BRA  TakeReadings
lookOut              CALL WriteString;
                     ASZ 'Go to Monitor? '
                     JSR  GetNoResponse;
                     BNE  Begin
ciEsc                JMP  CmdInterp

                     END  CapMeter
```