WATCOM
Waterloo Computing Systems Limited
158 University Ave. W., Waterloo, Ontario Canada N2L 3E9
(519) 884-2700

infoWAT

Volume 1, Issue 6

May 1983

Waterloo Software, Version 1.2

Version 1.2 of the Waterloo software for the IBM Personal Computer is now available. The following improvements have been made:

General
- significantly faster screen I/O and disk I/O operations enhance Editor performance and program execution
- the BACKTAB and ERASE-TO-END-OF-SCREEN keys are supported
- several problems of a minor nature have been corrected in APL, COBOL and FORTRAN
- appending to text files has been fixed
- the Waterloo Serial Adapter Board is no longer necessary to use the Waterloo software

EDITOR
- two new function keys allow the user to (i) reset the monitor to default characteristics, (ii) display the function key layout on the screen
- talk (terminal) mode handles data rates up to 9600 baud and supports new key combinations to switch between APL and ASCII
- the "stand-alone" EDITOR has improved support for APL overstrikes

BASIC
- quotes around filenames are optional for RUN, DIRECTORY, LOAD, MERGE, OLD, SAVE, STORE and TYPE commands
- the CHAIN statement with the "NAMES" option now passes matrices correctly
- matrices can be passed as parameters to a function or procedure

Version 1.2 of the IBM PC software is available from

WATSOFT Products Inc.
158 University Avenue West
Waterloo, Ontario
(519) 886-3700

Version 1.2 software has been made available as an update, free of charge, to purchasers of the corresponding earlier IBM PC versions.

Interrupt Handling on the SuperPET

Frequently we receive requests for additional information on interrupt handling in the SuperPET. This article will describe how to incorporate a user-written interrupt handler into the system.

The MC6809 microprocessor chip processes interrupts from programs or devices by selecting an address from the read-only memory (ROM) locations at the high end of memory ($FFF0-$FFFF). Different types of interrupts cause different addresses to be selected. The memory locations and the type of interrupt which selects the memory location are listed below.

```
$FFF0 RESV Reserved For Future Use
$FFF2 SWI3 Software Interrupt 3
$FFF4 SWI2 Software Interrupt 2
$FFF6 FIRQ Fast Interrupt Request
$FFF8 IRQ  Interrupt Request
$FFFA SWI  Software Interrupt
$FFFC NMI  Non-Maskable Interrupt
$FFFE RSET Reset
```

Thus if an IRQ type of interrupt occurs, the 6809 processor looks at locations $FFF8 and $FFF9 for the address of the routine that handles this particular kind of interrupt.

Let's look briefly at each type of interrupt. Some of them are caused by hardware (electrical signals). Others can be caused by software. RESV is reserved by the designer of the 6809 processing unit for future use. All of SWI, SWI2, and SWI3 are caused when a "software interrupt" instruction of the appropriate type is executed by the processor. The names of the instructions are SWI, SWI2 and SWI3. A FIRQ interrupt is caused by a signal on the FIRQ pin of the 6809 processor chip. Similarly, IRQ and NMI are caused by a signal on the IRQ and NMI pins of the 6809 processor chip. The last one in the list above is RESET. This interrupt occurs when the SuperPET is switched on and the 6809 CPU is selected or when a switch is made from the 6502 microprocessor to the 6809. The RESET interrupt causes software in the ROM to initialize the operating system. More information on interrupts can be found in most 6809 microprocessor handbooks.

In ROM is a very simple version of an interrupt handler, called the "First Level Interrupt Handler" or FLIH. The FLIH handles all interrupts except for RESET by dispatching them through special locations in random access memory (RAM). A second vector of addresses, which corresponds to those in ROM, is used to determine the address of the routine that actually handles the interrupt. This vector of 7 addresses is found at location $0100. It is initialized by the RESET interrupt handler. Each location is assigned as follows:

```
$0100 RESV Reserved For Future Use
$0102 SWI3 Software Interrupt 3
$0104 SWI2 Software Interrupt 2
$0106 FIRQ Fast Interrupt Request
$0108 IRQ  Interrupt Request
$010A SWI  Software Interrupt
$010C NMI  Non-Maskable Interrupt
```

Thus if the FLIH determines that an IRQ type of interrupt has occured it calls the subroutine whose address is stored in locations $0108 and $0109. A very important thing to note here is that the actual interrupt handler is called as a subroutine. The interrupt handler, whether it is the one supplied in ROM or one written by you, the programmer, must execute a "return from subroutine" or RTS instruction. It is the FLIH that eventually executes a "return from interrupt" or RTI instruction.

All known possible sources of interrupts are handled by various routines in the ROM library. If a new device is added to the system and if this device can cause an interrupt then the user must add an interrupt handler for this device to the system. As well, the user may wish to supersede an existing interrupt handler because of some deficiency in its support of a particular device.

Let's take a case in point. The existing IRQ handler looks after several devices that can cause an IRQ type of interrupt. The IRQ handler determines which device caused the interrupt by examining the "status register" of every device in the system. For example, one of the devices, called a Programmable Interface Adapter (PIA), causes an IRQ interrupt many times per second. The IRQ handler calls a clock interrupt handling routine when the PIA is recognized as the source of the interrupt. A scan of the keyboard is also performed at this time to determine if a key has been pressed.

The 6551 Asynchronous Communications Interface Adapter (ACIA) can also cause an interrupt. Apart from acknowledging that the ACIA caused the interrupt, nothing else is done about it. A user who is not satisfied with this treatment of ACIA interrupts may wish to supply a better routine. The new routine must take over the handling of IRQ interrupts. Since it is the intention of the programmer to only handle interrupts from the ACIA, the routine should check for an interrupt from this device, process the interrupt if there is one and otherwise let the normal IRQ handler take care of all other interrupts (such as those from the PIA). In this way the user routine gets "first crack" at an IRQ interrupt. The process of checking for an interrupt from a particular device has the side effect of clearing the interrupt condition of that device.

Let's look at the following segment of assembler code as might be found in a representative user-written interrupt handler.

```
; User-written Interrupt Handler
; - test status reg for interrupt
; - if ACIA interrupted then
; -     call ACIA handler
; - else
; -     do other IRQ processing
;       provided by ROM routines
        xref    IRQHndlr
ACIA    equ     $EFF0
IOR     equ     0
STATR   equ     1
CMDR    equ     2
CNTLR   equ     3
INTERRUPT equ   $80
        xdef    MyIRQHndlr
MyIRQHndlr      equ     *
        LDB     ACIA + STATR
        ANDB    #INTERRUPT
        if ne
            JSR MyACIAHandler
        else
            JMP [IRQHndlr]
        endif
        RTS
        end
```

The details of the user-written ACIA handler are beyond the scope of this article. We should now consider the mechanism for installing the user interrupt handler as the one to be called by the operating system. We wish to remember the address of the SuperPET's IRQ handling routine since we must let it handle any IRQ interrupts that we are not interested in handling ourselves. If this is not done then the system will most likely crash. The following are examples of user-written "connect to interrupt" and "disconnect from interrupt" routines.

```
IntVctr equ     $0100
IRQ     equ     8
; Connect to IRQ Interrupts
; - save current handler address
; - call system routine to connect
;   to IRQ type interrupts
        xref    MyIRQHndlr
        xref    ConBInt_
        xdef    Connect
Connect equ     *
        LDD     IntVctr + IRQ
        STD     IRQHndlr
        LDD     #IRQ
        PSHS    D
        LDD     #MyIRQHndlr
        JSR     ConBInt_
        LEAS    2,S
        RTS
; Disconnect from IRQ Interrupts
; - restore address of previous
;   IRQ handler
        xdef    Disconnect
Disconnect equ  *
        LDD     IRQHndlr
        STD     IntVctr + IRQ
        RTS
; Place to save previous
; IRQ handler address
        xdef    IRQHndlr
IRQHndlr fdb    0
        end
```

The ROM library routine "ConBInt_" is used to establish an interrupt handling routine by specifying the address of the interrupt handling code and the type of interrupt it will handle (e.g., SWI, NMI, IRQ, etc.). The Waterloo 6809 Assembler manual describes the routine "ConBInt_". Use of this system routine allows us to place the interrupt handler in bank-switched memory should we wish to do so.

The interrupt handler described above consists of three routines. The "connect" routine causes the interrupt handler to be incorporated into the system. The "disconnect" routine causes the interrupt handler to be removed from the system. The "handler" routine is invoked whenever an IRQ type of interrupt occurs. The procedure for handling other types of interrupts is similar.

The interrupt handler may be loaded into the high address range of RAM by linking it with a suitable origin (ref., Linker ORG statement). To ensure that it remains there without being "walked over" by a language interpreter the handler's initialization code should alter the high memory address limit. The low and high memory address limits are used by the language interpreters as the bounds within which a user program (e.g., a BASIC program) may reside. For those users of APL it should be noted that a workspace that has been previously saved will not be "compatible" with a smaller work area. In this case the workspace must be "copied" into a "clear" workspace in memory. The following is an example of a typical initialization routine.

```
; Initialization
; - alter highest address that
;   languages may use but
;   don't bother protecting
;   initialization routine
; - set menu "EXIT" code
; - return to system menu
MemBeg  equ     $20
MemEnd  equ     $22
Service equ     $32
        xref    Connect
        xdef    Init
Init    equ     *
        LDD     #Connect - 1
        STD     MemEnd
        CLR     Service
        RTS
        end
```

Using the linker, the various components of the interrupt handler must be combined such that the "init" routine is first in memory, the "connect" routine and "disconnect" routines come next, and the "handler" would come last. The following is an example of a typical linker "command" file.

```
"handler"
org $7F00
"init.b09"
"condis.b09"
"handler.b09"
```

The handler is loaded into memory from the menu by typing in the filename of the executable module. In the above example, this would be "disk.handler.mod". A picture of the memory layout of the SuperPET after loading the handler from the menu follows.

```
$0000 .
$0020 (0A00) MemBeg
$0022 (7F07) MemEnd
      .
$0A00 .
      .
$7F00 . Init
$7F08 . Connect
      . Disconnect
      . Int. Handler
$7FFF . End of Handler
$8000 . Start of Screen Memory
```

The language interpreters all support a "sys" or "usr" function which allows you to call machine language subroutines. You are now ready to "sys" to the "connect" and "disconnect" routines whenever you wish to start or stop handling of interrupts by your own interrupt handler. Communication of information between the handler and a program written in one of the languages could be done using the "peek" and "poke" facilities of the language.