

## I/O

---

### Announcements

- nottuBHack due
- PalmTasker out. Remember, this one is harder than the rest.
- Review session for PalmTasker either this Friday or next (voting in class).

### Last time

- Operating systems
  - How does an OS get started?
  - Examples of OSes
    - A little more
    - Extreme cases
  - Tasks
- Multitasking
  - How does multitasking work?
  - Specifics of tasking
  - A concrete example: Simplified PalmTasker

### This time

- I/O

## I/O

So far in class, we've focused on the CPU itself. We've talked about the various instruction and addressing modes, we've talked about data structures and (some) algorithms, we've even talked about operating systems. However, except for a brief aside when talking about interrupts, we haven't really talked about how we get information into or out of the CPU.

### *Instructions for I/O*

There are two basic schemes for talking to devices that are external to the CPU:

1. We can have special "I/O" instructions for doing things. Whenever we want to talk to something off-chip, we use these instructions to do it.
2. We can reserve some area of memory as a communication channel to an I/O device. Reading from this memory gets data from an I/O device, writing to this memory receives data from an I/O device. This is called *memory mapped I/O*.

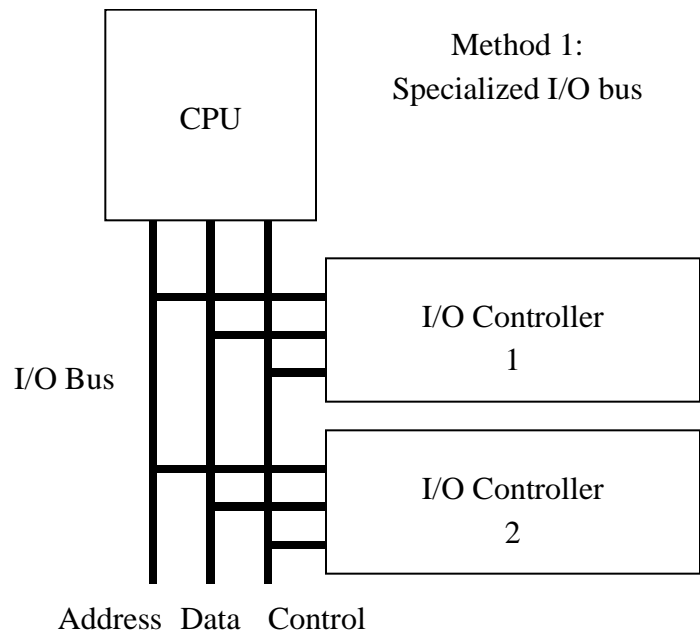
Computers have been created using both of these methods. There are advantages to both strategies. Option #1 is good because the CPU can easily identify I/O operations. If it wants to deal with I/O operations differently than memory operations, it can do it without a problem. Option #1 also has the advantage that it doesn't waste address space. Option #2 is good because CPU designers don't need to encode extra instructions into their CPU, making things all that much simpler (and faster). These days, almost everyone builds computers with option #2. The reason is that option #2's disadvantages aren't that important and it is a lot simpler (and less expensive) to implement.

Because memory mapped I/O is used by almost every computer, we will focus on it.

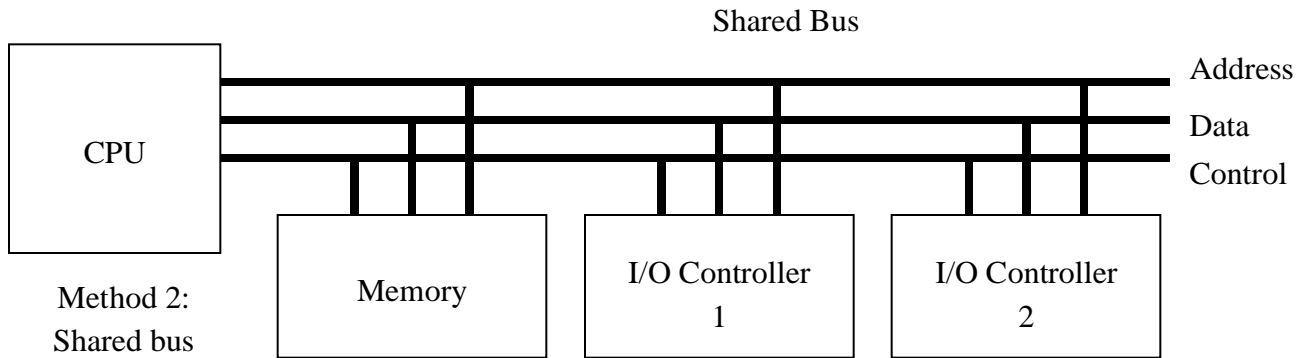
### *Architecture for I/O*

Let's draw some pictures to see how everything works. Somehow, the CPU is going to have to be connected to the various I/O devices. The devices themselves aren't connected directly to the CPU. Instead, I/O devices are connected to I/O controllers. These I/O controllers are then connected to the CPU. We have two possibilities for the connection:

One option is to have a specialized I/O bus. Remember earlier we drew a picture of the connection between the CPU and memory. This was the memory bus. The memory bus was used by the CPU to read and write values to and from memory. Here, we have an I/O bus. The I/O bus we have has three sets of wires: address, data, and control (note: they are sets of wires, not just one—that's why the lines are thick). The address wires are used to tell the I/O controller what data is needed. The data wires are used to transfer data. The control wires are used to keep multiple I/O controllers from talking at once (and also to handle interrupts). Note that these sets of wires vary a little from platform to platform, but they are basically the same.



When the CPU wants to talk to an I/O device, it uses the I/O bus to get the controller's attention and then talks to it.



We said that this looks an awful lot like the connection to the memory bus. Why not share the memory bus and just put I/O devices on it. That would look something like the picture above. Here, we have used the same set of wires leading out of the CPU to talk to both memory and to I/O controllers.

The two methods for organizing the hardware seem awfully similar to the two methods for organizing the instructions. Are they related? Not necessarily, but it is very likely. It would be difficult for a CPU that had memory-mapped I/O to have a specialized I/O bus. The CPU would have to internally route requests to the correct bus. Thus, CPUs that use memory-mapped I/O almost always use a shared bus (if not always). Do CPUs which have specialized I/O instructions always use a specialized I/O bus. Here, it wouldn't be difficult for the CPU to use either method. Since there really aren't any modern architectures that have specialized I/O instructions, I don't actually know whether any chips that have specialized I/O instructions have a shared bus. I doubt it.

What are the advantages of the different methods? The dedicated I/O bus is more expensive. It means that the CPU needs a separate set of pins coming out of it for each bus, which is costly. Which one is faster? The separate I/O bus is probably faster for conventional I/O (I/O devices don't need to tie up the memory bus plus the I/O behavior can be tailored to the characteristics of I/O). Having a shared bus might actually be faster in some cases though (as we will see later). The shared bus also has the advantage that we can use all the addressing modes of the 68K to access I/O controllers, since they are just at addresses. Again, I'm not an expert.

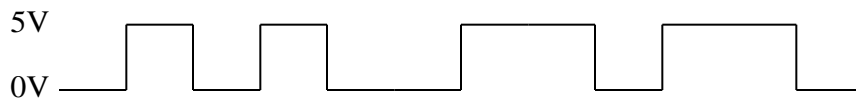
### ***Doing I/O***

How do we actually do I/O to a device? It actually varies by quite a bit by the type of I/O and also by the controller. There are all kinds of I/O devices that we might want to talk to. Some instances are disk drives, ethernet, keyboard, serial, and display. It helps to take a concrete example. We will focus on serial communication using the RS 232 protocol (the most common) using the somewhat defunct Motofolla 6850 ACIA chip (yes, this is not, on the Palm...sorry).

## Serial communication

Before we actually look at any assembly code, we first need to get some idea of what we're trying to do. With serial I/O, what we want to do is to be able to send and receive bits from one computer to another. Most people these days (except maybe those with iMacs) have a standard serial port on their computer that uses the RS-232 protocol. Using a cable, you can connect the serial port of one computer to the serial port of another computer (or a device that isn't a computer). What does that give us? A serial connection by definition allows you to send one bit over it at a time. To send larger amounts of data, we send sequences of bits. A simplified serial connection consists of 1 wire going in each direction. Each computer uses one wire to read from and the other wire to write to. To write a bit, the sender applies +5 volts to its wire. The other end detects this voltage and receives the bit.

Let's look at an example. We will send the letter 'M', whose ASCII value is 01001101. Let's look at one possible set of signals that could go over the wire to send the M.



What have we done?

- We sent 1 bit to signal that we were starting to send data (this is called the *start bit*).
- We sent 8 bits of data (01001101).
- We sent 1 bit to signal that the data was done (the *stop bit*).

Sound simple? It's actually not quite as easy as it first looks. Different devices use slightly different ways to talk over the serial port. Here's the things we need to look at:

1. What speed that bits are sent at? The sender and receiver must agree on this precisely. If they don't, there is no way that one person can read the other's bits. This parameter is known as the bit rate (also known as bits per second/bps or baud rate). There are several standard rates:  
75, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600
2. How many bits are in a byte? It turns out that for backward compatibility, you can have anywhere from 6 to 8 bits.
3. Details about start/stop bits (are they present, how many of them, etc).
4. Is there parity? Parity is a simple form of error detection. Basically, two parties that are using a parity check agree beforehand whether to use even parity or odd parity. When using even parity, there should be an even number of 1's in each byte. When using odd, there should be an odd number of 1's. To make this possible, one bit of the transmission is reserved and is called the "parity bit". This bit is set so as to make the above condition true.

If we wanted to add a parity bit to our above transmission, we would have instead send:

| start | data    | parity | stop |
|-------|---------|--------|------|
| 1     | 1001101 | 0      | 1    |

Note that I have now only send 7 bits of data (to keep the total at 8). You can see that that there are an even number of 1's in our transmission. If instead we sent 'L', we would send:

| start | data    | parity | stop |
|-------|---------|--------|------|
| 1     | 1001100 | 1      | 1    |

This time, the parity bit was a 1 to make sure that the total number of 1's was still even. Parity is a very simple means of catching errors. It won't catch all errors.

### Back to CS 110: the 6850 ACIA

Ok, that was enough of a detour into learning how serial communications work. Now, let's learn how to actually send our serial data. I said that I/O devices are connected to the CPU through an I/O controller. The I/O controller that we will be using is the Motorola 6850 ACIA. We control the ACIA using memory mapped I/O. That means that we can send messages to the ACIA by writing to certain memory locations and we can read information from the ACIA by reading from certain memory locations. The specific memory locations used depend on the hardware configuration.

The ACIA has four "registers" that are mapped into memory. There is a picture of these registers on page 569 of the optional Wakerly textbook. I have also drawn the picture and included it on the next page. We will say that the Control/Status registers are mapped to 0xA0000000 and the Send/Receive registers are mapped to 0xA0000004. That means that we can put something in the send register by doing:

```
MOVE.B    D0, 0xA0000004
```

We're actually using absolute addressing! Notice that I have mapped two registers to each location. This is because the Send register is read only and the Receive register is write only. The ACIA takes advantage of this by mapping them to the same location. The status and control register are similar.

What can we do with the ACIA registers? The control registers is used to control the operation of the chip. The included diagram shows what the different values of the control register mean. Bit 7 is used to enable/disable receive interrupts. Bits 5-6 control send interrupts and also information about flow control. Bits 2-4 control the format. Bits 0-1 control the speed.

**CONTROL (Write)**

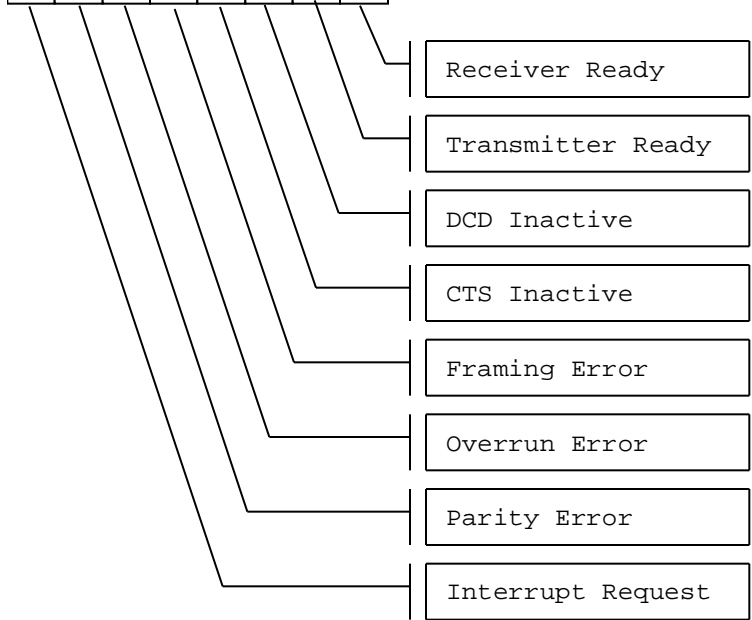
|    |              |
|----|--------------|
| 00 | 1            |
| 01 | 16           |
| 10 | 64           |
| 11 | Master Reset |

|     |                                       |
|-----|---------------------------------------|
| 000 | 7 data bits, 2 stop bits, even parity |
| 001 | 7 data bits, 2 stop bits, odd parity  |
| 010 | 7 data bits, 1 stop bit, even parity  |
| 011 | 7 data bits, 1 stop bit, odd parity   |
| 100 | 8 data bits, 2 stop bits, no parity   |
| 101 | 8 data bits, 1 stop bit, no parity    |
| 110 | 8 data bits, 1 stop bit, even parity  |
| 111 | 8 data bits, 1 stop bit, odd parity   |

|    |  |
|----|--|
| 00 | RTS active, XMT interrupt disabled               |
| 01 | RTS active, XMT interrupt enabled                |
| 10 | RTS inactive, XMT interrupt disabled             |
| 11 | RTS active, XMT interrupt disabled sending BREAK |

|   |                        |
|---|------------------------|
| 0 | RCV interrupt disabled |
| 1 | RCV interrupt enabled  |

**STATUS (Read)**



MC6850 ACIA  
(based on page 569  
of Wakerly)

**XMT DATA (Write)**

**RCV DATA (Read)**

The status register can be used to see the status of the transmission. It contains a lot of error detection and also lets us see whether our transmission finished.

The XMT and RCV registers are used to send and receive bytes. If we put a byte in the XMT register, it will be transmitted. When we receive a byte, it goes into the RCV register.

### Busy-wait I/O

We know how to put stuff into the ACIA registers and a little bit about how it works. Let's see an example of using the ACIA to get a better feel for how to use it. This example (and the next few) are taken straight out of the course notes of Tim Chou, the previous instructor of this course (consequently, they are using Cöder assembly):

```

CRTCS    EQU    $----    Console ACIA control and status
CRTDTA   EQU    $----    Console ACIA xmit and rcv data
CRESET   EQU    $03     ACIA mode control reset
CMODE    EQU    $11     ACIA mode control, 8 data bits
*        *
*        *              no parity, 2stop bits, RTS active
*        *              divide-by-16 clock, no interrupts

RCVRDY   EQU    0       Receiver ready bit # in ACIA port
XMTRDY   EQU    1       Transmitter ready bit #

CINIT    MOVE.B    #CRESET,CRTCS    Initialize console ACIA
          MOVE.B    #CMODE,CRTCS    by sending master reset
          RTS                          and setting proper mode

CRTIN    BTST.B    #RCVRDY,CRTCS    Check for character
          BEQ      CRTIN             No, busy wait
          MOVE.B   CRTDTA,D0         Yes, read the character

CRTWT    BTST.B    #XMTRDY,CRTCS    Can we echo it?
          BEQ      CRTWT             No, busy wait
          MOVE.B   D0,CRTDTA        Yes, output it
          RTS

```

In this example, we have functions to initialize the ACIA, send one byte, and receive one byte. If there is no byte to receive, we just wait until there is one. If the ACIA is already busy sending a byte, we wait until we can send.

Let's see a more complex example. Subroutine STROUT is used to print C style text strings on the display. A C style text string is defined to be a sequence of non-NULL ASCII characters terminated by NULL(0). To print a string, a program calls STROUT with X containing the address of the first character of the string. STROUT uses busy-wait I/O to output the string and returns to the calling program.

```

1.  Crtcs    EQU      $30041    Display cntrl & status port
2.  Crtdda  EQU      $30043    Display data port
3.  Xmtrdy  EQU      1         Transmitter ready bit

;          OUTPUT routine - strout

      Strout  PROC
4.          MOVE.B    (A0)+,D0    Get first character
5.          BEQ      Strudun      String is finished
6.  Chrout   BTST.B    #Xmtrdy,Crtcs char been displayed
7.          BEQ      Chrout       Wait until done
8.          MOVE.B    D0,Crtdda
9.          BRA      Strout
10. Strudun  RTS
      ENDPROC

```

The above routine spends a lot of time sitting and waiting for I/O operations to complete. That's because I/O operations are really slow. Here are some times:

19.2 Kb (19,200 baud) = 50  $\mu$ sec per bit = 500  $\mu$ sec to transmit an 8 bit word

20Mhz MC68000 = 0.05  $\mu$ sec (\* 4 cycles) = 0.2  $\mu$ sec to R/W an 8/16 bit memory location

There about a 1000x difference in the amount of time to do an I/O operation compared to the time it takes to fetch stuff from memory.

### Interrupt I/O

Ideally, we would like to do something better while waiting. With our simple model, we can't. However, we've already talked about how devices can be given the ability to interrupt the CPU and on the ACIA we can see control and status bits for using interrupts. Let's see an example.

```

; To print a string, call STROUT with X containing
; the address of the first character of the string. STROUT
; enables the display interrupt, prints the first character
; and returns to the calling program. Subsequent character
; strings are printed by the interrupt svc routine STRINT.

```

```

1.  crtcs    EQU      _____ Console ACIA control &status
2.  crtdda  EQU      _____ Console ACIA xmit & rcv data
3.  crtvect  EQU      $0064 ACIA lvl 1 autovector loc
4.  creset   EQU      $03   ACIA mode control reset
5.  cmode    EQU      $11   ACIA mode control,8 databits
6.  *
7.  *
8.  cinton   EQU      $20   ACIA XMT interrupt enable bit
9.  contoff  EQU      $00   ACIA interrupts disabled.

```

```

11. *           Global and local variables

12. crtbsy     DS.B           1           Nonzero means a string is
                                           currently being displayed
13. bnpt       DS.L           1           Ptr to next character to display

19. *
20. * CRT Output Init, pointer to string passed in A0
21. *
22. CRTOUT     TST.B          CRTBSY(A5)    Still busy prevs str?
23.           BNE            CRTOUT        Yes, wait for finish
24.           MOVE.B         #1,CRTBSY(A5)  Mark CRT busy
26.           MOVE.B         (A0)+,CRTDTA  Send 1st char to ACIA
27.           MOVE.L         A0,BPNT(A5)   Save ptr to next char
28.           MOVE.B         #CMODE+CINTON,CRTCS Enable interrupts
29.           RTS                                Done, return
30. *
31. * CRT output interrupt handler
32. *
33. CRTINT     MOVEM.L        D0/A0,-(SP)    Save registers
34.           MOVE.L         BPNT(A5),A0    Get ptr to nxt char.
35.           MOVE.B         (A0)+,D0      Fetch the character
36.           BEQ            CDONE         If 0 we're done
37.           MOVE.L         A0,BPNT(A5)   Else save updated ptr
38.           MOVE.B         D0,CRTDTA     print the character
39. CRTRET     MOVEM.L        (SP)+,D0/A0   restore registers
40.           RTE                                and return

41. CDONE     MOVE.B         #CMODE+CINTOFF,CRTCS Disable inter.
42.           CLR.B          CRTBSY(A5)    CRT not busy
43.           BRA            CRTRET        Restore & return

44. *
45. * MAIN PROGRAM
46. *
   ASYNCH1   MAIN
47.           CLR.B          CRTBSY(A5)    Mark CRT not busy
48.           LEA            CRTINT(PC), A0
49.           MOVE.L         A0, crtvect   Init inter tbl
50.           MOVE.B         #CRESET,CRTCS  Reset ACIA,
51.           MOVE.W         #$2000,SR     Super, CPU pri=0
52.           LEA            MSG1,A0      Get addr of msg
53.           JSR            CRTOUT        and print it.
57.           ENDMMAIN

```

This example shows how to use interrupts to do I/O. It uses a few things that we haven't really talked much about (we skimmed over them when talking about interrupts earlier). One of these

things is which vector location is used by a given I/O device. The other is the IPL. Let's look at the IPL first.

### The IPL

When I/O controllers perform an interrupt, they associate a priority with that interrupt. On the 68K, this interrupt can be anywhere from 1 to 7. The idea is that certain I/O devices are more important than others. This allows one I/O controller to interrupt another I/O controller if it has a higher interrupt level.

How does this work? Partially, we have to believe in the hardware magic. When an interrupt occurs, we just have to believe that somehow it can specify an interrupt level. When the CPU goes to process that interrupt, it sets the interrupt priority mask in the status register. Setting the IPM keeps the CPU from letting other, lower level, interrupt handlers interrupt. When the interrupt is done, the IPM is restored.

Normally, interrupts at the same level don't interrupt each other. The one exception is interrupt level 7. It is known as a NMI (non-maskable interrupt). Level 7 interrupts can't be masked.

### Assigning vectors

I'm still going to handwave the question of which interrupt handler is chosen. However, you can see that the IO device can pass data to the CPU over the various control lines to figure this out.