Advanced Computer Architecture

Lecture No. 28

Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 8
Computer Systems Design and Architecture — 8.3

Summary

- Comparison of Interrupt driven I/O and Polling
- Design Issues
- Interrupt Handler Software
- Interrupt Hardware
- Interrupt Software

Comparison of Interrupt driven I/O and Polling

Interrupt driven I/O is better than polling. In the case of polling a lot of time is wasted in questioning the peripheral device whether it is ready for delivering the data or not. In the case of interrupt driven I/O the CPU time in polling is saved.

Now the design issues involved in implementation of the interrupts are twofold. There would be a number of interrupts that could be initiated. Once the interrupt is there, how the CPU does know which particular device initiated this interrupt. So the first question is evaluation of the peripheral device or looking at which peripheral device has generated the interrupt. Now the second important question is that usually there would be a number of interrupts simultaneously available. So if there are a number of interrupts then there should be a mechanism by which we could just resolve that which particular interrupt should be serviced first. So there should be some priority mechanism.

Design Issues

There are two design issues:

1. Device Identification
2. Priority mechanism

Device Identification
In this issue different mechanisms could be used.

- Multiple interrupt lines
- Software Poll

- Daisy Chain

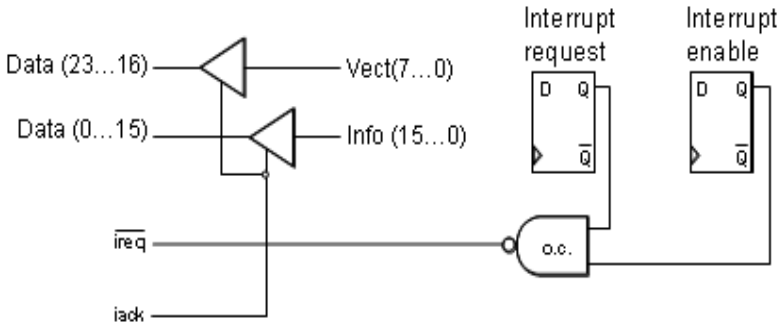
1. Multiple Interrupt Line

This is the most straight forward approach, and in this method, a number of interrupt lines are provided between the CPU and the I/O module. However, it is impractical to dedicate more than a few bus lines or CPU pins to interrupt lines. Consequently, even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it. Thus on each line, one of the other technique would still be required.

2. Software Poll

CPU polls to identify the interrupting module and branches to an interrupt service routine on detecting an interrupt. This identification is done using special commands or reading the device status register. Special command may be a test I/O. In this case, CPU raises test I/O and places the address of a particular I/O module on the address line. If I/O module sets the interrupt then it responds positively. In the case of an addressable status register, the CPU reads the status register of each I/O module to identify the interrupting module. Once the correct module is identified, the CPU branches to a device service routine which is specific to that particular device.

Simplified Interrupt Circuit for an I/O Interface

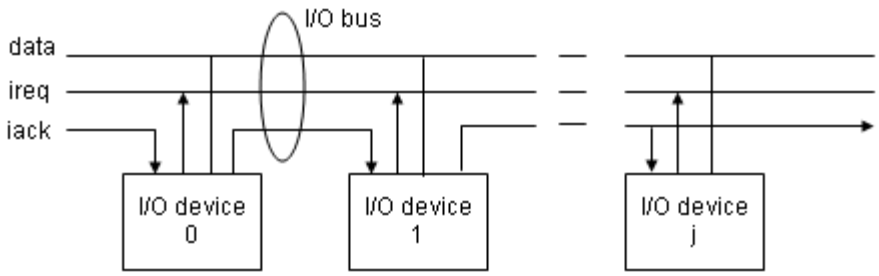


For above two techniques the implementation might require some hardware. The hardware would be specific to the processor which is being used. For example, for the case of SRC, simple hardware machanism is indicated. Now the basic technique is handshaking and in this case of handshaking, the peripheral device would initiate an interrupt. This interrupt needs to be enabled. We will have a mechanism of ANDing the two signals. One is interrupt enable and other is interrupt request. Now these two requests would be passed on the CPU. The CPU passes on the acknowledge signal to the device. The acknowledge signal is shared and it goes on to different devices.

The information about interrupt vector is given in 8-bits, from bit 0 to 7, which is translated to bit 16 to 23 on the data bus. Now the other 16-bits, from 0 to 15 are mapped to the data lines from 0 to 15. Now both of these are available through the tri-state buffers, which would be enabled through interrupt acknowledge.

3. Daisy Chain

The wired or interrupt signal allows several devices to request interrupt simultaneously. However, for proper operation one and only one requesting device must receive an acknowledge signal, otherwise if we have more than one devices, we would have a data bus contention and the interrupt information would not be resolved. The usual solution is called a daisy chain. Assuming that if we have jth devices requesting for interrupt then first device 0 would receive the acknowledge signal, so therefore, iack0=iack. The next device would only receive an acknowledge i.e., the jth device would receive an acknowledge if the previous device that means j-1 does not have an enabled interrupt request, that means interrupt was not initiated by the previous device. Now the figure shows this concept in the form of a connection from device 0 to 1. From 0, we see the acknowledge is generated for device 1, device 1 generates acknowledge for device2 and so on. So this signal propagates from one device to other device. Logically we could write it in the form of equation:



$$iack_j = iack_{j-1} \wedge (req_{j-1} \wedge enb_{j-1})$$

As we said that the previous device should not have generated an interrupt, that means its interrupt was not enabled and therefore, it passes on the acknowledge signal from its output to he next device.

Disadvantages of Software Poll and Daisy Chain

The software poll has a disadvantage is that it consumes a lot of time, while the daisy chain is more efficient. The daisy chain has the disadvantage that the device nearest to the CPU would have highest priority. So, usually those devices which require higher priority would be connected nearer to the CPU. Now in order to get a fair chance for other devices, other mechanisms could be initiated or we could say that we could start instead of device 0 from that device where the CPU finishes the last interrupt and could have a cyclic provision to different devices.

Interrupt Handler Software

Example using SRC

(Read from Book, Jordan page395)

Example using FALCON-A

As an example of interrupt-driven I/O, consider an output device, such as a parallel printer connected to the FALCON-A CPU. Now suppose that we want to print a document while using an application program like a word processor or a spread sheet. In this section, we will explain the important aspects of hardware and software for implementing an interrupt driven parallel printer interface for the FALCON-A. During this discussion, we will also explain the differences and similarities between this interface and the one discussed earlier. To make things simple, we have made the assumption that only one interrupt pin is available on the FALCON-A, and only one interrupt is possible at a given time with this CPU. Implications of allowing only one interrupt at a time are that

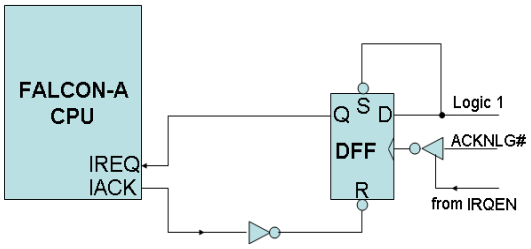
- No NMI is possible
- No nesting of interrupts is possible
- No priority structure needed for multiple devices
- No arbitration needed for simultaneous interrupts
- No need for vectored interrupts, therefore, no need of interrupt vectors and interrupt vector tables
- Effect of software initiated interrupts and internal interrupts (exceptions) has to be ignored in this discussion

Along with the previous assumption, the following assumptions have also been used:

- Hardware sets and clears the interrupt flag, in addition to handling other things like saving PC, etc.
- The address of the ISR is stored at absolute address 2 in memory.
- The ISR will set up a stack in the memory for saving the CPU's environment
- One ASCII character stored per 16-bit word in the FALCON-A's memory and one character transferred during a 16-bit transfer.
- The calling program will call the ISR for printing the first character through the printer driver.
- Printer will activate ACKNLG# only when not BUSY.

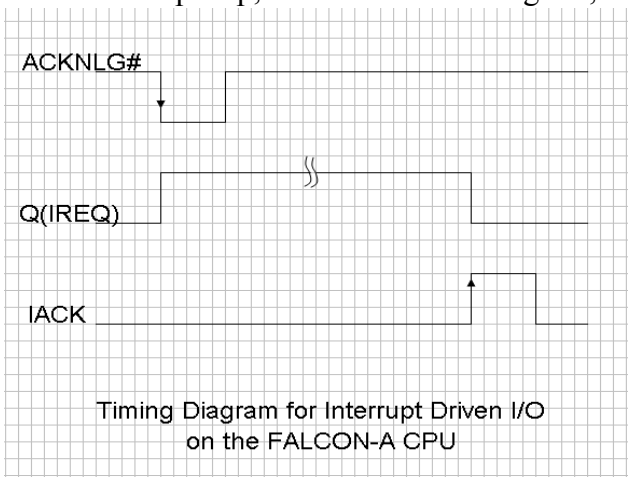
Interrupt Hardware:

The logic diagram for the interrupt hardware is shown in the Figure. The interrupt request is synchronized by handshaking signals, called IREQ and IACK. The timing diagram for the handshaking signals used in the interrupt driven I/O is shown in the next Figure. The printer will assert IREQ as soon as the ACKNLG#



Logic Diagram for Interrupt Driven I/O on the FALCON-A CPU

signal goes low (i.e. as soon as the printer is ready to accept new data) provided that IREQN=1. The processor will complete the current instruction and respond by executing the interrupt service routine. The inverting tri-state buffer at the clock input of the D flip flop is enabled by IRQEN. This will make sure that after the current print job is complete, additional requests on IREQ are disabled. This can happen as a result of the printer being available even through the user may not have requested a print operation. The IACK line from the CPU is connected to the asynchronous reset, R, of the D flip flop so that the same interrupt request from the printer is not presented again to the CPU. The asynchronous set input of the D flip flop, labeled S in the diagram, is permanently connected to logic 1. This will make sure that the flip flop will never be set asynchronously. The D input is also permanently connected to logic 1, as a result of which the flip flop will always be set synchronously in response to ACKNLG# provided IRQEN=1. Recall that IRQEN is bit 4 on the centronics control port at logical address 2, and this is mapped onto address 60 of the FALCON-A's I/O space. The rest of the hardware is case of the same as in the case of the programmed I/O example.



Timing Diagram for Interrupt Driven I/O on the FALCON-A CPU

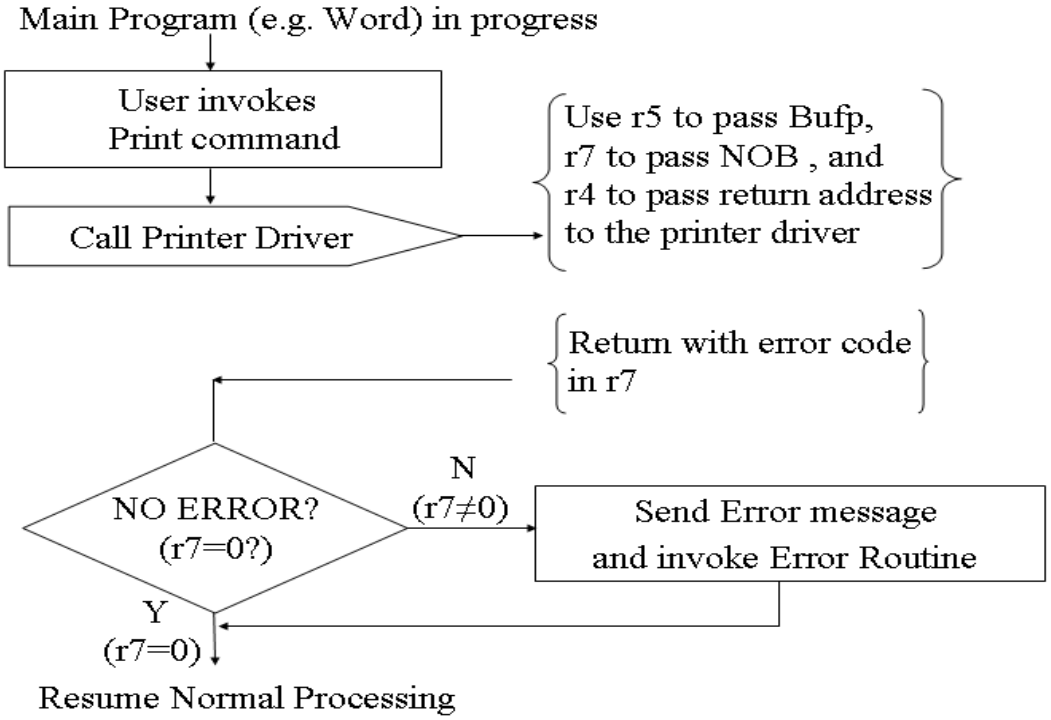
Interrupt Software:

Our software for the interrupt driven printer example consists of three parts:

1). Dummy calling program
2). Printer Driver
3). ISR

We are assuming that normal processing is taking place[19] e.g., a word processor is executing. The user wants to print a document. This

[19] Since only one interrupt is possible, a question may arise about the way the print command is presented to the word processor. It can be assumed that polling is used for the input device in this case.



document is placed in a buffer by the word processor. This buffer is usually present somewhere else in the memory. The responsibility of the calling program is to pass the number of bytes to be printed and the starting address of the buffer where these bytes are stored to the printer driver. The calling program can also be called the main program.

Suppose that the total number of bytes to be printed are 40. (They are placed in a buffer having the starting address 1024.) When the user invokes the print command, the calling program calls the printer driver and passes these two parameters in r7 and r5 respectively. The return address of the calling program is stored in r4. A dummy calling program code is given below.

Bufp, NOB, PB, and temp are the spaces reserved in memory for later use in the program. The first instruction is **jump [main].** It is stored at absolute memory address 0 by using the **.org 0** directive. It will transfer control to the main program. The first instruction of the main program is placed at address "**main**", which is the entry point in this example. Note that the entry point is different in this case from the reset address, which is address 0 for the FALCON-A. Also note that the address of the first instruction in the printer driver is stored at address "**a4PD"** using the **.sw** directive. This value is then brought into r6. The main program calls the printer driver by using the instruction **call r4, r6.** In an actual program, after returning from the printer driver, the normal processing resumes and if there are any error conditions, they will be handled at this point. Next, consider the code for the printer driver, shown in the attached text box.

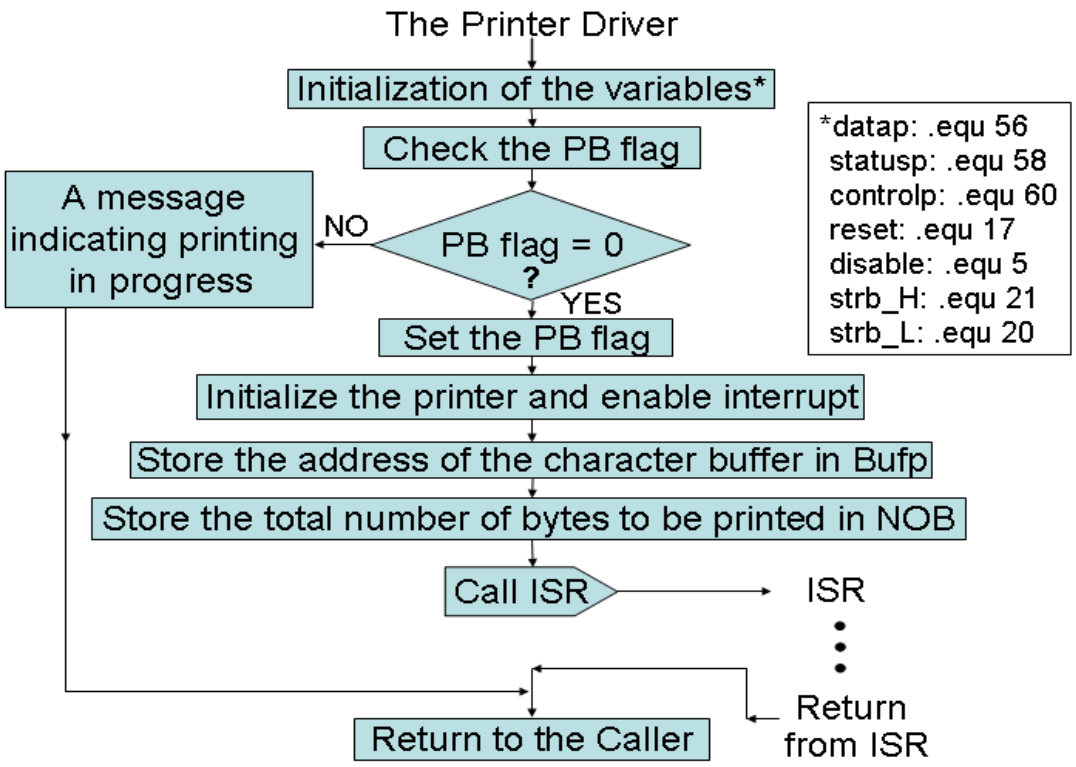
```
; filename: Example_Falcon-A .asmfa
;This program sends a single character
;to a FALCON-A parallel printer
;using an interrupt driven I/O interface
;
; Notes:
; 1. 8-bit printer data bus connected to
;    D<7..0> of the FALCON-A (remember big-endian)
;    Thus, the printer actually uses addresses 57, 59 & 61
;
; 2. one character per 16-bits of data xfered ;
;
    .org 0
    jump [main]
a4ISR:     .sw  beginISR
a4PD:      .sw  Pdriver
dv1:              .sw 1024
dv2:              .sw 40
Bufp:      .dw  1
NOB:       .dw  1
PB:        .dw 1
temp:      .dw 6
;
; Dummy Calling Program, e.g., a word processor
;
    .org 32
main:      load r6, [a4PD]        ;r6 holds address of printer driver
;
; user invokes print command here
;
    load r5, [dv1]          ;Prepare registers for passing
    load r7, [dv2]          ; information about print buffer.
;
;
; call printer driver
;
    call r4, r6
; Handle error conditions, if any , upon return.
; Normal processing resumes
;
    halt
```

The printer driver is loaded at address 50. Initialization of the variables includes setting of port addresses, variables for the STROBE# pulse, initializing the printer and enabling its IRQEN. The variables can be defined anywhere in the program because they reserve

no memory space. When the printer driver starts, the PB flag is tested to make sure that a previous print job is not in progress. If so, the ISR is not invoked and a message is returned to the main program indicating that printing is in progress. This may display a "printer busy" icon on the user's screen, or cause some other appropriate action. If the printer is available, it is initialized by the driver. The following activities are also performed by the driver (see the attached flow chart also).

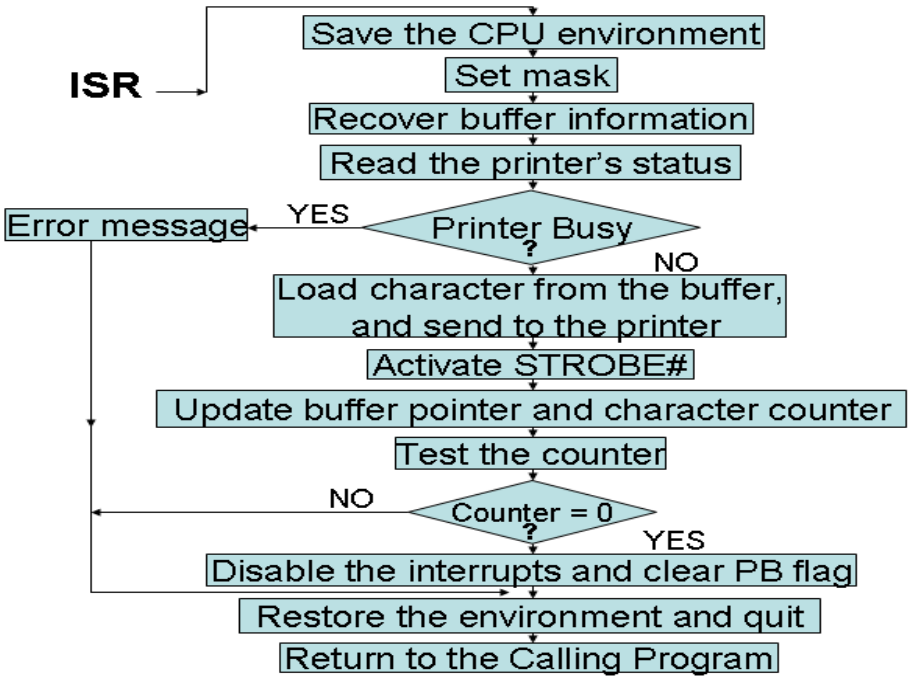
- Set port addresses
- Set up variables for the STROBE# puls
- Initialize printer and enable its IRQEN.
- Set up printer ISR by pointing to the buffer and initializing counter
- Make sure that the previous print job is not in progress
- Set PB flag to block further print jobs till current one is complete
- Invoke ISR for the first time
- Pass error message to main program if ISR reports an error
- Return to main program

The code and flow chart for the interrupt service routine (ISR) are discussed in the next few paragraphs.



```
; Printer driver
;
        .org 50                 ; starting address of Printer driver
;
datap:          .equ 56
statusp:        .equ 58
controlp:       .equ 60
;
reset:          .equ 17         ; or 11h
; used to set unidirectional, enable interrupts,
; auto line feed, and strobe high
disable:        .equ 5
;
strb_H:         .equ 21         ; or 15h
strb_L:         .equ 20         ; or 14h
;
; check PB flag first, if set,
; return with message.
;
Pdriver: load r1, [PB]
        jnz r1, [message]
        movi r1, 1
        store r1, [PB]          ; a 1 in PB indicates Print In Progress
        movi r1, reset          ; use r1 for data xfer
        out r1, controlp
        store r5, [Bufp]
        store r7, [NOB]
;
;
        int
;
        jump [finish]
message: nop                    ; in actual situation, put a message routine here
                                ;to indicate print in progress
finish:  ret r4
;
```

We have assumed that the address of the ISR is stored at absolute memory address 2 by the operating system. One way to do that is by using the **.sw** directive (as done in the dummy calling program). The symbol **sw** stands for "storage of word". It enables the user to identify storage for a constant, or the value of a variable, an address or a label at a fixed memory location during the assembly process.



These values become part of the binary file and are then loaded into the memory when the binary file is loaded and executed. In response to a hardware interrupt or the software interrupt **int,** the control unit of the FALCON-A CPU will pick up the address of the first instruction in the ISR from memory location 2, and transfer control to it. This effectively means that the behavioral RTL of the **int** instruction will be as shown below:

int IPC← PC, PC ← M[2], IF ← 0

The IPC register in the CPU is a holding place for the current value of the PC. It is invisible to the programmer. Since the **iret** instruction should always be the last instruction in every ISR, its behavior RTL will be as shown below:

iret PC ← IPC, IF ← 1

The saving and restoring of the other elements of the CPU environment like the general purpose registers should be done within the ISR. The five **store** instructions at the beginning are used to save these registers into the memory block starting at address **temp**, and the five **load** instructions at the end are used to restore these registers to their original values.

```
;   ISR starts here
```

```
      .org 100
beginISR: movi r6, temp
      store r1, [r6]
      store r3, [r6+2]
      store r4, [r6+4]
      store r5, [r6+6]
      store r7, [r6+8]
      movi r3, 1
      shiftl r3,r3,7          ; to set mask to 0080h
      load r5, [Bufp]         ; not necessary to use r5 & r7 here
      load r7, [NOB]          ; using r7 as character counter
      in r1, statusp
      and r1,r1,r3            ; test if BUSY = 1 ?
      jnz r1, [error]         ; error if BUSY = 1
      load r1, [r5]           ; get char from printer buffer
      out r1, datap
      movi r1, strb_L
      out r1, controlp
      movi r1, strb_H
      out r1, controlp
      addi r5, r5, 2
      store r5, [Bufp]        ; update buffer pointer
      subi r7, r7, 1          ; update character counter
      store r7, [NOB]
      jz r7, [suspend]
      jump [last]
suspend: store r7, [PB]       ; clear PB flag
      movi r1, disable        ; disable future interrupts till
      out r1, controlp        ; printer driver called again
      jump [last]
error: movi r7, -1            ; error code in r7
; other error codes go here
;
last: load r1, [r6]
      load r3, [r6+2]
      load r4, [r6+4]
      load r5, [r6+6]
      load r7, [r6+8]
      iret
      .end
```

After setting the mask to 80h in r3, the current value of the buffer pointer and the number of bytes to be printed are brought from the memory into r5 and r7 respectively. After a byte is printed, these values are updated in the memory for use by the ISR when it is invoked again. The rest of the code in the ISR is the same as it was in case of the programmed I/O example. Note that we are testing the printer's BUSY flag within the

ISR also. However, the difference here is that this testing is being done for a different reason, and it is done only once for each call to the ISR.

Memory Map for our ISR

0	Entry Point
2	Address of ISR
4	Data and Pointer Area
32	Main Program (Dummy Calling Program)
50	Printer Driver
100	ISR
	. . .
1024	Print Buffer
	. . .

The memory map for this program is as shown in the Figure. The point to be noted here is that the ISR can be loaded anywhere in the memory but its address will be present at memory location 2 i.e. M[2].