M1: Memory Mapped Interfacing
or
No pins left.

John N. Carter

Revised March 2015

Abstract

This experiment aims to give you practical experience of interfacing simple components to a computer bus. You will extend the memory available for data on an AVR micro controller and add extra input and output so that you don't loose functionality.

You will see how the extra memory can be used to store data and how the additional io ports can be created and used.

Note that this experiment requires the use of the Agilent Logic Analysers. If you are not familiar with these please ask for help early in the lab.

1 Introduction

The aim of this experiment is to introduce you to the concepts behind interfacing to a computer via an address and data bus. You will also have an opportunity to practice the skills of: reading and interpreting a circuit diagram and reading data sheets for complex integrated circuits. You will write a number of test programs to demonstrate how the interface circuitry you build works.

This experiment is in two main parts, spaced over two sessions. Specifically

1. In the first part you will investigate how a computer can be interfaced to external hardware via a simple bus.
2. You will interface an 8K SRAM to an Atmega162 . The bus takes up two i/o ports so you you will add additional input and output ports to the system.
3. You will test your memory interface and io ports.

This experiment concentrates on so-called parallel buses and does not treat the concept of serial busses which are increasingly common in todays computer systems.

2 Background

AVR's and PIC micro controllers are very easy to use because they have external connections that can be used for digital and analogue i/o. However as single devices they have severe limitations.

- They have a finite number of pins.
- They have a finite amount of RAM, 100's of bites for PIC's and up to 1-2k bytes for the smaller members of the AVR family.

- They are limited in the size of program they can hold. You will never run a word processor on a micro controller, though you can run a simple web server.

Micro controllers are this way because they are a compromise between being a computing device and a standalone electronic part. Under the hood, so to speak, they contain all the elements one would expect to find in an ideal computer i.e. an ALU, a controller, a data path and linking the different peripherals a bus with address, data and control. However because all this is hidden there is no indication that the device is actually a computer. This suggests that if we want to have more resources than the micro controller offers we need to use a microprocessor that has a full external bus.

Modern computers are characterized by an external interface for connecting external hardware to the CPU. This is called the computer bus. Depending on the actual architecture there may be more than one bus which may or may not share physical resources. Computers with so called Harvard architecture have two independent busses, one for data and the other for machine instructions, connecting to independent memories.

Interfacing via a computer bus takes as its paradigm the memory model. The address bus takes values which indicate which cell in memory is being accessed via the data bus. The former is unidirectional while the later is bidirectional. Each memory cell may be thought of as a potential source of electronic signals and as such only one memory cell can be active and drive the data bus at anyone time. Likewise the cpu is also a possible source of signals so the bus must continuously switch both the direction of information flow, plus the location from which it require the information to come from or go to. The cpu has a special place in this situation, it is the bus master[1]. It is normal for only one memory cell to be active at any given time. Thus a memory cell is active if it is addressed by the value on the address bus and can be either reading from the cpu , often known as writing in a cpu centric view, or writing to the cpu which to confuse matters is actually reading. The cpu centric model prevails and reads and writes are performed as the cpu directs. All other memory cells must ignore the transaction usually by setting their interface to high impedance.

Memory on the cpu bus can take the form of devices representing single cells, i.e. a simple latch or they may represent ranges of cells as in the case of a memory chip. In between are chips which present small ranges of memory to the bus, and these usually have specific and limited function. Examples range from disk controllers to graphics devices which make all their internal registers available to the cpu via the computer bus.

However with the micro controller all these things are hidden inside the package, the i/o pins are multifunction cells that can be programed to be inputs or outputs, analogue or digital. Digital outputs correspond to simple latches while digital inputs are simply simple buffers between the outside world and the computer bus.

In this experiment we are going to use a member of the AVR family which has an external bus to investigate general microprocessor interfacing.

Preparation: *There is significant preparation for this lab. You should read this document carefully and answer all the questions in italics (like this), writing your answers in your logbook. In writing the answer in you logbook you must also write down where you found it. This is as vital as the information itself.*

Preparation: *Revise your knowledge of C/C++ programming and Computer Architecture from 1st year.*

The following resources exist to support this experiment.

[1]In complex systems the cpu can relinquish this position, but usually only for short periods of time.

- This document.
- An M1 web site where you will find down loadable code, links to documentation (and where appropriate local copies). Notification of errors and corrections will be published here.
- There is an AVR C/C++ compiler and device programmer on all ECS workstations in the Electronics lab. The compiler is installed on all ECS workstations running Windows.
- A kit of parts including an Atmega162 and a MicroBus module mounted on two breadboards.

If code is written in advance it should be printed out, with line numbers before you come into the laboratory. Software development should be incremental and evolutionary as you are learning about new concepts. Follow the methodology of this experiment and write simple programs to explore how the interface works.

2.1 Background Reading

The following are good sources of information on the computer bus and other things related to this experiment.

Preparation: *You should look at all the following as part of your preparation.*

- C/C++ Programming.
- Notes and module text for 2nd year Digital Electronics.
- Module text.
- The following web pages may be useful.
  - `http://en.wikipedia.org/wiki/Memory-mapped_IO`
  - `http://en.wikipedia.org/wiki/Address_bus`
  - `http://en.wikipedia.org/wiki/Memory_address`
  - `http://en.wikipedia.org/wiki/Control_bus`

2.2 The Atmega162 micro controller

In this experiment we are using the Atmega162 micro controller. This is similar to the Atmega processor in other ECS boards and has:

- Bidirectional data ports.
- Counters and timers.
- A variety of external interrupts.
- It is programmable over its SPI bus.

However it is:

- A bigger package.
- Has lots i/o ports.

- Does not have a built in ADC or I^2C.
- The data memory space is expandable.

From your reading of AVR data sheets you will have noticed that the family are equipped with a number of different memory address spaces. The family are true Harvard architecture RISC processors and have separate program and user memory address spaces. These are physically separate as might be expected. The model is extended to a fuse and lock address space, though this only has a few bytes, and often an EEPROM space where data values can be stored without corruption by power loss. The EEPROM space varies from 10's to 100's of addresses or memory cells.

In most of the family the program space, organized as 16 bit words is larger that the data space. Typical values are 16k bytes of program memory and 1k bytes of SRAM. Normally neither of these are expandable and often limit the usefulness of the micro controller. The Atmega162 is an exception and its SRAM is expandable with an off-chip expansion bus.

The closed nature of the AVR limits its usefulness for interfacing works, but as noted before the Atmega162 is the exception. In this experiment we are going to use its expansion bus to add additional input and out put ports and extend the data memory size by 8k bytes, This demonstrates the general principles of memory mapped interfacing.

The Atmega162 achieves its expandability by sacrificing two 8 bit ports plus 3 other signals to form a 64k byte expansion bus. This extends the internal SRAM and provides a large range of possible addresses, see Figure 9a in the Atmega162 data sheet. Details of how to use the memory map of the expanded processor can be found in the data sheet on the web, see pages 25 and those following. Ignore all comments about 161 compatibility as this is a distraction we will use the chip in its native mode not emulating a legacy chip.

Preparation: *Which ports does Atmega162 use for its expansion address bus?*

Preparation: *Which port does it use for its expansion data bus?*

Preparation: *What are the 3 control signals used to control external memory?*

2.3 Addressing and the Computer Bus

Memory mapping is a very flexible scheme, that makes all input/output (i/o) appear as if at a fixed location in the memory map. This means that no special instructions are required to access the outside world. The AVR family is completely memory mapped except for 64 internal special registers which have additional instructions to allow individual bit access. These corresponds to the low 64 bytes of the i/o address space.

The computer bus has three components:

Address Bus This indicates which cell of memory is to be accessed by the cpu . It is write only by the cpu and read only by everything else.

Data Bus This is the path by which information flows from peripheral to cpu and vice versa. It is bidirectional, but only one active element may exist. Either the cpu when it is writing or a memory cell when it is writing (cpu is reading).

Control Bus This is the set of signals that controls the flow of information in to and out of the computer. It is generally write only by the cpu and read only by everything else.

There is *NO* way for the programmer to directly control any of the bits/signal on the expansion bus. Its activity is a side effect of the AVR executing a load or store instruction.

```
st Z,r20 ;Store contents of register 20 in
         ;location pointed to by the Z register.

ld r24,X ;Load register 24 from the location
         ;pointed to by the X register.
```

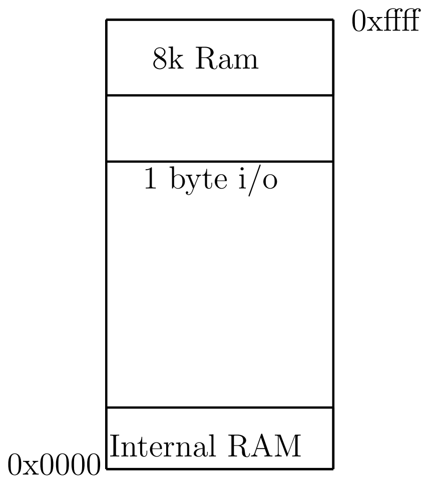
In the store operation the address and data to be specified are presented by the Atmega162 on the appropriate bus lines together with signals indicating that the processor is writing. The target peripheral must must respond and latch the value presented.

In the load operation the CPU data-path connects the desired address on the address bus and the peripheral must respond with the equivalent data which is latched into the CPU when the appropriate control signals are set. This is via the data bus.

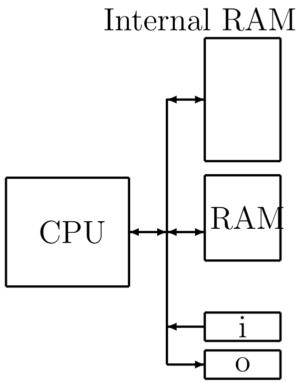
A computer system is often characterized by its address map. Figure 1(a) shows where different components might reside in the memory of a hypothetical computer. This simple Harvard architecture computer has 64k bytes of addressable read/write memory. The lower portion is reserved for internal RAM and special registers.

Preparation: *What is the highest address for the internal RAM in the Atmega162 ?*

In this exercise we want to interface 8kbytes of external RAM and 1 byte each of input and output.



(a) A 64k bytes address map.



(b) RAM, internal RAM, input (i) and output (o).

Figure 1: A simple computer system.

The simple system in Figure 1b might be implemented by one 8k RAM, an octal latch and an octal tristate buffer. Each device has a chip select signal, but the semantics of each are different.

LATCH has a single signal active when the latch is addressed and write enable is active, i.e. the union of these two signals.

BUFFER also has a single signal which should be active when the buffer is addressed and the cpu is reading from the device.

To generate these signals, a block of logic, often known as *glue logic*, is required to generate the chip select or combined signals. Specifically

RAM enable is active when the value on the address bus is in the top 8k of the 64k allowed range.

Latch enable is active when the cpu write line is active and the address bus has the value for the address of the latch.

Buffer enable is active when the cpu read line is active and the address bus has the the value for the address of the buffer.

This scheme is often known as full addressing. Take care in determining whether control signals and the components they control are active high or low.

Preparation: *What is the range of addresses occupied by the RAM in Figure 1? Give your answer in hexadecimal*

Preparation: *Design the logic required to implement the RAM enable signal from the address and control signals. Use only available discrete logic components.*

Full addressing can be wasteful of resources, and it is often possible to solve the problem using considerably less logic and possibly less cost.

For example the RAM, latch and buffer enable signal could be simply the top two bits of the address bus.

The RAM could be active when the four most significant bits of the address bus are 0b1000. The remainder of the address bus being connected to the RAM chip to specify the internal address.

The latch and buffer enable would be true when the top 4 bits are 0b0100. The read and write signals can be used to differentiate between latch and buffer. Alternately the chips might be given individual signals when the top of the address would be 0b0100 for the latch or 0b0010 for the buffer. It would be up to the software to select the appropriate values of the 4 most significant bit.

Preparation: *Suggest a simple circuit (1 or 2 '74 series logic chips) that could be used to implement address decoding when the top three bits of the address bus are used to select one of the RAM, latch or buffer. Hint: this method could select up to 5 other things.*

2.4 C and Memory addressing.

It is very easy to access an absolute memory location in C. It is also easy to do in C++. It is generally not possible in C#, Java or Python.

```
...
// Set things up
...
unsigned char *p;
unsigned char q;
...
p = (unsigned char *) 0xFFF0;
q=*p;
*p = 0xE1;
...
```

Program 1: Memory locations, accessed by pointers

The code in Program 1 shows how to set up a pointer to refer to an absolute memory location.

line 4 Defines `p` as a *pointer* to an 8 bit entity, an unsigned character. The pointer is actually a 16 bit word counting the address to be written.

line 5 Defines `q` as an 8 bit variable.

line 7 Makes `p` point to the memory location $0xFFF0$.

line 8 Reads location $OxFFF0$ and stores the contents in `q`.

line 9 Writes $0xE1$ to the location pointed to by `p`

During the execution of the program

at line 8 The micro-controller places $0xFFF0$ on the external data bus and latches what ever is on the data bus in response to the read signal.

at line 9 The micro-controller places $0xFFF0$ and $0xE1$ on the external address and data busses and generates a write signal for any external hardware.

3 Experimental Work

For this lab you are provided with an Atmega162 mounted on a MicroBus circuit board. It comes pre-mounted on two breadboards. This PCB has the Atmega162 programming interface, an 8k RAM, an interfacing latch, a tri-state buffer and a latch to de-multiplex the address/data bus. With the exception of the latch control signal no signals other than the data and address

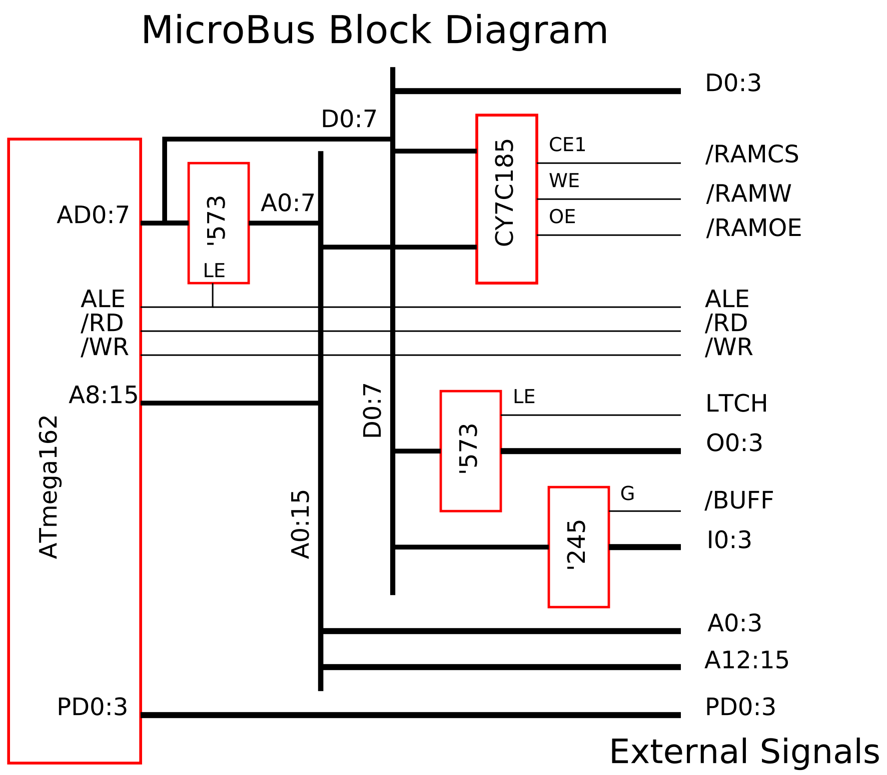


Figure 2: A block diagram of the MicroBus board, see the schematic diagram for details of pin numbers etc.

lines are connected on the PCB. All control signals and some buss signals are brought to the bottom edge of the board for observation and interface building.

Figure 2 is a block diagram of the MicroBus Board. You will find a picture and a circuit schematic on the web.

In this laboratory you will be making extensive use of the Agilent Logic Analyser. You should set this up when you arrive in the lab. If you are unsure how to use this as a supervisor.

3.1 First task (Session 1)

Preparation: *Read the sections in the Atmega162 data sheet concerning external memory. Then answer the following questions or do the following tasks.*

- Which bit in which register controls the availability of the external memory interface?
- Sketch the memory map of the Atmega162 with the external interface.
- How much of the memory space is available for external expansion? What is the highest address of the internal RAM?
- Which pins become the data bus and which pins become the address bus?
- How is it suggested in the data sheet that you can disambiguate between the dual functions.
- Sketch a hypothetical circuit to be used to implement the interfacing of a 64k bytes RAM.

Preparation: *Consult the AVR and latch data sheets. Record the polarity (active low or active high) of* ***/WR*** *and* ***Latch enable*** *on the '573. These are different. You will need an inverter to match these signals, see figure 3.*

Action: Connect ground and power to the MicroBus and any other breadboards.. Don't use long wires, use the shortest possible.

You are now in a position to program the Atmega162 .

- Refresh yourself on the use of AVR Studio and/or WinAVR.

Action: Plug the programmer cable into the 6 pin socket on the MicroBus.

BLACK MAGIC This experiment will not work correctly unless you disable the JTAG interface on the Atmega162 . This can be done by the following command.

```
avrdude -p m162 -c <programmer> -U hfuse:w:0xD9:m
```

This sets the default condition of the JTAG interface to off. You only need to do this once. However if for any reason you change the high fuse byte you must ensure that the JTAG bit is in the disabled state.

It is important when programming that you pay attention to the messages given by `avrdude` or any other device programmer. Unless the software says that the action has been carried out correctly then there is a problem either with the configuration or the command. To debug the programming connection: check that `/reset` goes low as `avrdude` runs; check you see a burst of 5 volt pulses on `sck` and on the two other lines. If you see all this, then the device is probably wired correctly. Since the PCB, connector and programming cable are pre-wired you should not have any problems.

```
...
// Set things up
...
unsigned char c = 0;
...
for(;;) {
    PORTA = c;
    c++;
    }
...
```

Program 2: Partial program to exercise PORT A.

Action: To show that the Atmega162 works, program it with a preprepared program that exercises the A port by writing an incrementing counter to it, see Program 2. Verify that Port A is showing the family of square waves as expected. The easiest way is to probe the Atmega162 directly on pin 39, 38, ...

Action: Modify the program to switch the micro controller into external memory mode. Compile, download and examine what port A is doing.

The next step is to exercise the external interface. Program 3 is a partial program that

```
1   ...
2   // Set things up
3   ...
4   unsigned char *p = 0xFF00;
5   ...
6   for(;;) {
7       *p = 0xFF;
8       tmp = *p;
9       }
10  ...
```

Program 3: Partial code to exercise memory location $0xFF00$.

could be used to exercise a single memory address. If the value of p was in the range 0x0000 to 0x04FF the program would read and write to the internal memory and special registers.

Preparation: *Why is this not good?*

Preparation: *What are lines 4, 7 and 8 doing.*

Action: Complete the code in Program 3 so that it compiles correctly and includes all necessary header files.

Action: Create, compile and download a completed, safe version [2] of Program 3

Action: Look on the Atmega162 and ALE and /WR pins. Can you see them changing.

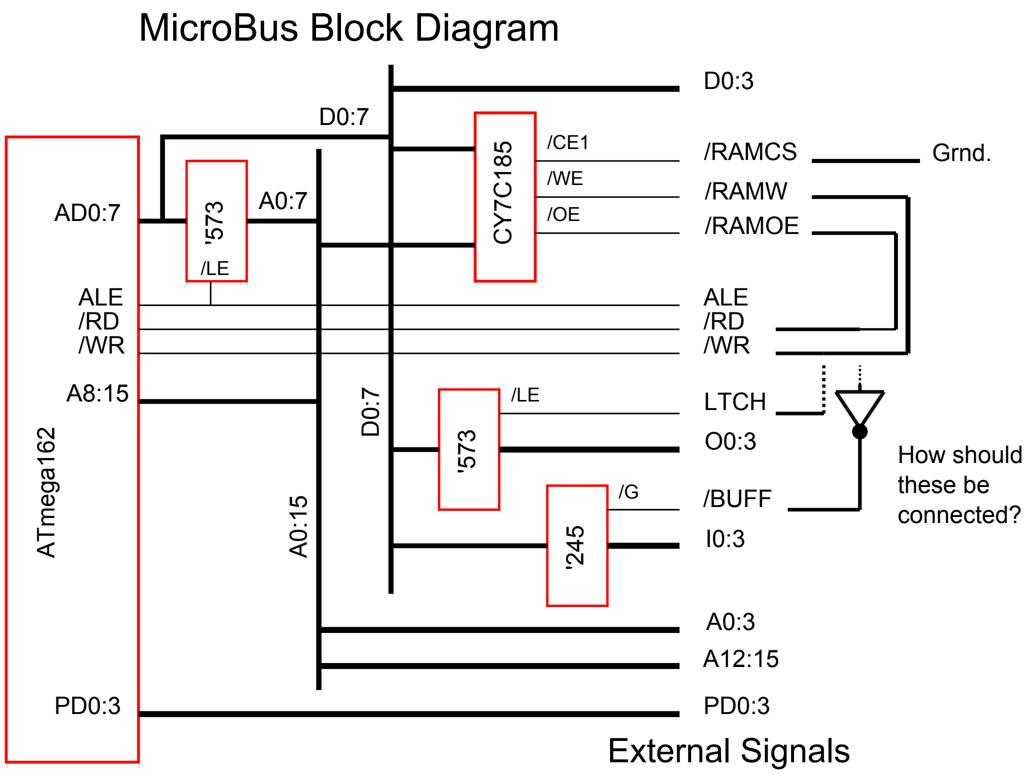


Figure 3: A block diagram of the MicroBus board, with connections marked

Signal	Logic Analyser Channel
ALE	0
/RD	1
/WR	2
PD0	3

Table 1: Logic Analyser and signal assignments. You should only need to add signals as you go along.

```
...
// Set things up
...
unsigned char *p;
...
while(1){ // must repeat the following infinitely
    ...
    for(p = 0x500; p != 0xFFFF; p++) {
        PORTD = 0;
        *p = 0xFF; // write 0xFF to the memory pointed to by p
        PORTD = 1;
        tmp = *p; // read from the memory pointed to by p
        }
    ...
}
...
```

Program 4: Partial code to exercise 64k memory space, less the internal memory.

Signal	Logic Analyser Channel
ALE	0
/RD	1
/WR	2
PD0	3
PD1	4
I0	5
I1	6
O0	7
O1	8

Table 2: Logic Analyser and signal assignments 2.

It is very difficult to see what is going on so modify your program to include a square wave generation on port D. Use the wiring conventions in Table 1

Now consider testing the whole memory space, using a completed version of Program 4. What are the different sections of the for loop doing (lines 8-13)?

You can use the signal on port PD0 to trigger the oscilloscope and stabilize the display. From the code you should notice that PD0 is low when writing to the external memory and is high when reading.

Action: Download the modified program

Action: Observe the relationship between port PD0, /WR, /RD and ALE. Record this.

Is anything missing? The program is reading and writing, so why is /RD constant?

Action: Change the definition of p to `volatile unsigned char *p;`.

What does `volatile` mean? What does it stop? Ask if you are confused.

3.2 Second Task (Session 1)

Action: Start wiring up the switches and LEDs. Note that we are *NOT* doing any explicit address decoding.

- Connect /WR from the Atmega162 to the LATCH signal on the MicroBus. Because of the mismatch in *polarity* of signals you will need to invert /WR. This inverted signal connects to the Latch Enable (LE) on the '573 whose inputs are connected to the outside world.
- Connect /RD to the buffer read signal (/G).
- The ALE signal is already connected to the latch enable (LE) of the address low latch. This will store the low byte of the address bus when it is multiplexed with the data byte.

Consult the PCB and/or the schematic for the names and positions.

You now have a functional latch and buffer. These will be written and read from any time the external signals read/write signals are generated by the Atmega162 .

[2]In this case safe means it does not overwrite the low memory.

Signal	Logic Analyser Channel
ALE	0
/RD	1
/WR	2
PD0	3
PD1	4
I0	5
O0	6
/RAMOE	7
/RAMW	8
/RAMCS	9
D0	10
D1	11
A0	12
A13	13
A14	14
A15	15

Table 3: Logic analyser and signal assignments 3.

1. Connect the logic analyser as shown in Table 2.
2. Connect Microbus inputs I0 and I1 to switches and AVR pins PD0 and PD1 to LEDs.
3. Modify your program so that it reads from 0xFFFF and writes to PORT D.

Action: Verify that this is happening.

To demonstrate that this is not some funny wiring effect consider inverting or shifting the output pattern.

4. Now connect Microbus outputs O1 and O2 to LEDs.
5. Modify your program so that it writes the value of the switches to a valid memory location.

Action: Modify your circuit for correct operation of the latch.

3.3 An external RAM or the third task (Session 2).

Disconnect the wires to the latch and buffer from /WD and /RD.

- Connect the /WD and /RD to the read (/RAMOE) and write (/RAMW) controls for the RAM.
- Wire the RAM select (/RAMCS) to ground.

Add the additional connections to the logic analyser as detailed in Table 3.

You should now have a functional ram in your external memory space.

It may be easier to see what is happening when you run a test program that writes a changing value to *p. An easy way to do this is to define an unsigned byte (`unsigned char x = 255;`) and change it from 0xFF or 0x00 on every write. This can be done by subtracting from 255 (i.e. `x = 255 - x;`). This is a general way of toggling between zero and something else.

Action: Chose values for the address and data for maximum change on the observed signals.

Action: What ranges of addresses should the RAM respond to?

Action: Modify Program 4 to exercise the high 8k bytes and check what is written and read back is correct. This checks everything works.

Action: Start your program running and capture a set of signals.

Action: What evidence is there that the RAM is being written to correctly? Take a screen dump.

Action: Do not disconnect the logic analyser.

3.4 The final phase: Glue Logic or Address Decoding.

Glue logic is the name given to the logic that ties a computer system together. In your laptop or desktop computer it normally resides in several large, hot and complex IC's on the motherboard. In this context it is the logic that generates the various chip select/enable signals from the address and control bus signals.

As you have discovered the RAM addresses are not unique when the ram select signal is permanently active. In this final subsection you are required to design some logic which will give a unique range of locations for the RAM and simple replicated locations for the latch and buffer.

You may use any components from the racks in the lab. While a CPLD might be the appropriate technology for this you may not use one for this exercise. Your aim in this design should be to use the minimum of logic/chip count/cost possible. Remember that the '74 logic family has devices other than simple logic circuits, things other than inverters, buffers, latches and gates..

Action: Design and build your logic. Discuss your solution with a lab supervisor or demonstrator so that you don't waste time.

Once you have designed the logic, add it to your circuit and using the LEDs and switches on the test bed demonstrate that you can read and write, using a suitable program. A single LED and single switch are all you need but do not connect them to the same bit on the data-bus. (Why ?) To aid debugging, connect the inputs and outputs of the glue logic to the logic analyser before you connect it to the Atmega162 and the ram, latch and tristate buffer. Unfortunately you do not have enough channels so reuse the pins that you don't need. Which ones do you not need to monitor at this stage? A minimal demonstration would be to

1. Read a value from the input (1 bit only)
2. Write the value to the Ram
3. Read it back from the Ram
4. Write it to the output.

Action: Write a demonstration program that uses all three components.

If you have time, consider coding up this simple sampling program. You are free to chose any method but a simple choice would be Program 5. This program samples the input from switches via the external buffer. It is like a mini logic analyser but without the screen. Stores the value in ram. Once the ram has been filled replay the contents using LEDs connected to the output latch. You will need to adjust the sampling and replay rate so that the recording time is 10 seconds, but it is displayed 4 times slower.

```
...
set up
repeat forever:
    while switch not pressed:
        do nothing

    for all external memory locations:
        read the external buffer
        write to sequential positions in the ram

    while switch not pressed:
        for all external memory locations:
            read from external memory
            Write to external latch
...
```

Program 5: Digital sample and replay test program.