NI-CD NOTES

I've had a PX-8 for almost a year, now. In that time, largely because I've had some trouble with them, I've learned quite a bit about the Ni-Cd batteries that power this computer and its portable peripherals. Following is a summary of what I've been able to glean, from reading and from observation, about Ni-Cd batteries in general and about how they are handled within the PX-8 in particular.

The PX-8's Batteries

The PX-8 main unit contains two rechargeable batteries. The main battery supplies power when you are using the unit and is recharged with an AC adapter that provides about 6 volts DC. This battery is rated at 1100 mAH at a nominal 4.8 volts. The backup battery supplies power when the unit is switched off or turns itself off. This backup power keeps data and programs stored in the PX-8's main memory intact. The backup battery is rated at 90 mAH, also at a nominal 4.8 volts. To insure that the backup battery is always in a fully charged state when it is needed, it is recharged in one of two ways: either by the AC adapter or, if the PX-8 is on but the adapter is not connected, by the circuitry that powers the LCD screen.

The Batteries of Some PX-8 Peripherals

PF-10 Battery
Like the PX-8 main battery, the PF-10 disk drive battery is rated at 4.8 volts, 1100 mAH. It, too, appears to be built from four rechargeable sub-C batteries. The PF-10 also has a backup battery, said to be good for about 10 minutes of operation after the main battery runs down. This battery looks identical to the battery on the main PX-8 board. It is soldered to the PF-10's main printed circuit board.

RAM Disk and Multi-Unit Battery
The RAM disk battery is rated at 4.8 volts, 450 mAH. It is built from four 'AA' size cells.

CX-20 Modem Battery
The external modem battery is identical to the RAM disk battery, except that it uses a slightly different connector.

General Characteristics of Ni-Cd Batteries
Ni-Cd batteries come in a variety of basic types. These include:

Resealable Vented and One-shot Vented
When sufficiently overcharged or over-discharged, Ni-Cd batteries can build up enough internal pressure to cause a vent to open. This releases the pressure and prevents an explosion, but it also releases some of the battery's electrolyte. Some Ni-Cd batteries are built with one-shot vents; these vents, once open, do not reseal, and the battery eventually dries out. Many modern Ni-Cd batteries are built with vents that will reseal once the situation creating the excess pressure has been corrected. These batteries may be somewhat reduced in capacity once they have vented (as is a lead-acid battery that has been allowed to run low on acid), but they will, in many cases, still be usable for quite a while. Cells that have vented will have white crystals growing near the positive contact or will show other obvious signs of leakage.

Normal Charge and Fast Charge
Some Ni-Cd batteries can only be recharged at a rate less than or equal to the so-called "10 hour rate," which is one-tenth the rated ampere-hour capacity of the battery (C/10). For example, the 10 hour rate for the 1100 mAH PX-8 main battery is 110 mA. Due to inefficiencies in the charging process, the battery must be recharged at this rate for 14 to 16 hours to achieve full charge. Other batteries are designed to be able to withstand a charge rate equal to one-fourth their rated capacity (C/4). These batteries are capable of sustaining repeated "fast charges" without damage. The batteries used in the PX-8 appear to be of the "fast charge" variety, since they are intended to withstand a charge rate greater than C/10 without shortening their life.

Normal Capacity and High Capacity
Normal capacity Ni-Cd batteries will continuously operate a device for approximately the same period of time, on one charge, as ordinary dry cells. High-capacity batteries will operate the same device for a longer period of time. (For example, the Radio Shack high-capacity 'C' cells have almost twice the capacity of the standard Ni-Cd 'C' cell, and the high-capacity 'D' cells have nearly four times the capacity of standard Ni-Cd 'D' cells.)

How Ni-Cd Batteries Can Be Damaged

Ni-Cd batteries can be damaged by improper use (over-discharging or discharging too quickly) or by improper recharging (overcharging or charging too quickly). Overly discharging a Ni-Cd battery can cause the weaker cells to be reverse-charged by the stronger cells. This can permanently damage them by causing them to vent. (In general, no cell in a Ni-Cd battery should be permitted to fall below 1.0 volts.) Overcharging Ni-Cds at a rate up to one-tenth the rating of the battery may temporarily weaken them, but the damage can often be mostly reversed, as described below. On the other hand, overcharging at too great a rate, as wellas discharging at a rate greater than C/2, can cause Ni-Cd's to overheat. Overheating can lead to venting or can cause internal shorts to develop. (Internal shorts can sometimes be repaired by burning them out using the technique described below.)

Ni-Cd batteries are also subject to a "memory" effect. This can happen after the batteries have been subjected to repeated, identical partial discharge/full recharge cycles. A battery with this problem will appear to be discharged when it "remembers" the point in its discharge cycle at which it is usually recharged. This effect, too, can be reversed and the battery restored to nearly full capacity.

How the PX-8 Charges Batteries

Main Battery
The main battery is recharged by a 6-8 volt DC power source connected to the "ADAPTOR" plug on the back of the PX-8. A diode in the PX-8 charge circuit is intended to protect the battery from discharging if the power supply connected to the computer falls below the battery voltage, and to protect the computer from damage due to reverse polarity power supplies or short circuiting the power supply connector. The circuitry also contains a Zener diode which will shunt moderate overvoltage (though I wouldn't go out of my way to test any of this protection, since it, too, is capable of being damaged).
The computer is designed to recharge the batteries at a moderately fast rate (150-200mA) for eight or eleven hours, depending on whether the computer is turned off or on, respectively, when the charge cycle begins. After that, charge drops to a trickle charge of about 40 mA. This is intended to allow you to recharge the computer fairly quickly and keep it in a charged state, while at the same time avoiding overcharging the batteries if the charger is left connected.
Unfortunately, if you are not careful about how you recharge the batteries, the circuitry that is intended to protect against overcharge can work against you. The manual supplied with the PX-8 does not adequately warn against this.
Here's the problem: The PX-8 recharge is monitored by circuitry intended to protect against overcharge. It does so by detecting when external power is supplied to the unit and at that time initiating its 8 or 11-hour charge cycle. It does so regardless of the current state of charge of the battery. This means that if the battery is partially charged when the power supply is attached, it will be overcharged for some portion of the next eight or eleven hours, and it will be overcharged at the full charge rate. Repeated overcharges of this sort will eventually damage the batteries. Although modern Ni-Cd batteries can "take" continuous charging at the C10 rate (1/10th their current rating) without suffering permanent damage, the PX-8 charge rate considerably exceeds this.
Therefore, you should not recharge your PX-8 unless the battery voltage has dropped below about 5 volts. (The PX-8 will switch from trickle charge to full charge by itself, if the voltage drops below 5 volts while trickle charge is in effect.) If you do need to recharge before this has occurred, you will be doing your computer a favor if you do not leave the charger connected for the full 8-hour period, or if you take the time to run the

battery down by using the serial port or tape drive. Since the charge rate is boosted to full each time the power supply is connected to the rear of the computer, you should also not disconnect and reconnect the adapter once a charge cycle has begun. Each reconnection begins another full charge cycle.

Backup Battery
The backup battery is designed to always be at peak charge while the PX-8 is on, so that it can take over when the main battery needs to be charged. If the AC adapter is connected, the backup battery is recharged in the same fashion as the main battery (full charge/trickle charge). If the adapter is not connected and the computer is turned on, the backup battery is recharged through the internal power supply to the LCD screen.

RAM Disk Battery
The RAM disk or Multi-unit battery is recharged in a manner similar to the main battery, and is subject to the same full charge/trickle charge cycle. It can only be charged when the RAM disk is connected to the PX-8, since it does not have a separate external power supply connection.

PF-10 or External Modem Battery
The PF-10 and CX-20 external modem do not seem to have overcharge protection circuitry. Their batteries can be damaged by being continuously connected to a charger (I can say from personal experience that the PX-10 batteries will overheat if they are connected too long). If the batteries have not leaked due to overcharge, most of the apparent damage can be reversed by cycling the batteries through one or more charge/discharge cycles. If they have leaked, they should be replaced. This can be done by ordering a battery pack from an EPSON supplier, or much more cheaply by purchasing the appropriate replacement cells (from an electronics supplier or from Radio Shack) and building a new battery pack from the original one, as described elsewhere in this article.

How to Squeeze a Little More Power Out of PX-8 Batteries
You can use the technique recommended for restarting the PX-8 after you have changed the battery to get a few minutes more life out of a discharged PX-8 battery -- enough to finish most low-power operations. If the unit has turned itself off because the battery got too low, turn the power switch to "off" and let it sit for a few minutes. While holding down the reset button, turn the unit on and release the reset button. The PX-8 will probably come on, and it will function for another five or ten minutes.

PX-8 Ni-Cd DO's and DON'T's
- Don't short out the battery. Do be careful when you remove or replace the battery from the PX-8.

- Don't overcharge. This will happen if you attach the charger for the full 8 hours when the battery is not fully discharged. Wait for the "CHARGE BATTERY" messge, or use a battery monitor program like px.com, battery.com, or pxutil and don't recharge until the battery is near the cutoff point (4.7 or 4.8 volts).

- Do let the battery rundown completely before you recharge. Don't partially discharge and then fully recharge. This can produce the so-called "memory effect." The memory effect, which can occur after repeated, nearly identical partial discharge/full recharge cycles, causes the battery to appear discharged at the point the premature recharge cycles previously began, therefore giving shorter service than it should.

How Damaged Ni-Cd Batteries Can Be Restored
You can restore a weak battery if it has not become physically damaged (internal short or leakage of electrolyte). Usually, all you have to do is repeatedly put the battery through full discharge/full recharge cycles, preferably bringing the terminal voltage down to 4.0 volts (but not lower!) for a 4.8 volt battery. To restore the PX-8 batteries, try fully discharging and then recharging the computer a few times. If this doesn't improve things sufficiently, try bringing the battery down below the 4.7 volts at which the PX-8 normally cuts out. You can do this by first letting the computer run the batteries down, then disconnecting the battery and inserting the ends of a 100 or 220 ohm resistor in the connector that normally plugs into the PX-8. Monitor the battery voltage with a

voltmeter clipped to the resistor leads.
As a last-ditch effort, you can try to restore batteries that have one or more shorted cells by jolting them with a short burst of excess charge. This charge burns through small short circuits within the cell itself. You can supply this charge by building and applying the circuit below (from Hands-On Electronics, June '87).

```
                          __
25  +                   __/\__ blast button
vdc o---////-------+----o    o-----+------------o to voltmeter plus
      150 ohm      |               |
      1 watt       |       o blast |
      resistor     |      /        | blast/charge
                   |     /         | switch
                   +----/    o-----+
                   |        charge |
                   |               |
                   | +             | +
       10,000 mf -----           -----
       capacitor -----            ---  cell to be blasted
       (or more)   |               |
                   |               |
                   |               |
25  -              |               |
vdc o--------------+---------------+-----------o to voltmeter minus
```

The idea is to leave the switch on "blast" and let the capacitor charge, then hit the blast button a few times, waiting a few seconds between each hit. Then switch to "charge" and watch the voltmeter; if you have succeeded in repairing the damage to the battery, the battery voltage will rise. If not, you may as well repeat the process a couple of times, then put the battery through a normal charge/discharge cycle and see what happens. I have also accomplished the same thing by momentarily shorting a dead cell of a battery across a six volt power supply, though I can't guarantee the safety of this procedure. In any event, batteries brought back to life with this method are not likely to last as long as batteries that have not been damaged, and you are probably better off replacing them.

Replacements for PX-8 Batteries

The main battery is easy to replace when it wears out (that is, will no longer hold a charge for a reasonable period of time). It is built from four 1.2 volt sub-C cells. A replacement pack is available from Epson, but one can also be constructed much more cheaply from tabbed sub-C cells available from Radio Shack and other electronics suppliers. On the other hand, the backup battery is soldered to the main circuit board of the PX-8 and would be difficult for a PX-8 owner to replace, even if a substitute could be found.
A replacement battery pack for the PF-10 is available from Epson. The PF-10 is also designed to run off ordinary 'C' batteries; with a simple modification, described below, you can enable it to use and to recharge commonly available standard or heavy duty Ni-Cd 'C' cells. (The heavy duty batteries provide almost twice as much power as the factory batteries.)
The battery for a wedge (RAM disk or Multi-unit) can be replaced easily by removing the retaining screws of the RAM disk unit, unplugging the battery pack, and replacing defective or worn out cells with tabbed 'AA' cells available from Radio Shack or other suppliers. The CX-20 modem battery pack is similarly constructed and can be repaired in the same way.

Modifying the PF-10 to take High-Capacity Batteries

The following is adapted from a description by John Cooper. It describes how to modify a PF-10 disk drive to use Radio Shack 1.8AH 'C' Ni-Cd batteries (Radio Shack part #23-141). I have performed the same modification, following his directions.

The project requires four batteries. The modification enables the PF-10 to recharge these or other rechargable batteries as well as the original battery pack. Note, however, that you should not operate the PF-10 with the charger attached when you are using ordinary, disposable batteries after you make this modification to the PF-10.

1. Set the PF-10 on a flat surface with the back of the unit facing you.
2. Remove the single screw between the Serial plug and the Adapter plug.
3. Turn the unit on its back and remove the battery cover and battery pack.
4. Remove both of the most rearward PF-10 case screws. Slide the plastic battery carrier and rear cover combined out to the rear.
5. Carefully disconnect the plug connecting the battery carrier to the main printed circuit board.
6. Turn the battery carrier over so you can see the small printed circuit board. Pry the board up a little to free the wires, then de-solder the white wire from this circuit board. Store the wire under the tape that dresses the plug wires on the carrier.
7. Connect the now-empty pad where the white wire was to the red wire pad, and then to the brown wire pad.
8. Reassemble. Make sure you plug the battery carrier wires into the main PCB the same way they came off, and don't forget the ON/OFF switch cover. A little organizing of the wiring harness may be necessary to keep it clear of the Serial cable sockets.

The PF-10 should run about twice as long between charges (though it will also take about twice as long to recharge). The new batteries should also be much less succeptible to damage from overcharge, since the charge rate of the charger that comes with the PF-10 charges the new batteries at less than the C/10 rate for these cells.

David Bookbinder
West Medford, MA
October 10, 1987

Direct communication with the 8251 chip on the PX-8 computer

by Earl Evans and Chris Rhodes

The PX-8 lap computer is equipped with an 8251 serial interface chip. BIOS subroutines are available to use the features of this chip; however, at times it may be desirable to communicate directly with the chip through the Z80 I/O ports. Before doing this, there are a few problems that must be recognized and solved.

First, we recognize that the BIOS functions are performing a considerable amount of functions transparent to the user. When speaking directly to the 8251, you must perform some of these functions yourself. These include turning on the RS232 port power, initialing the baud rate, and setting the RS232 parameters.

Second, the 8251 is normally interrupt driven, and the interrupts are serviced by the BIOS routines. If these interrupts are left enabled, they will maliciously swipe the characters from the I/O port. Disabling the interrupts is a must.

Third, we must make the system think that the port is open, so that it will not inadvertently attempt to open it during our conversation with the 8251. This will cause the interrupts to become enabled again.

Fourth, the memory map of the PX-8 CP/M maintains mirror images of the values currently in use in the 8251 chip. The BIOS routines update these automatically; however, we must update them ourselves, since we are circumventing the BIOS routines.

The following is a portion of the source code for DIOMODEM, a version of the MODEM program. This particular version of MODEM implements direct communication with the 8251 based on the principles we previously discussed.

```
MAPLEWB         EQU     0EC03H   ; PX-8 CP/M warm boot location

CTRL1           EQU     0F0B0H          ;\
CTRL2           EQU     0F0B2H          ; \
IER             EQU     0F0B3H          ;  | system storage locations
RS232MODE       EQU     0F6D0H          ; /
RS232CMND       EQU     0F6D1H          ;/
RSOPFLG         EQU     0F2C8H


;*************************************************************
;
; Handles in/out ports for data and status

;IN$MODCTL1 places the current control status in the A register.

IN$MODCTL1:

        IN      A,(0DH)
        RET

;OUT$MODDATP sends the contents of the A register out the data
;port

OUT$MODDATP:

        OUT     (0CH),A
        RET

;IN$MODDATP places the incoming character into the A register

IN$MODDATP:
        IN      A,(0CH)
        RET

;Calling ANI$MODRCVB and CPI$MODRCVR in sequence will set the Z
;flag if there is an incoming byte from the modem and reset the Z
;flag if not.

ANI$MODRCVB:

        AND     2            ;test this bit for receive ready
        RET

CPI$MODRCVR:

        CP      2
        RET

;Calling ANI$MODSNDB and CPI$MODSNDR in sequence will set the Z
;flag if it is all right to send out a character, and will reset
;the Z flag if not.

ANI$MODSNDB:
                                ;bit to test for send ready
        AND     1
        RET

CPI$MODSNDR:

        CP      1
        RET


; INITIALIZE THE PX-8 RS232 PORT

INITMOD:

; INITIALIZE THE PX-8 RS232 PORT

; let's close the port first

        CALL    MAPLEWB+3CH     ; close RS232 port
        LD      A,00H           ;
        LD      (RSOPFLG),A     ; Store 0 in RSOPFLG (makes the
                                ; system think the RS232 is open)

; Disable any 8251 interrupts

        LD      A,(IER)         ; get current interrupt enable state
        RES     1,A             ; clear RX interrupt
        RES     2,A             ; clear CD interrupt
        OUT     (4),A
        LD      (IER),A

; Turn power on with inhibit

        LD      A,(CTRL2)       ; get current value - lights, etc.
        SET     4,A             ; inhibit TX
        OUT     (2),A           ; send it out
        CALL    ST1ML           ; delay, says the BIOS
        SET     3,A             ; power on
        OUT     (2),A           ; send it out
        CALL    ST100ML
        RES     4,A             ; enable TX
        SET     5,A             ; enable internal RX
        OUT     (2),A           ; send it out
        LD      (CTRL2),A       ; save new values

; insure baud rate clock set

        LD      A,(CTRL1)       ; get current value baud rate
        OUT     (0),A

; force reset of 8251
```

```
        LD      A,0
        OUT     (0DH),A
        LD      A,0
        OUT     (0DH),A
        LD      A,0
        OUT     (0DH),A
        LD      A,40H           ; reset command
        OUT     (0DH),A
        CALL    ST100ML         ; delay, says the BIOS

; Set up 8251

        LD      A,(RS232MODE)   ; get current mode value
        OUT     (0DH),A
        LD      A,(RS232CMND)   ; get current command value
        OUT     (0DH),A

        RET

ST10US:                         ; delays about 10 uS
        PUSH    AF
ST10US1:
        DEC     BC
        LD      A,B
        OR      C
        JR      NZ,ST10US1
        POP     AF
        RET
ST1ML:                          ; 1 millisecond
        PUSH    BC
        LD      BC,100
        CALL    ST10US
        POP     BC
        RET
ST100ML:                        ; 100 milliseconds
        PUSH    BC
        LD      BC,10000
        CALL    ST10US
        POP     BC
        RET
```

* * *

Using the PX-8 with Asynchronous Communication Packages

by Christopher Rhodes

The PX-8 has built-in BIOS support for the RS-232-C port which is controlled by the 8251. These BIOS routines control all aspects of operation from setting the communication protocol to setting the address and size of the interrupt serviced input buffer. An alternative to BIOS calls is direct communication with the 8251. The article "Direct Communication with the 8251 Chip on the PX-8 Computer" (printed in the July 20 Mailbag) describes that procedure. This paper covers the use of the built-in BIOS calls for RS-232-C support and discusses the advantages and disadvantages of both methods. It includes recommendations for patching programs using either procedure.

Using BIOS Calls

The following BIOS calls are provided by the operating system:

```
RSOPEN
RSCLOSE
RSIN
RSINST
RSOUT
RSOUTST
RSIOX
```

These BIOS calls are used instead of the direct communication with the system UART normally performed by most machines. RSOPEN and RSCLOSE are responsible for the hardware and software initialization required before and after use of the RS-232-C port. Remember power must be supplied to the 8251 before it is used. The initialization is performed after the parameters chosen in CONFIG or those chosen by the general RS-232-C routine, RSIOX. RSIN and RSINST cover character input and input status, RSOUT and RSOUTST cover character output and output status. RSIOX can be used to accomplish any of the above individual calls and also can be used to determine the remaining chip status states such as parity, overrun, and framing errors. RSIOX is also used to configure the size and location of the buffers used by the 8251. Please refer to the PX-8 User's Manual for details.

There is a one to one correspondence between the BIOS calls and calls you would normally do with direct I/O to the UART.

```
BIOS Call            Assembly Routine

RSOPEN       -       General routine to initialize etc.

RSCLOSE      -       Any clean up.  Not normally required

RSIN         -       IN      A,(DATAPORT)

RSINST       -       IN      A,(STATPORT)
                             AND     RECRDYBIT

RSOUT        -       OUT     (DATAPORT),A

RSOUTST      -       IN      A,(STATPORT)
                             AND     TXRDYBIT

RSIOX        -       Not normally supported except for other
                     status information
```

Program Patching

RS-232-C support using the BIOS calls is no more difficult than direct I/O routines. In many ways your task is easier. Whenever you would do direct I/O to a UART, simply replace the code with a BIOS call. For example, to patch a modem program you would need routines to send and receive characters and to test the status of the chip prior to performing these operations. The following code would be used:

```
   PUSH BC
   PUSH DE
   PUSH HL
   CALL RSIN            ; or RSOUT or RSINST or RSOUTST
   POP  HL
   POP  DE
   POP  BC
   RET
```

Remember that the BIOS uses many of the registers, so if your program assumes a register will be unaffected by a simple IN or OUT to a port, it may not work properly unless you preserve the environment across the BIOS call. The BIOS call is also slower than DIRECT I/O to a port. Any programs basing timing loops on the port access time may need to be modified to account for the time difference.

Which to Use

There are advantages to both DIRECT I/O and BIOS calls. At the outset, realize that the Geneva cannot be used if the only things a program allows you to specify are the data and status ports and which bits to check. As discussed in "Direct Communication with the 8251," there are many other concerns; such as memory locations affected by interrupt routines, power to the 8251, etc.; which must also be performed before simple IN and OUT routines to the 8251 will work. Such limitations determine your ability to modify the source code directly or to write your own drivers and overlay or include them in the program.

The system BIOS calls are recommended for several reasons. They have already been written and debugged, and are available for your use. They allow all the necessary control without any additional coding. Since the input is being handled by interrupt, you can eliminate many data loss problems. You may specify your own buffer length and location. The BIOS calls support several types of handshaking such as XON/XOFF.

One disadvantage of the BIOS calls is the rather excessive overhead incurred, especially if your only need is to get data to and from the port. However, in many applications this will not be a concern, and in fact, there will be times when your application will run more quickly with the system calls.

The direct I/O offers the advantage of laying the code and operation of the chip before you. It leaves no question of what the BIOS may be doing. However, this simplicity is offset by the tricks you must perform to fool the BIOS into letting you take command of the chip. The pitfalls are numerous.

An example using the MODEM program may prove useful. I patched one version of MODEM724 to use BIOS calls and patched one to perform direct I/O to the 8251. I had initially expected the direct I/O version to produce the better results. The outcome was quite surprising.

The direct I/O version could not be made to run much faster than about 2400 baud for file transfer. The overhead of the program and the limited speed of the Z80 led to dropped characters and numerous retry conditions at speeds above 2400 baud. The BIOS version operated reliably at speeds up to and including 19.2K baud. I did not clock the overall transfer time, but doubt that it was significantly different. The BIOS version seemed faster and the ability to run at all the supported baud rates is desirable. I think the interrupt buffered input is the reason the BIOS version ran at the higher speeds.

I recommend you use the BIOS calls whenever possible and use the DIRECT I/O only as a last resort. There is no coding advantage to using DIRECT I/O because all the special Geneva support will have to be included in special drivers that will have to be written and added to source code or overlayed on executable code.

*   *   *   *   *

Using the Bar Code Reader on the PX-8

Introduction

Bar code input on the PX-8 occurs at the standard bar code 3-pin plug connector while using any normal bar reader wand.

The architecture and addressing scheme of the PX-8 interface is different from the HX-20, which uses the 6301 internal timer logic to capture bar code input. While there is a 6301 slave CPU in the PX-8, it is not used for PX-8 bar code operations. Instead, separate timer logic has been designed external to the PX-8 CPU chips, to perform the same basic function as the HX-20 6301 timer.

Consequently, any existing HX-20 assembly code driver would be unusable for this reason alone, not to mention the obvious fact that the HX-20 runs 6301 code while the PX-8 requires Z80 code.

Since this is the first exposure to PX-8 hardware for almost everyone, the following discussion of bar code is meant to be a guideline only, and does not represent the final, authoritative word on how to write bar code software for this machine.

Bar code interface architecture

The theory of operation of the bar code interface is basically this: A 16-bit Free Running Counter (FRC), similar to the one inside the 6301 CPU, is driven by a 614 kHz clock (1.6 us period). Its output is applied to a 16-bit Input Capture Register (ICR) comprised of two 8-bit (low and high byte) addressable registers ICRL and ICRH. The current value of the FRC is latched in the ICR by any enabled bar code data edge. Hence, as the wand scans across a bar symbol, each bar's leading and trailing edge will latch a count in the ICR, that can then be read directly and interpreted by the bar code driver software.

The signal to the software that an edge has been detected, and therefore a count latched, is provided by the Input Capture Flag (ICF) bit, which is set simultaneously with the latching of the ICR. This ICF bit is sensed by polling or interrupt handling, and is immediately reset by the ICRH-B read operation that follows. (Note: ICRH-B refers to I/O port 03H, the Input Capture High Byte bar code Trigger Register and should not be confused with ICRH-C port 01H, the Input Capture Register High Byte Command Trigger.)

When the 16-bit FRC reaches 0000H, the Overflow Flag (OVF) is set to signal the end of a 64K count. The driver software also senses for this bit through polling or interrupt handling and resets the flag with a WRITE operation to port 01H (Command Register) to set bit 2 (the RES OVF bit).

After power up, the OVF and ICF bits are in an indeterminant state and must be initialized to the reset condition before reading any bar code.

Software operations

The following discussion illustrates the I/O protocol required in communicating with the bar code I/F. All the registers that are directly related to bar code input capture are accessed with I/O instructions, in contrast to the memory mapped I/O of the HX-20.

One such register is control register 1. It is a write only register that controls the power to the bar code port, enables input data edge triggering, switches memory banks, and sets baud rates. Its I/O port address is 00H.

Example:

```
  LD    A,0FH         ;Bit 3 puts bar code power on.  Bits 2 & 1 enable
         ; rising and falling edge detect.  Bit 0 enables Bank 1.
  OUT   (00H),A       ;write to port 0, Control Register 1.
```

The status register at port 5 is a read only register that has 4 bits of information (0 to 3 bits) regarding machine status. Bit 1 indicates the level of the data signal (BRDT) input from the bar code reader. The other three bits don't relate to bar code. The BRDT bit is therefore usable as a means of monitoring the particular bar type (black bar or white space) that is being read. Since most bar readers have an open-collector output, a black bar or an "off-the-paper" condition will produce a TTL high (+5v) output reading. The white margin before a bar code symbol will normally produce a low (0v) reading.

However, in the PX-8, this input logic level is inverted at the status register to produce a binary 1 state for a white input and a binary 0 for a black input. BRDT polling therefore can be used to indicate the white "quiescent" state before actual bar code scanning can be validly performed. After identification of the white margin, the edge triggering bits 1 and 2 of port 0 can be enabled, and reader scanning can proceed with automatic edge activated latching of the FRC count into the ICR.

Example:

```
   WMAR:
     LD    B,80H      ;arbitrary count down value for a white margin
   WMLOOP:            ;white margin loop
     IN    A,(05H)    ;get status
     BIT   1,A        ;test for high input indicating a white margin
     JP    Z,WMAR     ;if low, then no white input detected yet.
     DJNZ  WMLOOP     ;if on white, then decrement the B reg and loop
                      ; until 0
     JP    MAIN       ;satisfied that we're on white margin, proceed
                      ; with scanning.
```

The Interrupt Enable Register (IER) at port 4 is used to enable or disable (mask or unmask) the six major interrupt sources. Sending an all 0 output to this register will disable all interrupts. Port 4 also allows the reading of the interrupt status. The six interrupt status bits, 0 to 5, can be read even if interrupts had been masked by an earlier port 4 write. The ICF (bit 3) and OVF (bit 4) are polled here during bar code operations.

It is much easier to keep interrupts disabled during bar code ops, since software to handle the usual interrupts, along with ICF and OVF interrupts, introduces the problem of learning what the operating system is doing with the 7508 CPU clock interrupts and the serial port interrupts. (It was discovered that the 7508 and serial chip interrupts cause bank switching to occur, thereby inadvertently turning off bar code port power, since the power bit is in the same register as the bank bit. This problem can be overcome by modifying certain operatins system memory locations. But now you need operating system source listings.)

Example:

```
     LD     A,0
     OUT    (04H),A         ;disable all interrupts during bar code ops

   LOOP:                    ;edge detection polling loop
     IN     A,(04H)         ;read the interrupt status register
     BIT    4,A             ;test for OVF from the FRC
     JP     NZ,OVFSUB       ;if OVF bit set, jump to overflow subroutine
     BIT    3,A             ;test for ICF
     JP     Z,LOOP          ;if ICF received, detected bar edge, continue
   MAIN:                    ;   with main program
```

The 16-bit ICR itself is a read only type accessed by four different ports.

Ports 0 and 1 are used for reading the FRC value at any time desired by the software. The ICRL-C (low byte read command) directed to port 0 latches in the full 16-bit FRC count in ICR, and then reads the LS byte of ICR. The subsequent ICRH-C read through port 1 inputs the MS byte of ICR.

Ports 2 and 3, however, are dedicated to the reading of the FRC count after a bar code edge has previously latched the FRC count in ICR. The ICRL-B (low byte bar code read) is directed to port 2. The ICRH-B (high byte bar code read) is directed to port 3 and performs the necessary extra function of reseting the ICF.

Example:

```
  IN    A,(02H)          ;ICRL-B read
  LD    (CNT),A          ;save low byte in CNT location
  IN    A,(03H)          ;ICRH-B read.  Resets ICF also.
  LD    (CNT+1),A        ;save high byte
```

A discussion of a sample PX-8 bar code driver program in Z80 code is continued in a document on the Epson BBS named MBRCD.DOC; its companion listing, MBRCD.MAC, is also available on the BBS.

continued in a document on the Epson BBS named MBRCD.DOC; its companion listing, MBRCD.

Performing PX-8 Analog-to-digital Conversions on the PX-8

The PX-8 analog to digital conversion operations are performed under control of the 7508 slave CPU. Most, but not all, of these A/D operations can be handled with just one BIOS call that makes all the Z80/7508 communication involved transparent to the user. However, temperature sensing with the A/D converter is not handled by this call and requires some extra programming, which will be discussed later. The special A/D converter used in the PX-8 is a uPD 7001C.

The mnemonic ADCVRT is used for this BIOS call, and it provides a common entry point for reading certain specific bytes of input data, such as an external analog voltage input, the bar code reader input voltage, the internal battery voltage, the DIP switch settings, and the power switch status. Note that the last two items do not involve the A/D converter as you would expect.

A/D Specifications

The A/D converter (uPD 7001C) specifications are:

Input level:	0 to 2.0 V
Resolution:	6 bits/32 mV
Channels:	Four channels, two of which are available to the user. Two other channels are used internally to detect battery voltage and temperature.
Conversion time:	140 us
Max. input voltage:	0 - 4.5 V

ADCVRT call description

The ADCVRT subroutine call is setup as follows:

Entry Point: WBOOT + 6F

Entry Parameters:

```
  Register C = 0     ; selects A/D channel 1 (input from A/D input jack)
  Register C = 1     ; selects A/D channel 2 (input from bar code reader
                       connector)
  Register C = 2     ; selects DIP SW1
  Register C = 3     ; selects battery voltage
  Register C = 4     ; selects power switch status
```

When any other value is set in Register C, the routine returns without doing anything.

Return Parameters:

Upon return from the ADCVRT call, register A will contain the data byte requested on entry as illustrated below.

```
  Bit  -   7   6   5   4   3   2   1   0     ;Register A bits
  --------------------------------------
  Data -  MSB                LSB             ;A/D channels 1 & 2
  Data -   8   7   6   5   4   3   2   1     ;DIP SW1
  Data -  MSB                LSB             ;Battery voltage
  Data -                        TRIG PWSW    ;Power switch
```

For an A/D channel 1 or 2 request, bits 2 to 7 will all be 1 if the input voltage is greater than 2.0 V. If the input voltage is negative, bits 2 to 7 will all be 0.

For a DIP SW1 status request, each bit returned denotes the corresponding switch on/off status.

For a battery voltage request, bits 2 to 7 will all be 1 if the input voltage is greater than 5.7 V. The greater than 2.0 V battery voltage is compensated for by a special divider circuit that prevents the A/D device from seeing more than the 2.0 volts at its input. The ADCVRT routine then returns a value in the A register that can be related to the actual voltage of the battery by a linear formula.

For a power switch status request, bit 0 = 1 indicates that the Power switch is On, while bit 0 = 0 indicates that it is Off. Bit 1 = 1 indicates that the TRIG status of the analog connector is On, while bit 1 = 0 indicates that it is Off.

A practical use of the ADCVRT call is presented by the source listing for the program BATTERY.ASM written by Bob Diaz, and available on the EPSON B.B.S. Running this program on the PX-8 produces a graphic display of the actual battery voltage of your unit. As you will notice, the simplest part of the program is the actual ADCVRT call, while the rest of the program deals with binary to ASCII conversions and screen display.

7508 CPU communications

For cases where no BIOS Call is provided, the programmer should keep in mind that communication with the 7508 CPU must take place across a serial interface that requires handshaking protocol. The discussion below will illustrate this.

Since all A/D converter operations are controlled by the 7508 slave CPU, the Z80 never writes or reads directly to or from the A/D converter. The Z80 program merely sends commands to and reads data from a register in the 7508 I/F. When a one byte command is stored in this register, it is then serially passed to the 7508 CPU, which in turn recognizes and executes the specific command. The result of the 7508 command execution is then serially passed back to the same I/F register for subsequent readback into the A register of the Z80.

As described above, the ADCVRT call does all this for you for the specific cases it was designed to handle. There is one A/D feature that this call does not handle, however. This is the temperature sensing feature.

The following example of temperature sensing will further illustrate how to communicate with the 7508 chip.

I/O address 06H points to the SIOR (Serial I/O Register) in the 7508 I/F, where a one byte command to the 7508 chip is sent. For sensing temperature this command is 1CH; for battery voltage reading it is 0CH. This parallel byte is then serially transferred to the 7508 as soon as the RES RDYSIO bit (bit 1 of CMDR address 01H) is set to a 1. Setting this bit to a 1 resets the RDYSIO bit (bit 3 of STR address 05H) and signals the 7508 to read the SIOR serially. While this command is being executed by the 7508, the RDYSIO bit (Serial I/O Ready) stays low indicating that the 7508 command is still executing. When the execution is finally complete, the result is passed serially by the 7508 back to the SIOR. A Z80 input command can now read this result and process it as needed.

To obtain a temperature reading perform the following routine in Z80 code:

```
    LD   A,1CH          ;1CH is the temperature sense command
    OUT  (06H),A        ;Output 1CH to the SIOR
    LD   A,02
    OUT  (01H),A        ;Set RES RDYSIO bit to 1 to reset RDYSIO
  ; at this point the command is being sent to 7508
  RDY:
    IN   A,(05H)        ;Read RDYSIO bit.  Low=7508 busy
    BIT  3,A            ;test it
    JP   Z,RDY          ;if busy, loop until 7508 op complete
; when here, SIOR contains result of requested command
    IN   A,(06H)        ;Read the raw temperature data
```

The data byte that is now in register A must be converted into a meaningful form with a nonlinear translation formula. This formula is too complex to detail at this time. However, a very rough linear approximation, as extracted from the PX-8 technical manual, could be constructed from the five values below.

If A reg = 60H then temperature = 50 degrees Centigrade.
If A reg = 80H then temperature = 40 degrees Centigrade.

```
If A reg = A0H then temperature = 32 degrees Centigrade.
If A reg = C0H then temperature = 25 degrees Centigrade.
If A reg = E0H then temperature = 18 degrees Centigrade.
```

Making BIOS Calls from BASIC on the Geneva

Many people are aware that you can use BIOS operations to give BASIC programs some of the power and speed that was formerly limited to machine language programs. Few, however, are aware of the techniques required to do this. In the example program below, you will see how a BIOS routine (ADCVRT) can make a BASIC program more powerful and read the internal and external voltages.

Learning how to use BIOS calls can be very useful for new programmers. It allows them the ease and simplicity of writing in BASIC and then reverting to the more difficult machine code only when specialty problems of speed or power arise. The end result can be powerful programs written with ease. Some knowledge of machine code and operating systems is required.

General Procedure

You begin by writing your program in BASIC, leaving the machine portion undone. When you are satisfied that all the BASIC keyboard I/O routines, menus, etc. are in working order, you can develop the machine code portions that will give you the additional power.

Developing the Machine Language Portion

Here is the sequence of tasks the machine language portion must accomplish to handle a BIOS call and pass the results back to BASIC:

1. Discover the warm boot address.

2. Add a certain number to it to call the particular BIOS routine. Each BIOS routine you call requires a different number to be added to it to address that routine.

3. Store this updated address to memory. The stored address (which is the sum of the WBOOT address and an added offset) equals the calling address of the desired BIOS routine.

4. Call the desired BIOS routine (in the example below it reads voltage through the A/D port).

5. Return to the BASIC program where the data retrieved from the called routine is converted to meaningful numbers to be displayed on the screen.

The machine language program that does this is shown below.

Once the machine language program is developed, a couple more steps remain. The machine language op codes must be be converted to decimal codes. Finally, the converted machine language (now in decimal format) must be inserted into upper memory (usually the USER BIOS area) with POKE statements assisted by READ/DATA statements.

If you choose the USER BIOS area to write your machine code portion in, you must set aside enough room with the CONFIG program. Since the machine program is less than one page of 256 bytes in the example program, the USER BIOS is located at hex address EB00. If the routine were more than one page, the starting address would move down one page for each page of code. For example, a two page routine would have to start at hex address EA00.

WARNING: Do not attempt to measure A.C. voltages of any magnitude or D.C. voltages greater than 2 volts. You can seriously damage the Geneva if you try to do this.

Sample Program that Reads the Geneva Battery and External Voltages

```
10 REM This is a program to read internal battery voltages and
       external D.C. voltages up to 2 volts.
20 REM It is written in BASIC, but it calls machine language
       routines in the USER BIOS.
30 CLS
40 IF PEEK(&HF00B)<>1 THEN PRINT "USER BIOS SIZE IS NOT 1 BLOCK--
   USE CONFIG PROGRAM TO CORRECT THIS: SET TO 1 PAGE.":END
50 INPUT "DO YOU WANT TO READ INTERNAL BATTERY OR EXTERNAL
   VOLTAGES(B/E)";V$
60 IF V$="E" THEN GOTO 300
65 IF V$="B" THEN 90
70 PRINT "USE CAPITALS -- ENTER B OR E"
80 GOTO 50
90 REM *** SECTION TO DEFINE INITIAL VALUES ***
100 DEFINT A-J,L-Y
110 K=5.7/63             'DIVIDE 5.7 V INTO 64 PARTS FOR INTERNAL
120 Z=2/64
130 ADRS=&HEB00                          'USER BIOS AREA
140 T=0
150 REM *** SECTION TO POKE MACHINE CODE DATA TO DISK ***:
160 FOR I=&HEB00 TO &HEB17:            'THIS IS THE USER BIOS AREA
170 READ A
180 T=T+A
190 POKE I,A
200 NEXT I
210 IF T<>2362 THEN PRINT "ERROR IN DATA LINES!!!!":END
220 IF V$="E" THEN POKE &HEB0B,0
230 REM *** SECTION TO CALL ROUTINE AND CONVERT DATA ***
240 CALL ADRS
250 X=PEEK(&HEB17)      'GET THE STORED NUMBER THAT = VOLTAGE NUM
260 PRINT "NUMBER RETURNED FROM A TO D PORT=" X
270 IF V$="E" THEN PRINT X*Z "VOLTS" ELSE PRINT X*K "VOLTS"
280 END
290 REM EXTERNAL VOLTAGE SECTION
300 CLS:PRINT "NEVER EXCEED 2 VOLTS - USE D.C. VOLTAGES ONLY****"
310 PRINT "ACCURACY LIMITED TO 6 BITS (=1 PART IN 64)"
320 PRINT:PRINT "HIT ANY KEY WHEN READY TO PROCEED"
330 INPUT Z$
340 GOTO 90
350 DATA 42,1,0,125,198,111,111,34,13,235,14,3
360 DATA 205,114,231,31,31,230,63,50,23,235,201,61
```

Detailed Look at the Machine Code Portion

```
EB00 LHLD      0001      2A,01,00           42,1,0
EB03 MOV       A,L       7D                 125
EB04 ADI       6F        C6,6F              198,111
EB06 MOV       L,A       6F                 111
EB07 SHLD      EB0D      22,0D,EB           34,13,235
EB0A MVI       C,03      0E,03              14,3,
EB0C CALL      EA72      CD,72,EA           205,114,231
EB0F RAR                 1F                 31
EB10 RAR                 1F                 31
EB11 ANI       3F        E6,3F              230,63
EB13 STA       EB17      32,17,EB           50,23,235
EBI6 RET                 C9                 201
```

The Geneva will read voltages internal and external if the ADCVRT BIOS function is called. To call it, you must know the address where this routine resides. The address is equal to the warm boot address plus the hex number 6F. So the operation of reading voltages takes place in these steps.

In all CP/M machines, the warm boot address can always be located by reading the data stored in locations 1 and 2. A 16-bit read instruction will read both locations at once, and store the warm boot location in the HL register if this instruction is:

```
  LHLD 0001
```

The LHLD instruction causes the 16-bit read that reads both addresses. The 0001 gives the starting address 1. The WBOOT address has now been read from locations 1 and 2.

Next, addition cannot be done in the HL register, so the data stored there must be moved from the HL (called L below) to the accumulator (called A below). All addition is done in the accumulator. This move operation is done with the instruction:

```
  MOV A,L
```

Now that the warm boot address has been moved to the accumulator, the next step is to add hex number 6F to it with this instruction:

```
  ADI 6F
```

The first stage is now done. The number in the accumulator represents the address of the BIOS routine that calls the ADCVRT routine. We must store this number for future use. This magic number is stored back to the HL register with the command:

```
  MOV L,A
```

Finally, the number must be stored one more time in a permanent memory location so it can be called when needed. It is done in this program with the command:

```
  SHLD EB0D
```

Now that the address to call has been determined and stored at the address directly after the CALL instruction, the next step is to determine whether the voltage read will be an internal voltage (battery voltage) or an external voltage. To make this choice, a certain number must be placed in the C register just prior to calling the voltage reading routine. In the example below, the internal battery voltage is read by placing a 03 into the C register with the instruction:

```
  MVI C,
```

At this point all the preliminary work has been done and it is time to call the routine that will read the voltage. Calling this routine will leave a number in the accumulator that represents the voltage read. This number will be from 0 to 63.

If you place a 03 in the register before calling the routine, you read battery voltage. Each of the 64 numbers in the accumulator represents the battery voltages from 0 to 5.7 in 64 steps.

If you had placed a 0 in the C register before calling the routine, each of the 64 numbers you find in the accumulator represent external voltages from 0-2 V.D.C. in 64 steps.

At this stage all preliminary work has been done. All you need to do now is call the routine. This is done with the command:

```
  CALL EA72
```

NOTE: The EA72 is a dummy address; it is replaced by the data provided by the SHLD instruction above.

The Voltage has been read by calling the ADCVRT routine, but the number stored is an 8-bit value and the port is only capable of reading with 6 bits of accuracy. So only 6 of the 8 bits read have useful information. To eliminate the right two useless bits, you issue rotate right commands like this:

```
  RAR
  RAR
```

Masking is performed to get rid of any useless carry bits that may have crept in from the left. This is done with the command:

```
  ANI 3F
```

The final results are now stored at a safe location waiting for BASIC to read and convert it. This is done with the command:

```
  STA EB17
```

You have now found the location, called the routine, read the data, removed the useless portions and stored the results. The final step is to return to BASIC. This is done with the command:

```
  RET
```

By multiplying a compensation factor by the number you stored, BASIC can now read voltages with an accuracy of 1 part in 64.

Notes on Using the ASM Program

The machine program shown must be processed by the ASM program to generate the hex file that will be used. Before the ASM program can use the machine file, certain alterations must be performed. First, a starting address is needed for the ORG statement. Since you are using a one-page USER BIOS, the program is ORGed (started) at EB00. Next, the assembler requires that all hex numbers end with the letter H. An example of how this looks when completed is shown below.

```
  ORG       0EB00H
  LHLD      0001
  MOV       A,L
  ADI       6FH
  MOV       L,A
  SHLD      0EB03H
  MVI       C,03H
  CALL      0EA72H
  RAR
  RAR
  ANI       03FH
  STA       0EB17H
  RET
```

Using the HEX file

When the ASM program has processed your source machine code file, it produces two new files: one with a .HEX extension, and one with a .PRN extension. It is the .HEX one that you need to get the HEX codes necessary to convert into your BASIC program. The HEX file looks like the one shown below:

```
  :10EB00002A01007DC66F6F2203EB0E03CD72EA1F50
  :07EB10001FE63F3217EBC9BD
  :0000000000
```

There is definite order to this seeming madness. As you can see by looking at the the organized version below, the first portion is starting line numbers and addresses. This is followed by the actual data. The last part of each line is a checksum digit to insure accuracy. The file is terminated by a line of zeros instead of the usual control Z. The only part that is useful is the 16 pairs of numbers on each line (separated by parentheses in the example below). These are the number pairs that are converted to decimal and POKEd into upper memory to form the machine language part of the program.

```
:10  EB00 00   (2A 01 00 7D C6 6F 6F 22 03 EB 0E 03 CD 72 EA 1F) 50
:07  EB10 00   (1F E6 3F 32 17 EB C9) BD
:0000000000
```

The Hex pairs of numbers can be converted in BASIC by using the PRINT command with the &H prefix. For example, to convert the first two pairs on hex numbers above you would enter:

```
  PRINT &H2A
  PRINT &H01
```

The computer would return these values (in BASIC):

```
  42
  1
```

By repeating this sequence, all of the hex data pairs can be converted to decimal numbers to be poked into the upper memory or the USER BIOS area.

There is a simpler way to directly insert the hex data into upper memory. The DDT program will automatically load the file into upper memory if the file has been given a starting address with the ORG command. This is done by entering a statement like this:

```
  DDT FILE.HEX
```

□has been given a starting address with
the ORG command. This is done by entering a s

PX-8 Technical Information

We have recently been graciously supplied by Epson America with of good deal of technical information on the PX, information that supplements many areas not found in the PX-8 System Essentials Guide. The following discussion of USERBIOS originates from this documentation. If there are other areas of technical information that you would like presented here, please let us know, and we shall do our best to present them.

USERBIOS

The PX-8 Systen Essentials Manual lists amd explains most of the BIOS calls of the PX. One particular call that is not explained, however is USERBIOS. The folowing discussion illustrates the use of this call.

USERBIOS is one of the extended CP/M BIOS calls of the PX-8. It provides an entry point through which an application program can makes BIOS calls after loading its own BIOS routine in the RAM USERBIOS area. It neither requires nor returns any parameter.

The following procedure must be observed when using a user-provided BIOS routine through the entry point at USERBIOS.

1) Load the BIOS routine into the RAM USERBIOS area.

2) Replace the contents of addreses (WBOOT + 7EH) + 1 and (WBOOT +7EH) + 2 with the entry address bytes of the user routine in the USERBIOS area.

3) Call this BIOS in an application program.

Notes on Programming the USERBIOS Area

The user BIOS area mau be shared by more than one program or block of data by placing a 16 byte header at the end of the area. The header is used by thenapplication program to check whether the program or data to be used ia available in the user BIOS area.

The header is always located at EBF0H-EBFFH since the bottom address of the user BIOS area is fixed, while the top address differs depending on the size user BIOS.

1) The header ID, 2 bytes in length, and fixed to "UB".

2) The routine name, 8 bytes in length. The name of the routine loaded in the user BIOS area. Any name may be specified in ASCII, as long as it is not used in another routine.

3) The size of the routine loaded into the user BIOS area in 256-byte units stored in binary. This is 1 byte in length.

4) The overwrite flag indicates hwether the currently loaded routine can be overwritten. This is also 1 byte and if set to 00H disables the overwrite feature.

5) The release address area. The processing routine at this address is executed before a routine currently loaded in the user BIOS area is overwritten by a new routine. This release processing routine may be executed only when the overwrite flag for the currently loaded routine is set to 00H. The relaese address must fall within in the user BIOS area. The release processing routine must end with a RET instruction.

6) Not used and fixed to 00H.

7) The checksum, which is laoded with the resilt obtained by subtracting the contents of the 15 bytes (from the header top to the item preceeding checksum) from 00H, sequentially one byte at a time. This result is used for checking the validity of the header data.

Overwrite Flag and Release Processing Routine

Set the overwrite flag to 00H when loading a routine which must be resident in the user BIOS area (such as scheduler resident routines) once it is loaded. This routine can be deleted from the user BIOS area only by the program that loaded it. For routines that allow loading of new routines after execution of a release processing routine, a nonzero value must be specified to allow a new routine to be loaded into the user BIOS areawhen this area can be restored to the original state after the executon of a release processing routine. Set this flag to a nonzero value for routines which alter the system area at load time, but which can restore the system area into the original state by executing the release processing routine and loading a new one into the user BIOS area.

A user BIOS routine which is to modify the contents of the system area (hook or jump table, for example) must save the original contents of the ssytem area into the user BIOS area before starting execution. The release processing routine is called to restore the system into the state before the user BIOS routine is loaded by placing the saved contents back into the system area and setting all header fields to 00H. The header must be cleared even if the system area need not be restored to the original state. The release processing routine must be placed in the highest 256 bytes (including the header) of the user BIOS area.

Using User BIOS with an Application Program

The application program must verify that the user BIOS routine is available before accessing the routine. The procedure illustrated below must be followed to check this.

```
***********************
FIGURE TO BE TYPSET
***********************
```

(A) Checks whether the correct header is present by matching the header ID with "UB" and checksum. If the header ID field contains "US", it is unconditionaly concluded that the scheduler is using the user BIOS area because it defines the header as "US".

(B) Check to determine whether the required user BIOS routine is loaded into the ser BIOS area by checking the routine name in the header.

(C) Call the routine addressed by the release address in the header.

(D) Load a new routine to the user BIOS area and update the header contents.

Compiling PX-8 BASIC Programs

There is no specific compiler designed for the version of BASIC implemented on the PX-8. MicroSoft's BASCOM will work to a certain degree with PX-8 BASIC, however. It is available in 3 1/2 disk format for use on the PF-10 from SofTeam (800-438-7638). David Western has taken the time to outline the specific areas of incompatiility between BASCOM and the version of BASIC on the PX, with certain suggestions on how to overcome some of these descrepencies. Here are his comments.

Here is a list of many important commands that I have found to be incompatable in some way and a corresponding solution wherever possible.

by David Western

All commands commonly entered on the command line such as LIST, AUTO, LOAD, and SAVE can not be used. Other files can be chained, however, using the PX-8 compatable CHAIN command.

ALARM$ None of the alarm functions work in BASCOM. I have found no way other than a machine language routine to fix this problem.

BEEP The only thing you can do to make much noise in BASCOM is to use PRINT CHR$(7) to activate the bell.

CLS Use PRINT CHR$(12) instead of CLS.

FRE(0) This will return 0 according to the BASCOM manual.

INSTR The INSTR function is not supported. Here is a short routine to do the same thing. A$ contains the string you are searching and E$ contains the string you are finding in A$. Z returns the position where found or 0 if not found.

```
1000 FOR I=1 TO LEN(A$)
1010 IF MID$(E$,I,LEN(E$))=MID$(A$,I,LEN(E$)) THEN
     X=I:RETURN
1020 NEXT I
1030 X=0:RETURN
```

LINE LINE is not supported. Here is an ESC sequence that will do the same thing once in SCREEN 3. Use the way listed here to get into that screen mode.

```
PRINT CHR$(27);CHR$(198);CHR$(X1);CHR$(X2);CHR$(Y1);
      CHR$(Y2);CHR$(X3);CHR$(X4);CHR$(Y3);CHR$(Y4);
      CHR$(Z1);CHR$(Z2);CHR$(Z3);
```

```
X1 = 1st byte of 1st X coordinate
X2 = 2nd byte of 1st X coordinate
Y1 = 1st byte of 1st Y coordinate
Y2 = 2nd byte of 1st Y coordinate
X3 = 1st byte of 2nd X coordinate
X4 = 2nd byte of 2nd X coordinate
Y3 = 1st byte of 2nd Y coordinate
Y4 = 2nd byte of 2nd Y coordinate
Z1 = 1st byte of mask pattern
Z2 = 2nd byte of mask pattern
Z3 = 1 for OFF, 2 for ON, 3 for COMPLIMENT
```

.pa

LINE (400,18)-(18,18) is

```
PRINT CHR$(27);CHR$(198);CHR$(1);CHR$(144);
CHR$(0);CHR$(18);CHR$(0);CHR$(18);CHR$(0);CHR$(18);
CHR$(&HFF);CHR$(&FF);CHR$(2);
```

LOCATE Here is an ESC sequence to use as LOCATE does not work with BASCOM.

```
PRINT CHR$(27);"=";CHR$(Y+31);CHR$(X+31);
```
X and Y stand for the X and Y position.

OPEN Only files that access the disk drives can be used. All other specs are considered part of the filename. IN and OUT must be used along with a call to the open routine to directly use the PX-8 RS232 port. LPRINT works fine, though.

OPTION COUNTRY Option Country is not supported in BASCOM. In order to use this feature you must use the procedure described on pages C-3 and C-4 of the PX-8 BASIC manual.

OPTION Option Currency does not work and I can find no control

CURRENCY	sequence to remedy this.
PCOPY	PCOPY is not used as there are not seperate BASIC areas in the BASIC compiler at this time.
POINT	The POINT command, which returns the status of a dot on the screen, is not implemented. It would be possible to peek the graphics memory to get this but I don't know where that is or how it is formatted. I wish it DID work because then I could compile my OKIDATA screen dump routine which takes forever in BASIC.
POWER	All of the POWER commands (POWER OFF, CONT, or RESUME) do not work. I know of no remedy for this either.
PRESET	PRESET can be done by using control sequences. The same basic sequence is used by both PSET and PRESET. A full description is on page C-8 and C-9 of the PX-8 BASIC manual under the section for ESC CHR$(199).
SCREEN	The SCREEN command along with all of its options are supported by control codes. All of them are on page C-9 of the BASIC manual under the section describing ESC CHR$(208). All Features are supported using these.
SOUND	I know of no way to use SOUND in BASCOM. I imagine there are routines you can call but EPSON doesn't seem to think anyone would be interested.
STAT	STAT is not handled in any way, but there is no real need for it.
STOP KEY	I can find no alternative for the STOP KEY command, but I am pretty sure there is one. If anyone knows of an MBASIC command or routine to do this please notify someone about it.
TAPCNT	TAPCNT can not be used though I am sure there is some location that can be peeked for the value. I'm not an expert on the special memory locations for the PX-8 as there is no information for it that I can get a hold of.
TIME$	No time routines are present. This was because all TIME routines are different and Microsoft wanted to make sure BASCOM worked on most CP/M machines. There is a machine language routine for it floating around somewhere...
TITLE	TITLE is not used as multiple program areas are not used.
WIND	No command to control the micro-cassette's motor could be found though it is likely that one exists.

That is the end of the list. I hope that it has proved helpful to PX-8 Geneva users. If anyone has anything to add to the list then login to the SOCIS Epson Connection BBS and leave mail to David Western or Bob Hermann. Otherwise call The Board Room at (803) 548-1243 and leave mail to the SYSOP or upload the file.