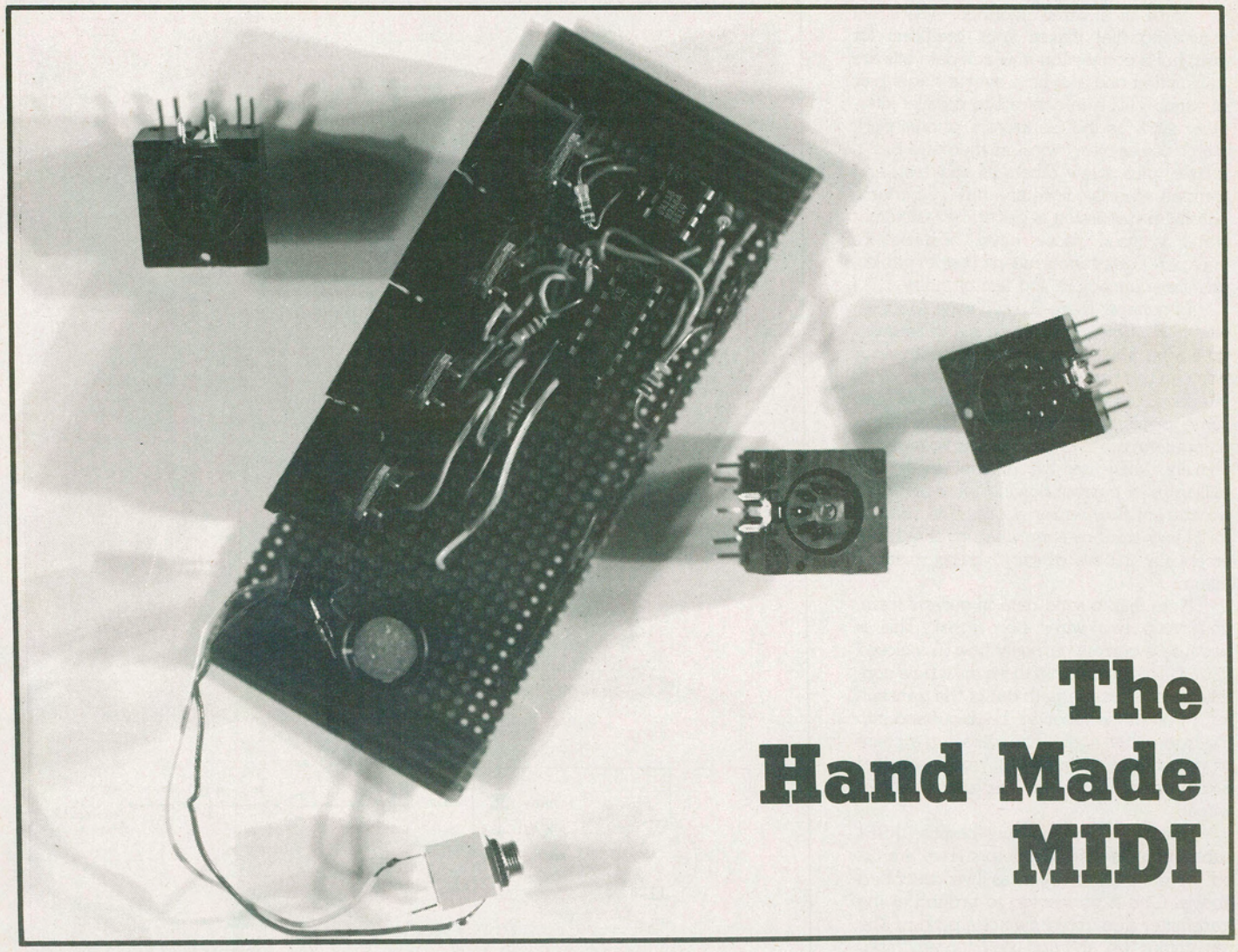


The Hand Made MIDI

One does not normally think of older CP/M based systems as being capable of supporting MIDI. However, if you want to get right into the grotty details you can get them singing. Here are the details, for hackers only.

by Shane Dunne

This is not a full fledged construction project article, but rather a hacker's eye view of the hardware and software involved in coaxing your favourite computer to speak MIDI. However, I have included enough information here for someone with some hardware experience to build their own MIDI interface using as few as three chips.

If the smell of solder makes you feel faint, if you don't feel comfortable modifying existing circuit designs, or if you just don't want to hack furiously into the innards of your prized computer, then I would advise you to purchase a commercial MIDI interface rather than trying to build one.

About This MIDI Business

MIDI is two things. It's a hardware interface specification and a method of encoding data for that interface. These two aspects are quite distinct, and may be separated. For example, the Wersi company makes a series of digital organs which communicate using MIDI data, but over an RS-232 interface... that's the kind used with most computer terminals. A group of artists in Montreal are now building a computerized multimedia studio in which lights, video and laser projectors will be connected to a computer by MIDI interfaces. Naturally these will not communicate using the regular MIDI music codes. New codes will be designed for their special needs.

Let's first look at the hardware specification. This is the part of MIDI that is most clearly defined by the available specification documents.

MIDI uses a five milliamp current loop to carry asynchronous serial data at 31.25 kilobaud. Let's look at that one step at a time.

The Hand Made MIDI

MIDI is a serial interface. That's not something that makes your breakfast for you, but a connection in which data bits are transmitted one at a time over a single pair of wires. This is in contrast to parallel interfaces such as the Centronics printer port, which use as many wires as there are bits in a byte, plus a few others to add zest, and transmit an entire byte at a time. Of course parallel transmission is generally faster, but serial is used more often because it's cheaper. Fewer wires mean cheaper cables, cheaper connectors and less circuitry.

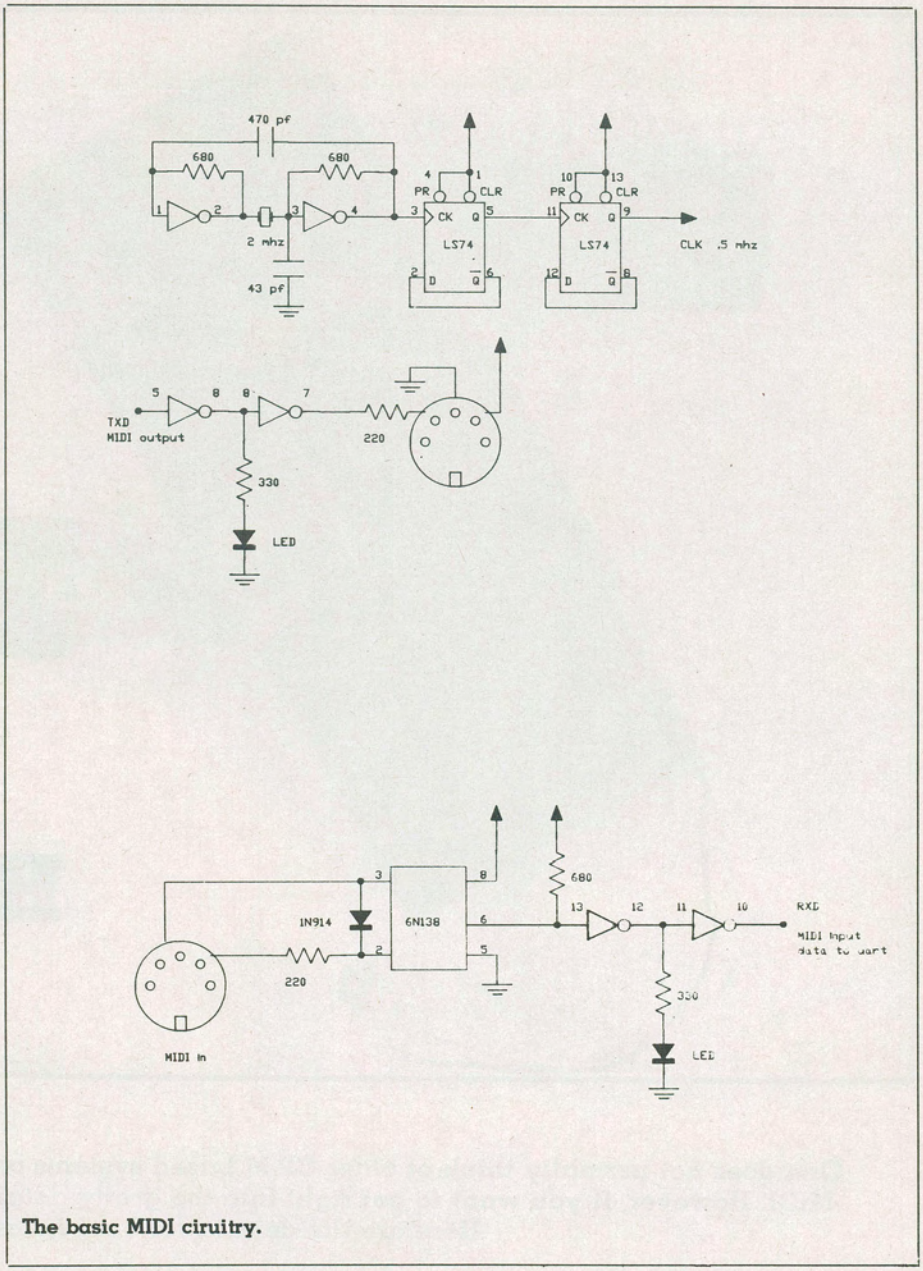
In general, there are two ways to represent bits... I mean binary ones and zeroes... on a wire. You can use two voltage levels or you can use two current levels. The RS-232 interface, used with most terminals and some printers, works with voltages. Most computers also represent bits as voltages internally. MIDI uses two current levels, five milliamps of current flowing for a zero and no current flowing for a one. The reasons for doing this are somewhat involved, but essentially it boils down to being cheaper again.

Note that to send data in current form, you need two wires per signal. This is because a current can only flow in a closed circuit. This means that there must be a continuous conducting path out of the transmitter, through the receiver, back out and into the transmitter again. Draw that on a piece of paper and, hey presto, you'll see that each current loop cable has to contain two wires.

Here's where the fun begins. MIDI cables contain five conductors. Two are used for data transmission as I've described above. One is connected to ground at the transmitter side to reduce noise pickup. The remaining two are not used for anything. The reasons for these extra wires aren't as interesting as one would think they might be. The MIDI designers wanted to use cables that were already available, and which had keyed connectors... the kind you can't force in backwards. The five pin DIN cables normally used to connect tape decks to stereos were a good choice. They're available around the world, they're keyed, and they're cheap, costing about five bucks each at Radio Shack.

There are two ways to transmit serial data, these being synchronously and asynchronously. Synchronous transmission requires a separate clock signal along side the data, and in general is complicated and expensive. I think you've already guessed that the MIDI designers chose the cheap, simple asynchronous method.

Asynchronous transmission avoids the need for a clock by adding a start bit and a stop bit to each character. The start bit is always a zero, and an idle line is always held in the one state, so the transition from one to zero triggers the receiving hardware that a byte is coming in. From there on, the receiver knows approximately when each bit of the character will come in, given that bits are sent at a fixed rate which is known in advance.



The basic MIDI ciruitry.

MIDI moves data around at the rate of 31250 bits per second. A baud is one signal change per second, and for serial interfaces in which each signal change represents one bit, the bit rate equals the baud rate. MIDI falls into this category. Having to know about this sort of thing helps communications engineers command fat salaries, and keeps the rest of us hopelessly confused.

Most MIDI devices have three terminals, marked MIDI in, MIDI out, and MIDI through. MIDI in is for data coming into the device, MIDI out is for data being output by the device, and MIDI though outputs a copy of the data coming in at MIDI in.

UART A Fool, Horatio

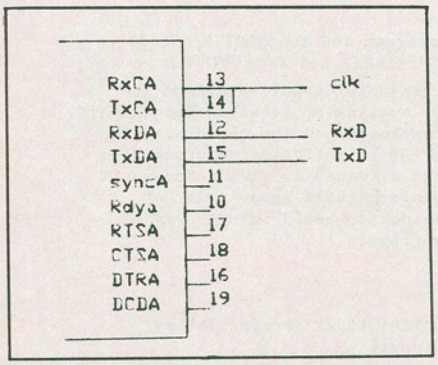
A MIDI interface for a computer consists of two main sub-systems. The first is a serial communications chip, which converts serial data to parallel and vice versa, handling gremlins like start and stop bits on the way. The second is the analogue circuitry used to represent serial bits in current loop form.

To get information into and out of a microprocessor system, it is necessary to use I/O chips. These look to the microprocessor like registers... places where data can be read or written. For asynchronous serial I/O a chip called a Universal Asynchronous Receiver/Transmitter, or UART, is used. When the processor writes a byte to the output register of a UART, the chip clocks the bits of this byte out onto the serial port. When a serial byte is

received from the port, it is placed into the UART's input register to be read by the microporcessor when it is ready.

MIDI uses a very common format for serial communication, this being eight bits per character, one start bit, one stop bit, and no parity with an idle line held at a logic one level. This is exactly the same arrangement as is used by RS-232 terminal lines and printer interfaces, so the same UART chips can usually be used for both. The only difficulty is finding a UART that can work at thirty-one kilobaud. The Zilog Z80-SIO will, and so will the Motorola 6850, but the venerable Intel 8251 will not. However Intel does make a faster chip called the 8256 which will handle up to a million bits per second.

You may be wondering where the strange value 31250 comes from. Well,



Connecting a Z80-SIO to the MIDI circuitry.

most UARTs need a clock input whose frequency is sixteen times the data rate. Sixteen times 31250 is half a megahertz, which is a frequency that should be easy to derive in any microprocessor circuit.

I have a "big board" Z80 computer from Digital Research Computers, which uses a Z80-SIO chip for serial I/O. By supplying my own clock signal to this chip, I was able to use it for the MIDI interface, and hence only had to build the analogue part. If your computer uses a serial chip capable of operating at the right speed, and if you can manage to supply it with the right clock frequency, you can probably do the same thing.

If you can't use your computer's existing UART chip, you will have to add one to the system somehow. This is where I warn all but the most seasoned hardware hackers to stick to commercial hardware, as adding a new peripheral chip to a computer is no mean trick.

The schematic in this article shows the circuit for my MIDI interface, which consists of three parts. The clock generator uses a two megahertz crystal and generates the half megahertz signal used to clock both the receiver and transmitter portions of the UART. The output section buffers the outgoing data and converts it to current loop

Listing 1: Example of MIDI I/O from Turbo Pascal on a Z80 systems.

```
{first declare the port addresses for your UART.  The following ones are
 those for my system.}
CONST   cport = 6;        {UART control port address}
        dport = 4;        {UART data port address}

{now declare the data structures for MIDI i/o}
VAR     mqueue: ARRAY[0..255] OF BYTE;  {the MIDI input queue space}
        icur,ocur: INTEGER;             {input and output cursors for queue}
        OKvec: INTEGER ABSOLUTE $FF0C;  {interrupt vector for normal UART
                                          input interrupt}
        ERvec: INTEGER ABSOLUTE $FF0E;  {interrupt vector for abnormal UART
                                          input interrupt (ie framing error)}
        ErrStat: BYTE;  {if a UART error occurs, the interrupt handling
                         procedure RxErr puts the UART status in here}
        done: BOOLEAN;  {set           ue when user types a character while
                                        waiting for MIDI input}

{of course you'll need a lot of other declarations for a complete
 program.  I'm only showing what's required for MIDI i/o}

{here is a procedure to handle abnormal interrupts from the UART.  We
 use Turbo's INLINE statement to enter machine code directly in the
 Pascal source file here.  Unfortunately you have to hand-assemble the
 code to get it in hex form, which is all INLINE will accept.  However
 you can look on the bright side and figure that this will make you
 use machine code only where it's really needed.  (Or you can just shout
 and scream.  That's what I did.)
 Of course this code is highly machine-dependent.  Here I'm showing you
 what I use in my system, which has a Z80-SIO for serial i/o}
PROCEDURE RxErr;
BEGIN
  INLINE (
    $F5/                {PUSH   AF              ; free up the A register}
    $3E/1/              {LD     A,1             ; prepare to read from  }
    $D3/<cport/         {OUT    (cport),A       ;  SIO read-register #1 }
    $DB/<cport/         {IN     A,(cport)       ; get RR1 (SIO status)  }
    $32/ErrStat/        {LD     (ErrStat),A     ; save for main program }
    $3E/$30/            {LD     A,30H           ; send "error reset"    }
    $D3/<cport/         {OUT    (cport),A       ;  to the SIO           }
    $F1/                {POP    AF              ; restore AF            }
    $FB/                {EI                     ; re-enable interrupts  }
    $ED/$4D             {RETI                   ; return from interrupt:
                                                ;  resets Z80 i/o chips }
  )
END {RxErr};

{here is the handler for normal input interrupts from the UART.  Again
 this is the code which I use with my Z80-SIO, but it should actually
 be quite similar for any system, except for the port addresses.  Read
 this code carefully to see how it implements a circular queue buffer}
PROCEDURE Stash;
BEGIN
  INLINE (
    $F5/                {PUSH   AF              ; free up               }
    $E5/                {PUSH   HL              ;  some                 }
    $D5/                {PUSH   DE              ;  registers            }
    $21/mqueue/         {LD     HL,mqueue       ; HL will be a pointer
                                                ;  into the queue       }
    $ED/$5B/icur/       {LD     DE,(icur)       ; get icur value        }
    $1C/                {INC    E               ; increment modulo 256  }
    $ED/$53/icur/       {LD     (icur),DE       ; store back in icur    }
    $19/                {ADD    HL,DE           ; use this as an offset
                                                ;  to index into mqueue }
    $DB/<dport/         {IN     A,(dport)       ; get MIDI in byte to A }
    $77/                {LD     (HL),A          ; store it in the queue }
    $D1/                {POP    DE              ; restore               }
    $E1/                {POP    HL              ;  the                  }
    $F1/                {POP    AF              ;  registers            }
    $FB/                {EI                     ; re-enable interrupts  }
    $ED/$4D             {RETI                   ; return from interrupt:
                                                ;  resets Z80 i/o chips }
  )
END {Stash};

{You will also need some code to "install" these interrupt handlers, and
 generally prepare the system for MIDI i/o.  This means setting up the
 UART in the correct mode, and making sure that the interrupt vectors
 for UART interrupts point to the routines RxErr and Stash, as appropriate.
```

form. The input section uses an opto isolator to transform incoming current loop data to TTL compatible form for input to the receive portion of the UART. Both the input and output sections also contain LEDs which light whenever a zero bit is detected, thereby allowing you to observe activity on the MIDI line. The reason it lights for zero, not one, is that idle MIDI lines are held in the one state. Thus the light stays off when the line is idle, and flickers when data comes along.

I've also indicated how these circuits would be connected to a Z80-SIO and a Motorola 6850. These are not complete circuits, but are intended to show how the analogue and UART sections fit together. The exact connections will depend on your specific system.

Note that the half megahertz clock signal, identified as CLK in these figures, is a fairly high frequency beast. Therefore it's best to keep the clock wire as short as possible to avoid generating radio frequency interference. Ideally, keep the clock circuit in the same box as the UART, or run a shielded cable if this is not possible.

Programming With MIDI

Once you have a MIDI interface, home-built or commercial, you will need to do some programming to make it do things. MIDI programming is a rich subject, one which is beyond the scope of this article to do justice to. However, I'll point out a few things specific to this project here.

The trickiest thing about MIDI programming is speed. The 31250 bits per second means one byte every three hundred and twenty microseconds. This means that your program has to be able to respond mighty fast to avoid losing input bytes. If you are only interested in MIDI output however, that is, to use a synthesizer to play pre-programmed tunes, blazing speed is not strictly necessary.

There are some commercial MIDI interfaces, notably the Roland MPU-401, which contain their own microprocessors and handle all the nasty time critical details of MIDI communications. This is cool, but with a little careful programming you can get by quite nicely without having to buy one of these rather expensive gadgets. The following discussion assumes that you will not be using such an intelligent interface.

If you want your programs to receive MIDI input, you will almost certainly not be able to write them all in interpreted BASIC, because interpreted programs are so slow. Assembler is fine, because you can predict how many instructions will be executed between input bytes. Compiled languages like Pascal, Fortran, and the like may be suitable, but can bring on nightmares because you cannot generally predict execution speed in this way.

The best way to handle MIDI input is to

```
{This code is so system-dependent that it's not worth showing you mine.
 Instead I'll just outline the basic structure and highlights}
PROCEDURE InitMIDI;
BEGIN
  icur := 0; ocur := 0;          {zero both queue cursors}
  {      .
         .
         .
   here you set up your UART for the correct mode.  Turbo Pascal's
   "PORT" input and output facilities are very useful for this.
         .
         .
         .}
  OKvec := Addr(Stash);          {set the main UART input interrupt vector
                                  to point to the Stash routine}
  ERvec := Addr(RxErr);          {set the UART error interrupt vector to
                                  point to the RxErr routine}
  {      .
         .
         .
   here you will probably need to send a last command or two to your
   UART to enable its interrupts.
         .
         .
         .}
END {InitMIDI};

{The interface between your Pascal main program and the MIDI i/o code
 consists of just two routines: GetMIDI for input and SendMIDI for output}
{GetMIDI is a function which returns the next MIDI input byte from the
 MIDI input stream.  If there isn't a byte available, it waits for either
 a MIDI byte to arrive, or for a key to be pressed on the console.  If
 a key is pressed, it returns immediately (note: the value returned will
 be garbage!), having set the global variable "done" to TRUE.  You don't
 have to do it this way, but since many programs will spend a lot of
 time waiting for MIDI input, it's a good idea to build in some way of
 interrupting this wait, i.e. to stop the program.}
FUNCTION GetMIDI: INTEGER;
BEGIN
  WHILE (icur = ocur) AND (NOT done) DO
    done := KeyPressed;          {wait for MIDI input or console key}.
  IF NOT done                    {if MIDI input}
  THEN BEGIN
    IF ocur = 255
    THEN ocur := 0
    ELSE ocur := ocur + 1;       {increment output cursor mod 256}
    GetMIDI := mqueue[ocur]      {get next MIDI byte from queue}
END {GetMIDI};

{SendMIDI is a procedure which outputs its integer argument as a MIDI
 byte to the MIDI output stream.  This version does not make use of
 any lower level machine-language support routines, but just waits for
 the UART transmit section to become ready (if it is not already) and
 then sends the byte to the UART directly.  If you want to be able to
 do a lot of processing during MIDI i/o, you may want to have MIDI
 output also interrupt-driven.  In this case, SendMIDI would put the
 next MIDI byte into a queue, and a machine-language interrupt handler
 (activated when the UART transmit buffer becomes empty) would take
 bytes out of the queue and pass them to the UART.  If you want to do
 this, make sure your interrupt handler doesn t do anything wierd when
 there isn't any pending MIDI data.}
PROCEDURE SendMIDI(dat: INTEGER);
VAR status: INTEGER;
BEGIN
  REPEAT
    status := port[cport]                {keep getting UART status}
  UNTIL (status AND 4) <> 0;             {until bit 2 set, meaning output
                                          buffer is empty}
  port[dport] := dat                     {pass MIDI data to output buffer}
END {SendMIDI};
```

```
5 'an example of MIDI access in BASIC
10 DIM RXERR%(8)        'reserve 18 bytes for RxErr routine
20 DIM STASH%(12)       'reserve 26 bytes for Stash routine
30 DIM MQUEUE%(127)     'reserve 256 bytes for the queue buffer
40 ' make sure all your simple variables are initialized before
50 ' executing this code.
100 'load the RxErr routine
```

set up your UART to generate an interrupt every time it receives a character, and then write a very fast interrupt handler in assembler. This interrupt handler just takes the byte from the UART and appends it to a queue which is accessible from the main program. The main program itself can then be written in any language... even BASIC.

A queue is a data structure which allows data to be buffered in a first-in first-out discipline. That is, the first byte to be placed in the queue by the interrupt handler will be the first byte retrieved from the queue by the main program. If bytes come in faster than the main program can process them, they are piled up in the queue, just like people waiting in a line for a bank teller... and will be processed in proper sequence when the main program eventually does retrieve them.

Listing one shows how to write a queue-oriented interrupt handler in Z80 machine language and use it with a Turbo Pascal main program. Listing two shows how to do the same thing with a Microsoft BASIC program. These are not complete programs, but just outline the essential code needed for MIDI communications. The actual implementation will vary with different computers. This code is adapted from the code I have used with my Z80 big board system. Unless your machine is identical, you should treat these programs only as a guide.

I have shown two different interrupt handling routines, called Stash and RxErr. This is because many UARTs will generate one kind of interrupt when a character is received normally, and another kind when some kind of error is detected. The most common is a framing error, which happens when the UART receives a start bit, then clocks in eight data bits, then looks for the stop bit... which is always a one... and finds the line in the zero state instead. If you notice a lot of framing errors, it could be because your UART isn't being clocked correctly.

The approach to storing and using machine code within BASIC, shown in listing two, is a kluge. The Microsoft BASIC interpreter provides various methods of using machine language routines, but all of them are messy. If you are using Microsoft's BASIC Compiler you can write all the machine language stuff in assembly code, use M80 to assemble it, and then link it into your compiled BASIC program using L80.

Parting Words

MIDI has its complications, but need not be as mysterious as it sometimes appears. It is based on the very common technology of asynchronous serial transmission, just like the RS-232 interface used with terminals and printers.

To those who want to brave the slings and arrows and build MIDI interfaces from scratch, I wish the best of luck.

```
110 FOR ADR% = VARPTR(RXERR%(0)) TO (VARPTR(RXERR%(0)) + 17)
120  READ DAT%
130  POKE ADR%,DAT%
140 NEXT ADR%
150 DATA &HF5           'PUSH   AF
160 DATA &H3E,1         'LD     A,1
170 DATA &HD3,6         'OUT    (cport),A       ;cport=6 is hard-coded here
180 DATA &HDB,6         'IN     A,(cport)
190 DATA &H32,0,0       'LD     (ERRSTAT%),A    ;see lines 260,270 below
200 DATA &H3E,&H30      'LD     A,30H
210 DATA &HD3,6         'OUT    (cport),A
220 DATA &HF1           'POP    AF
230 DATA &HFB           'EI
240 DATA &HED,&H4D      'RETI
250 'fill in address of ERRSTAT% which can't be put in a DATA statement
260 POKE VARPTR(RXERR%(0)) + 8, (VARPTR(ERRSTAT%) MOD 256) 'low byte
270 POKE VARPTR(RXERR%(0)) + 9, (VARPTR(ERRSTAT%) \ 256)   'high byte
300 'now load the code for the Stash routine
310 FOR ADR% = VARPTR(STASH%(0)) TO (VARPTR(STASH%(0)) + 25)
320  READ DAT%
330  POKE ADR%,DAT%
340 NEXT ADR%
350 DATA &HF5           'PUSH   AF
360 DATA &HE5           'PUSH   HL
370 DATA &HD5           'PUSH   DE
380 DATA &H21,0,0       'LD     HL,MQUEUE%      ; see lines 510,520 below
390 DATA &HED,&H5B,0,0  'LD     DE,(ICUR%)      ; see lines 530,540 below
400 DATA &H1C           'INC    E
410 DATA &HED,&H53,0,0  'LD     (ICUR%),DE      ; see lines 550,560 below
420 DATA &H19           'ADD    HL,DE
430 DATA &HDB,4         'IN     A,(dport)       ; dport=4 is hard-coded here
440 DATA &H77           'LD     (HL),A
450 DATA &HD1           'POP    DE
460 DATA &HE1           'POP    HL
470 DATA &HF1           'POP    AF
480 DATA &HFB           'EI
490 DATA &HED,&H4D      'RETI
500 'fill in address values not expressible in DATA statements
510 POKE VARPTR(STASH%(0)) + 4, (VARPTR(MQUEUE%(0)) MOD 256)  'low byte
520 POKE VARPTR(STASH%(0)) + 5, (VARPTR(MQUEUE%(0)) \ 256)    'high byte
530 POKE VARPTR(STASH%(0)) + 8, (VARPTR(ICUR%) MOD 256)
540 POKE VARPTR(STASH%(0)) + 9, (VARPTR(ICUR% \ 256)
550 POKE VARPTR(STASH%(0)) + 13, (VARPTR(ICUR%) MOD 256)
560 POKE VARPTR(STASH%(0)) + 14, (VARPTR(ICUR%) \ 256)
570 ' we need some code to set up the hardware for MIDI i/o, in
580 ' particular to set up the UART for interrupt-based operation.  As
590 ' before I can only give an outline of this initialization code.
600 'initialize MIDI i/o
610 ICUR% = 0 : OCUR% = 0
620 ' Put code to set up UART for interrupt operation here.  BASIC's
630 ' IN and OUT instructions will be useful here.
750 'write pointers to Stash and RxErr routines in interrupt vectors
755 '    Note: the addresses FF0C, FF0D, etc. are the ones in my
756 '    system.  Yours will probably be different.
760 POKE &HFF0C,(VARPTR(STASH%(0)) MOD 256)     'lo byte of Stash
770 POKE &HFF0D,(VARPTR(STASH%(0)) \ 256)      'hi byte of Stash
780 POKE &HFF0E,(VARPTR(RXERR%(0)) MOD 256)    'lo byte of RxErr
790 POKE &HFF0F,(VARPTR(RXERR%(0)) \ 256)      'hi byte of RxErr
800 ' Enable UART interrupts, generally get set to go...
810 'Here are the two interface subroutines. BASIC doesn't have
820 'decent parameter passing or return mechanisms, so I'm assuming
830 'the existence of variables MIDI.IN% (where the input routine
840 'puts the next MIDI input byte) and MIDI.OUT% (where the output
850 'routine looks for the next MIDI output byte).
1000 'MIDI input subroutine (GetMIDI)
1010 IN$ = INKEY$       'get console keyboard status
1020 IF (IN$ = "") AND (ICUR% = OCUR%) THEN 510
1030 IF (IN$ <> "") THEN 5000   'go to "keyboard interrupt" code
1040 IF (OCUR% = 255) THEN OCUR% = 0 ELSE OCUR% = OCUR% + 1
1050 MIDI.IN% = PEEK(VARPTR(MQUEUE%(0)) + OCUR%)
1060 RETURN
2000 'MIDI output subroutine (SendMIDI)
2001 '   note: the constant port addresses and AND mask are the ones
2002 '         I use in my system.  Yours will probably be different.
2010 WAIT 6,4   'a concise way to express a wait loop in BASIC
2020 OUT 4,MIDI.OUT%
2030 RETURN
```

CN!