The COMPUTER JOURNAL

Programming - User Support

Applications

Issue Number 46 September / October 1990 $3.95

Build a Long Distance Printer Driver

Embedded Systems for the Tenderfoot

Foundational Modules in Modula 2

The Z-System Corner

Animation with Turbo C

Z80 Communications Gateway

Real Computing

The Computer Corner

ISSN # 0748-9331

The Computer Journal

Editor/Publisher
Art Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Tim McDonough
Bridger Mitchell
Clem Pepper
Richard Rodman
Jay Sage
Dave Wenstein

The Computer Journal is published six times a year by Technology Resources, 190 Sullivan Crossroad, Columbia Falls, MT 59912
(406) 257-9119

Subscription rates—$18 one year (6 issues), or $32 two years (12 issues) in the U.S., $24 one year surface in other countries. Inquire for air rates. All funds must be in U.S. dollars on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912, phone (406) 257-9119.

The COMPUTER JOURNAL

Issue Number 46 September / October 1990

Editor's Page

The Regulators are Coming!!

The bureaucrats usual response to a problem is to pass laws and regulations to control and regulate the physical objects involved. Their misguided efforts on gun control haven't decreased armed felonies (and won't), but that hasn't stopped them from trying the same approach with computers.

Modern electronics has greatly simplified the counterfeiting of U.S. currency, and The Secret Service and the Bureau of Engraving and Printing have endorsed a bill to toughen laws against counterfeiting. Carl V. D'Alessandro, assistant director of the bureau, told the committee that currency and other securities can be counterfeited with remarkable accuracy using widely available computers, printers, and copiers.

"The potential counterfeiter no longer requires any particular expertise in printing, but only the inclination and the electronic devices," he said.

The measure D'Allesandro and William J. Ebert (head of the Secret Service's counterfeit division), endorsed would make it a crime to possess any laser-type devices the Treasury Department concludes would help a counterfeiter.

Kenneth A. Wasch, executive director of the Software Publishers Association, said that the bill is so broad that "many heretofore legal activities may now be made illegal."

He said the prohibition on private possession of certain optical devices could "mean document scanners, desktop computers, a number of software programs, laser printers and other high-tech wonders could be outlawed even though they are in use everywhere."

We used to laugh about the fact that Russia tightly controlled the availability of office copy machines. Only authorized agencies were allowed to possess them, every copy had to described and accounted for, and government security people checked the locked counters against the logged use.

Apparently bureaucrats everywhere have the same (limited) mentality. At the very time when Russia is loosening up on their rigid controls, our leaders are trying to impose oppressive controls here.

While I doubt that such restrictive legislation will be passed at this time, I fear that some form of restrictions will be enabled--and that they will be further expanded a little at a time. The point which greatly concerns me is that our bureaucrats don't have any better solution than to limit the access of law-abiding citizens to the high-tech equipment we need. For example, I had heard the rumor that the Treasury Department was trying to prevent the sale of color copiers.

I encourage you to watch these developments very closely, and to let your representatives in the Congress and the Senate know your view on the matter. Also remember that once the Secret Service or the Treasury Department is given the power to "regulate" they can write and enforce their own regulations without specific congressional approval. I would appreciate any further information on developments on this subject.

Z Festival 1990

Lee Bradley, editor of *Pieces Of 8* (The Bimonthly Newsletter of the Connecticut CP/M Users' Group) advises that Z Festival is tentatively scheduled for October 20, with a probable location of the RPI campus in Hartford, CT. One disadvantage of living out here in the mountains is that I can't attend these meetings which are so helpful. You should attend if at all possible. Contact Lee at 24 East Cedar Street, Newington, CT 06111 for the latest details.

Embedded Applications

I have been talking to a lot of people about embedded applications, and find that there is more activity than I anticipated. It is ironical that embedded controllers was the intended application when microprocessors were invented. Microcomputers didn't even exist until after the microprocessor was readily available; but after the hardware hackers showed what microcomputers could do, that's where the action was. Embedded applications didn't die, the controllers are used in machine tools, printing presses, microwave ovens, VCRs, automated cameras—in fact most of our modern electronic marvels. Two of the fastest growing embedded applications are communications and automotive.

As the price of processors and TTL devices came down, there was a flurry of activity in hobbyist construction projects to fill the needs for the new items which could now be built. As people became more involved with the microcomputers (which took up all of their available time), activity in the hardware construction projects dwindled. Industrial, commercial, and military activity accelerated, but there were few individuals working on hardware projects for their own use. Now, when microcomputers are powerful tools to work **with**, but no longer any fun to work **on**, people are again becoming interested in hardware projects.

I have long been interested in using processors to control mechanical devices, but I was always stopped because I didn't know how to accomplish the hardware portions. Some of the devices will be battery operated portable using CMOS chips, while others will be machine tools which will be interfaced to either desktop computers or stand-alone embedded controllers. I finally decided that the years are going by too fast, and that I had better establish my goals and set my priorities.

Since the first of the year, I have acquired a two channel scope, 8051 crossassembler and simulator (PseudoCorp), 8031 prototyping board (Cottage Resources), EPROM programmer

(Continued on page 38)

Build a Long Distance Printer Driver

by Stuart R. Ball

I needed to move a printer into a closet so that my wife could use the computer without waking the children. Unfortunately, the closet is a fairly long distance from the computer, and the normal output from a computer cannot reliably drive a cable that far. My solution was the circuit described herein, which allows me to move my printer hundreds of feet from the computer.

Why not just run a longer cable from your computer to your printer? Here's why: The printer output from an IBM PC, AT, or clone is a parallel interface. The computer sends 8 bits of data and several control signals to the printer. The printer returns several status bits (such as paper empty) back to the computer. Both the computer and the printer drive the interface cable with TTL (Transistor-transistor logic) gates. The outputs are single- ended, which means that each signal is referenced to the computer ground. TTL is fine for short cable runs, less than ten or twenty feet. The problem with TTL is that a logical zero level is only from 0 volts to about 1 volt, and a TTL output only drives down to around .2 volts. This means that less than a volt of noise can cause the receiving end to see a one where a zero was intended. The longer a cable is, the more likely it is to pick up noise, so the possibility of an error increases as the cable gets longer. The problem is made worse if the computer and printer are plugged into different wall outlets, as the grounds from the outlets may be at slightly different potentials, which can increase the possibility of errors. For this reason, TTL signal levels are normally used only for short cable runs, less than ten or twenty feet. TTL can and is used over longer distances, but reliable operation depends on the cable quality and the level of ambient electrical noise. To reliably drive my printer, I had to change the signals from TTL levels to something else. Of course, I wanted to do this external to the computer, without changing the existing printer interface board.

I considered using RS-232. RS-232 signal levels are also single-ended, but the voltage levels are greater. RS-232 outputs swing between positive and negative voltages for greater noise immunity. RS-232 is intended for serial communications, but there is no reason that it could not be used for a parallel interface as well. RS-232 has been used over very long distances, but it is only specified for about fifty feet at high baud rates. The signal timings for a parallel printer port correspond to a very high serial data rate, so I chose not to use this approach. There are other schemes to reduce the noise sensitivity of single-ended signals, but I went a different route.

RS-422 (see Figure 1) is a different kind of interface. RS-422 is a differential interface. This means that each signal is not referenced directly to ground, but to another signal. In an RS-422 interface, each signal is sent on a pair of wires. As one wire of a pair goes high, the other wire goes low. The receiver, at the other end of the cable, does not look for one wire to change with respect to ground, but instead looks for one of the wires to change with respect to the other. The voltage difference between the two wires in the pair determines the logic state, not the voltage with respect to ground. Noise that is picked up on the cable or caused by a potential difference in the grounds will be common mode noise. This means that both signals in the pair will be affected the same way, so the receiver will see no change in the difference voltage. For this reason, RS-422 is very immune to noise, and is specified to work at data rates up to 100k bits/sec at distances of up to 1000 meters (that's about 3000 feet).

How the Cable Driver Works

Conceptually, the cable driver is very simple. The driver consists of two boxes, one at the computer, and one at the printer, connected by a ribbon cable. The circuitry in the driver box, at the computer end, takes the data and command signals that are output from the computer, converts them to RS-422 levels, and drives the connecting ribbon cable with them. All of the status lines, which are inputs from the box at the printer end of the cable, are converted from RS-422 to TTL and sent to the computer.

The receiver box, at the printer end, does the exact opposite; signals that are outputs from the computer box are converted from RS-422 to TTL and sent to the printer, and signals output from the printer are converted to RS-422 and sent up the cable to the computer box.

Both boxes are powered with a wall transformer/power supply at the computer end; power is sent up the cable from the computer box to the printer box. Note that even though the board at the computer end is called the driver because it drives the data and control lines, it also receives the status lines. Similarly, the board at the printer end is called the receiver because it receives the data

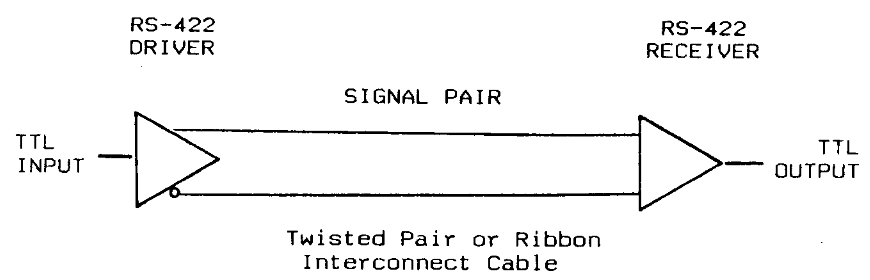


Figure 1: Typical RS-422 connection.

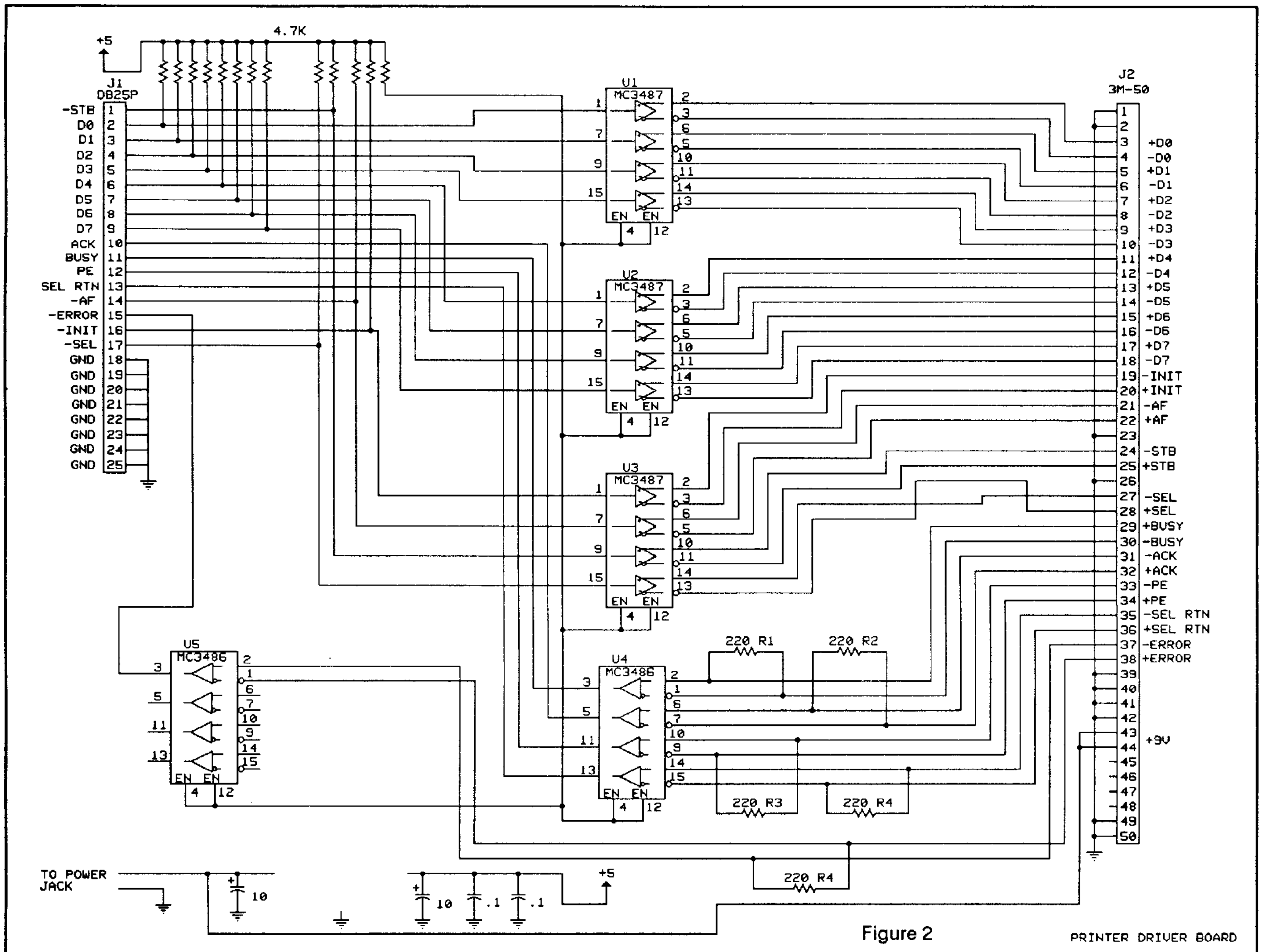


Figure 2

and command lines, but it also drives the status lines back to the driver board.

The interconnecting cable between the two boxes is a fifty conductor ribbon cable. I only needed a length of about fifty feet, but the circuit has been used over a distance exceeding 100 feet, and will work over much greater distances. Each RS-422 signal is terminated at the receiving end with a 220 ohm resistor. This value was chosen to reduce the power requirements for the circuit. If you are going to drive a very long cable (500 feet or so), you should change the 220 ohm resistors to 100 ohms, but you will need a bigger power supply.

Circuit Construction

There is nothing particularly difficult about circuit construction. The prototype circuits, shown in Figures 2 and 4, were constructed on perfboard, with the 25 pin connector at one end, and the 50 conductor ribbon connector at the other end. The component placement is shown in Figures 3 and 5. On the printed circuit boards, to simplify mounting, the 25 pin connector

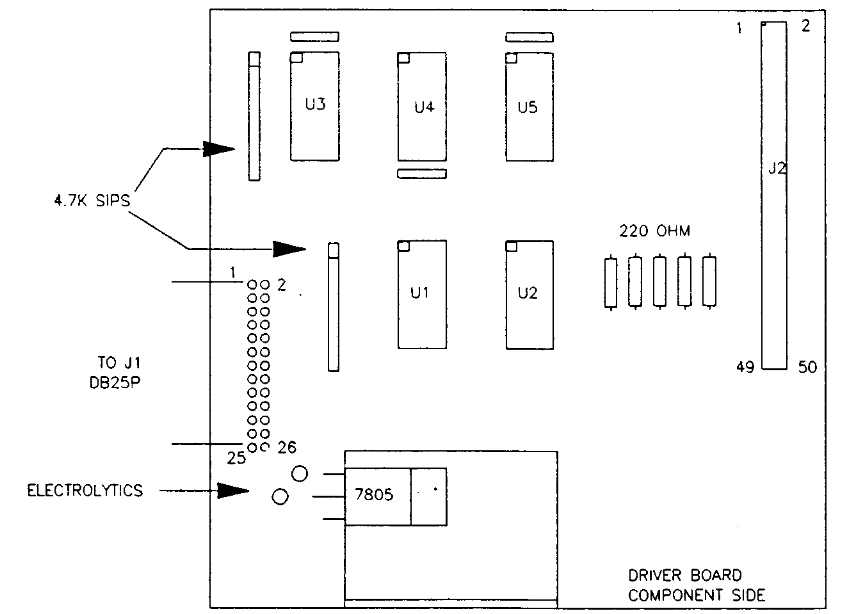


Figure 3: Driver board component placement.

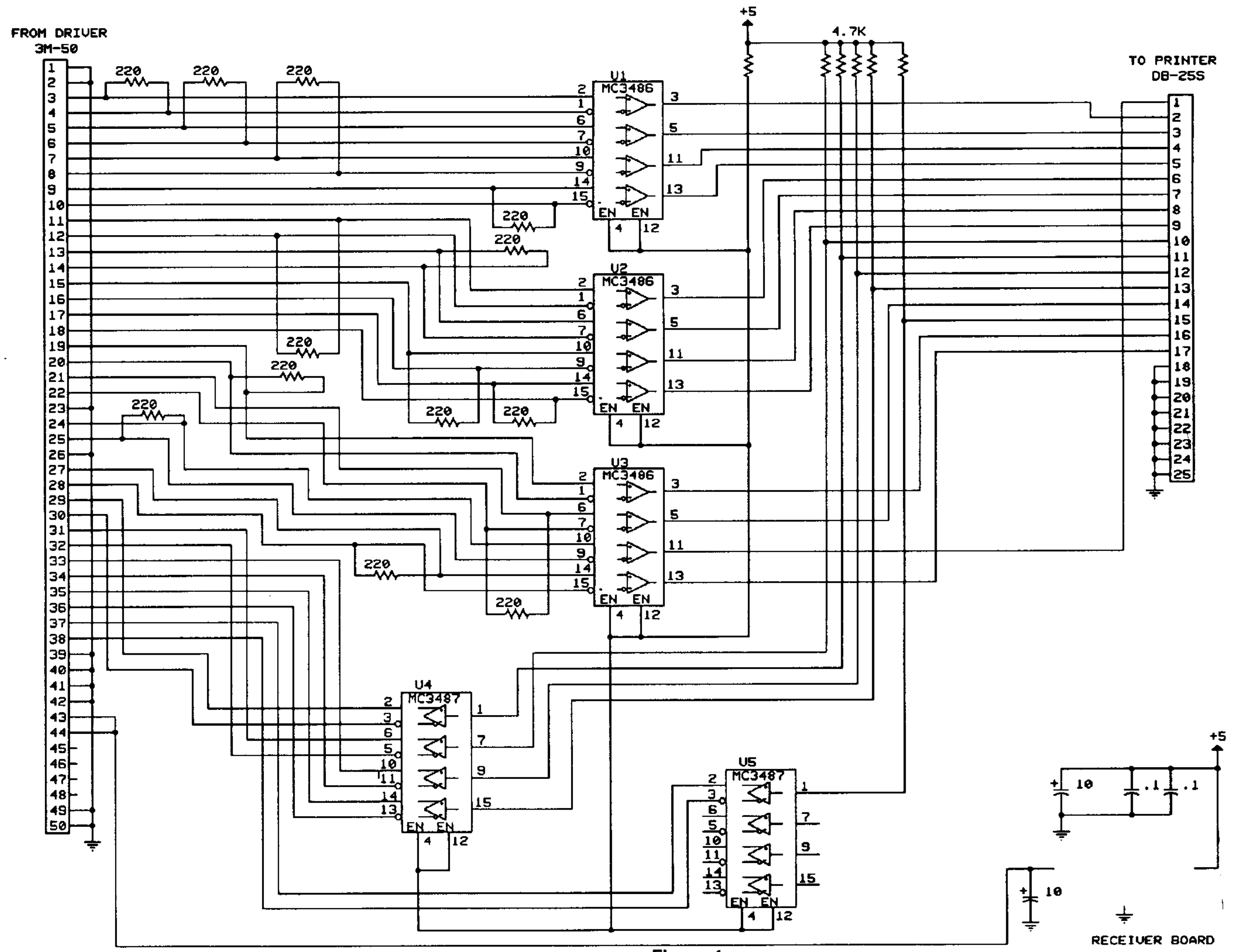


Figure 4

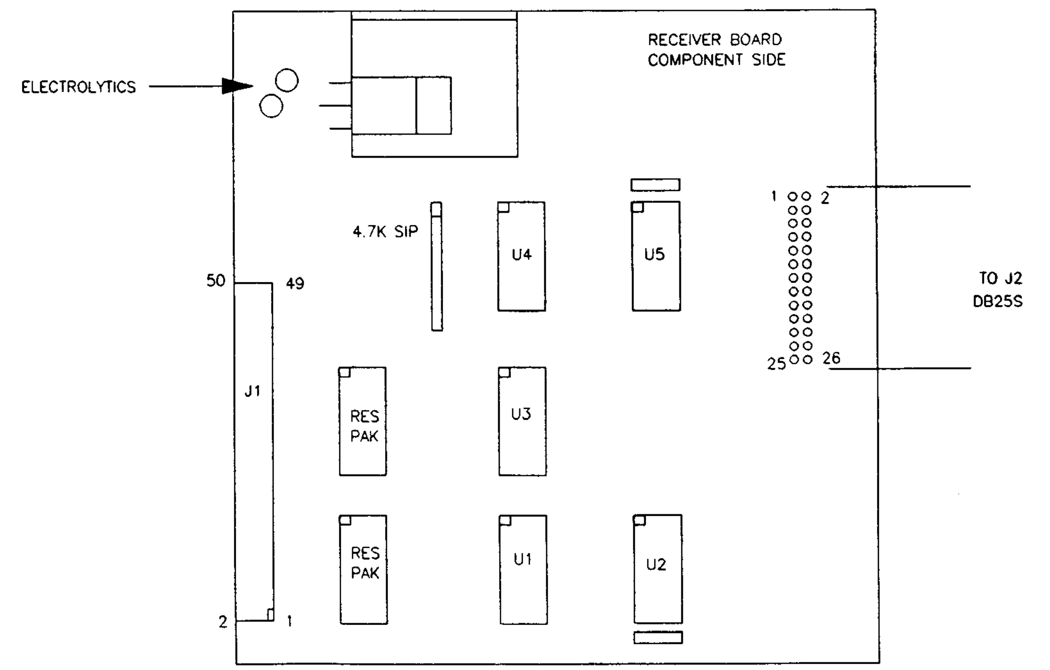


Figure 5: Receiver board component placement.

does not mount directly to the boards themselves. Instead, a 26-pin ribbon cable header mounts to the boards, and the 25 pin connector is an assembly consisting of the 25 pin connector, a 26 pin ribbon header, and a short interconnecting ribbon cable. See Figure 6 "DB25 Cable Assemblies" for details.

The driver circuit at the computer end has a jack for a 9vdc, 1 amp wall transformer. The receiver, at the printer end, has no wall transformer, but 9 volts is passed up the ribbon cable to the receiver. Both boards have a 7805 5 volt regulator to bring the 9v level

PARTS LIST - DRIVER BOARD

QUAN	DESCRIPTION
3	MC3487 RS-422 DRIVER ICs
2	MC3486 RS-422 RECEIVER ICs
2	10K SIP PULLUP RESISTOR PACKAGES 10 pins, 9 resistors per package.
5	220 OHM, 1/4 W RESISTORS
1	COAX POWER CONNECTOR (RADIO SHACK #274-1565)
1	7805 REGULATOR IC
1	DB25P IDC (ribbon) CONNECTOR
1	26 PIN HEADER 3M 3593 series or equivalent
6"	26 CONDUCTOR RIBBON CABLE
1	50 PIN HEADER, 3M 3596 series or equivalent.
2	.1 UF CAPACITORS
2	10 uF, 25V ELECTROLYTIC CAPACITORS
1	9V 1A WALL TRANSFORMER POWER SUPPLY

PARTS LIST - RECEIVER BOARD

QUAN	DESCRIPTION
2	MC3487 RS-422 DRIVER ICs
3	MC3486 RS-422 RECEIVER ICs
2	220 OHM DIP RESISTOR PACKAGES, 8 RESISTORS/PACKAGE
1	10k SIP PULLUP RESISOTR PACKAGE 10 pins, 9 resistors per package.
1	7805 REGULATOR IC
1	DB25S IDC (ribbon) CONNECTOR
1	26 PIN HEADER, 3M 3593 series or equivalent
6"	26 CONDUCTOR RIBBON CABLE
1	50 PIN HEADER, 3M 3596 series or equivalent.
2	.1 UF CAPACITORS
2	10 uF, 25V ELECTROLYTIC CAPACITORS

MISC: Perfboard, hookup wire, 50 pin ribbon cable with connectors, chassis boxes.

Note: A pair of blank printed circuit boards is available from the author, for a cost of $20 per board or $40 per pair.

Table 1: INPUT AND OUTPUT CONNECTIONS

DRIVER = DRIVER BD,
RECEIVER = RECEIVER BD

SIGNAL	INPUT CONNECTOR AND PIN NO.	OUTPUT CONNECTOR AND PIN NO.	RIBBON CABLE CONNECTIONS H	L
D0	DRIVER J1-2	RECEIVER J2-2	3	4
D1	DRIVER J1-3	RECEIVER J2-3	5	6
D2	DRIVER J1-4	RECEIVER J2-4	7	8
D3	DRIVER J1-5	RECEIVER J2-5	9	10
D4	DRIVER J1-6	RECEIVER J2-6	11	12
D5	DRIVER J1-7	RECEIVER J2-7	13	14
D6	DRIVER J1-8	RECEIVER J2-8	15	16
D7	DRIVER J1-9	RECEIVER J2-9	17	18
-INIT	DRIVER J1-16	RECEIVER J2-16	19	20
-AF	DRIVER J1-17	RECEIVER J2-17	21	22
-STB	DRIVER J1-1	RECEIVER J2-1	24	25
-SEL OUT	DRIVER J1-17	RECEIVER J2-17	27	28
BUSY	RECEIVER J2-11	DRIVER J1-11	29	30
-ACK	RECEIVER J2-10	DRIVER J1-10	31	32
-PE	RECEIVER J2-12	DRIVER J1-12	33	34
-SEL IN	RECEIVER J2-13	DRIVER J1-13	35	36
-ERROR	RECEIVER J2-15	DRIVER J1-15	37	38

Why Not Serial?

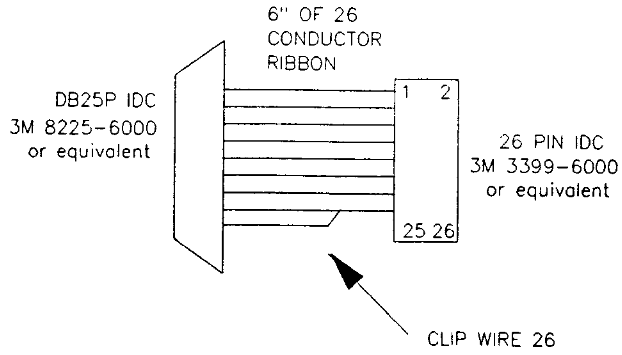
There are several products available that will allow you to move your printer a long distance from the computer. I did not use any of these for a couple of reasons, but mainly because they all have one thing in common: they convert the parallel information from the computer into serial. This means sending information 1 bit at a time instead of 8 bits at a time. If you are printing text, that is probably fine. But if you are printing graphics, a serial interface can be a real bottleneck. As an example, I have seen a laser printer connected to a serial interface, printing a 300 dots per inch graphics page, that required 15 minutes to send the page to the printer. This same printer, printing the same page, required about one minute when it was connected to the parallel printer interface on the same computer. Why does it take so long? Well, look at the math. If your graphics page is to be printed on 8.5 x 11 inch paper, with one-half inch margins all around, then the graphics printing area is 7.5 x 10 inches. 7.5 inches x 300 dots/inch by 10 inches x 300 dots/inch is 6,670,000 bits of information. A serial interface always has at least 20% overhead, because a serial byte consists of 8 data bits and at least one start bit and one stop bit. So at 9600 baud, our 6.6 million pixel page needs 14.6 minutes to to send. At 19.2k baud, it takes 7 minutes, and at 38.4k baud it takes 3.6 minutes.

down to 5v for the logic. The 7805s must be heatsinked to a fairly large piece of metal. I recommend using an aluminum chassis for the boards, and mounting the regulator to the chassis with a chunk of right angle aluminum. This allows the chassis to provide the heat sink. If you don't use an aluminum chassis, make the heat sink from a strip of aluminum about 1½ inches wide, 2 inches long, and about 1/16 inch thick.

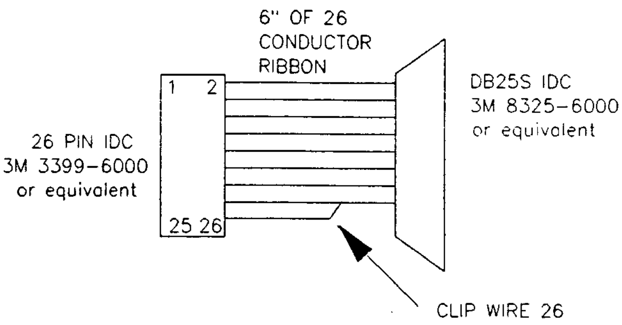
The MC3487 ICs are the RS-422 drivers, and the MC3486s are the RS-422 receivers. The 220 ohm terminating resistors are in DIP packages, although discrete resistors could be used. If you change the terminators to 100 ohms for long cable runs, you will need a bigger power supply than the one specified, or you can put a supply at each box. If you use two separate supplies, be sure not to connect the cable pins that carry 9v, or you will short the two supply outputs together. Be sure to install the 0.1 uf decoupling capacitors on the board. Also use two electrolytic capacitors on the 7805 regulators to prevent oscillation. These capacitors should be 10 to 25 uf, 25v. One should be connected from the input of the 7805 to ground, and one from the output to ground.

Circuit Checkout

After the circuit boards are constructed, and before installing the ICs into their sockets, connect the power supply to the driver board. Plug in the power supply, and check the voltage from pin 8 to pin 16 of each IC socket with a DVM or VOM. All of the ICs should have +5v on pin 16, and ground on pin 8. If this checks out, unplug the wall transformer and plug in the 50 conductor ribbon cable. Connect the receiver box to the other end of the cable, plug the transformer back in, and perform the same voltage check on the receiver that you did on the driver. If the



DRIVER CABLE



RECEIVER CABLE

IDC-26 PIN NO.	DB25 IDC PIN NO.	IDC-26 PIN NO.	DB25 IDC PIN NO.
1	1	2	14
3	2	4	15
5	3	6	16
7	4	8	17
9	5	10	18
11	6	12	19
13	7	14	20
15	8	16	21
17	9	18	22
19	10	20	23
21	11	22	24
23	12	24	25
25	13	26	N.C.

DB25 cable assemblies.

receiver voltage is correct, unplug the wall transformer and install the ICs. Be sure not to plug any ICs in backwards or bend the IC legs under the IC body. With the ICs installed, plug the transformer back in, and check the power pins on one IC on each board to insure that both boards still have +5v. If you construct the circuit using the printed circuit boards, you should still check the driver and receiver voltages before connecting to the computer or printer. A solder bridge could connect 9v to one of the signal lines.

At this point, it is a good idea to make sure that all of the inputs and outputs work. If you are the daring type, you can just skip over this part and hook the boards to the computer and printer, and hope for the best.

If you want to check each signal for proper operation, you will need a DVM, oscilloscope, or logic probe, and a short jumper wire. Table 1 shows each signal, which board and connector/pin is the input for the signal, and which board and connector/pin is the output. Referring to Table 1, use a jumper wire to connect each input pin shown to ground. Check the corresponding output pin with the DVM, 'scope or probe. With the input pin grounded, the output pin should go low. After checking for a low, remove the grounding jumper (all inputs have 10k pullups to insure that they go high when open). Now the corresponding output should go high. If any input does not work correctly, you will need to isolate it to one board or the other (remember, each signal has a driver on one board and a corresponding receiver on the other board). To do this, probe the 50 pin cable connections (also shown in Table 1) for the correct high and low states. The pins labeled H should go high when the input is high and low when the input is low. The pins labeled L are inverted; they go low when the input goes high and high when the input goes low. If any of the signal outputs don't respond, and the differential signals on the ribbon work as they should, then the problem is in the board that has the receiver. If the differential signals don't work either, then the problem is in the driver board or the ribbon cable itself. (To check the ribbon, unplug it and check the differential signals again). Correct any wiring errors before connecting the boards to your computer and printer.

Circuit Operation and Use

To use the printer driver, use a short cable to connect the 25 pin connector on the driver circuit to the printer output of the computer. The cable that would ordinarily hook the printer to the computer will now connect the printer to the 25 pin connector on the receiver. Connect the driver box to the receiver box with the 50 pin ribbon, plug the wall transformer output into the driver power jack, and you are ready to print. No special software is needed, just print as usual. ●

Embedded Systems for the Tenderfoot

Communicating with the Real World

by Tim McDonough

In the last issue of TCJ, I presented a simple 8031 Single Board Computer circuit that emulated an exclusive-OR gate. The focus of that article was to get you thinking about embedded systems and provide an overview of the hardware and software required to begin developing embedded systems of your own.

I hope that you've tried the brief example presented last month and expanded on the idea, perhaps by mimicking the function of other logic packages. You've probably been amazed at how much you can accomplish after mastering only a few of the 8031's instructions.

One of the more common projects that people like to consider building is a gadget of some sort that communicates with their PC via the serial communications port. The 8031 is a natural for this type of gadget since it has a built in serial port that can be programmed to operate from 1200 to 19200 baud quite easily.

The subject for this issue is the minimal additional hardware required for a serial port and the software needed to interpret commands sent from the PC and carry them out. The particular project presented allows you to send commands to the 8031 board that will control two bits of an output port that might be used to control a pair of relays. As with the last issue's example, the important thing is how you program the 8031 for it's task, not what the relays may actually do.

Figure 1 shows the basic 8031 computer presented last month with an additional integrated circuit. The MAX232 converts the 0 and 5 volt logic signals supplied by the 8031's UART to the +/-12 volt levels used in RS232 communications. Note that the serial port on our 8031 computer is "RS232 Compatible." What this means in this case is that I've chosen to leave out all of the hand shaking signals found in a "real" serial port and implement only the transmit (Tx), receive (Rx), and ground (GND) connections. For basic communications needs where either no hand shaking or software hand shaking is used, these three lines are all that is required.

Listing 1 is somewhat longer than the XOR.ASM program presented in the last issue but if we look at it a little piece at a time, there's nothing to it.

RELAY.ASM contains several major sections of code. It also uses three subroutines to make repetitive tasks easier. The program as a whole allows you to energize or de-energize one of two relays whose drivers are attached to bits 3 and 4 of Port 1.

Tim McDonough is the President of Cottage Resources Corporation. The company manufactures and distributes several single board computers based on the 8031 and is a dealer for PseudoCorp brand 8031 cross-assemblers and cross-simulators. He may be contacted at: Cottage Resources Corporation, Suite 3-672, 1405 Stevenson Drive, Springfield, Illinois 62703, (217) 529-7679.

The equate directive assigns some reader friendly names to the values and I/O lines we'll be using. The ASCII values for linefeed (LF), carriage return (CR) and end of text (EOT) will all be used when the program sends messages from the 8031 system to the user's terminal. The "relay1" and "relay2" equates identify the I/O lines I used for my two relays.

Before I proceed, there is another important reason for using the equate directive besides making your code easier to read. It can also help make modifications simpler in the future. Suppose, for example, that this program were hundreds of lines long and not just a couple dozen. By using the equate to assign the name "relay1" to Port 1, Bit 3, I need only edit the equate statement and make appropriate hardware modifications if I want to attach the relay to Port 1, bit 7. I won't have to search my code for every occurrence of "P1.3" and hope that I didn't overlook any. Get into the habit of using equates now and your life will be simpler as your projects grow in complexity. End of lecture.

The next few lines of code are the most cryptic of the lot. They set up the serial port in the 8031 to operate at 1200 bps using an 8-bit word, no parity and one stop bit. I chose these settings because they are very commonly used when using a communications program to talk with a modem and most of you will already have a PC set up to use these parameters.

The serial port of the 8031 has several modes of operation. Each could easily be the topic of one or more articles in itself. The relay application uses what Intel calls Mode 1. All of the modes are described in detail in the Intel Embedded Controller Handbook, Volume I.

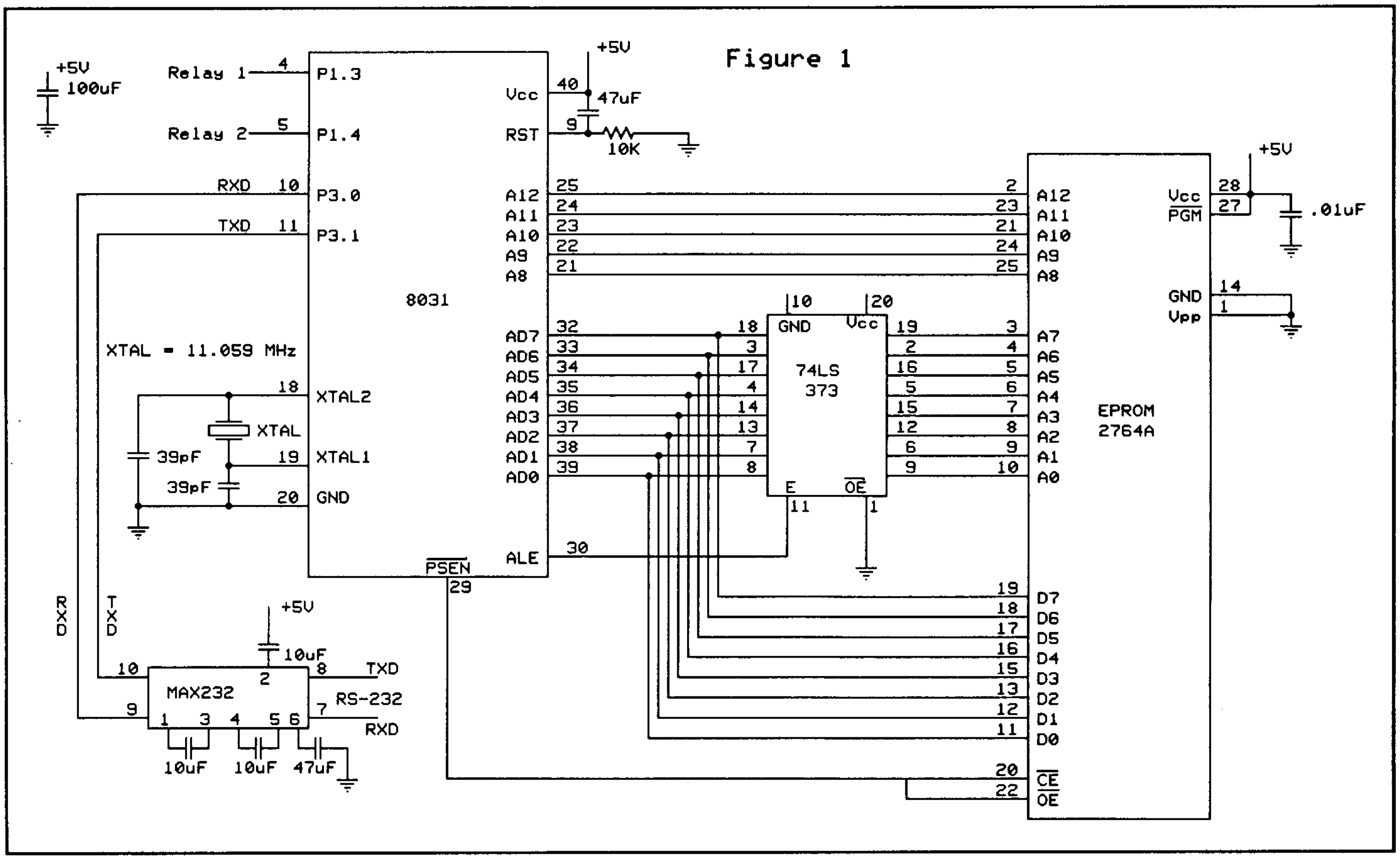
The 8031's UART is controlled by the SCON (Serial CONtrol) register. This register is used to set the mode, receiver enable flag, transmit and receive bit 8 (not used in Mode 1), the transmit and receive interrupt flags and another bit that is used in a special mode for multi-processor communications. These registers are shown in Figure 2 along with their desired initial values.

Editor's Note: The bits are numbered 0 thru 7 for 8 bit code. Bit 8 is the 9th bit.

The difference between the TI and RI flags will be explained in a moment. The value that we load into SCON using the MOV instruction is 01010010 (binary), 52 (hexadecimal) or 82 (decimal).

The 8031 uses interrupts to communicate the status of the transmitter and receiver. Initially the receiver interrupts are enabled so we will be able to receive characters that make up the system commands. The transmit interrupt is disabled since we have no characters to send.

The UART uses TIMER1 of the 8031 for baud rate generation. Setting bit 6 of the TCON (Timer CONtrol) register allows TIMER1 to run. The mnemonic symbols TR1 and TCON.6 may



be used interchangeably to access this bit although TR1 seems to be the more common.

The next register to be programmed is TMOD (Timer/counter MODe control). This register actually uses the four most significant bits for TIMER1 and the four least significant bits for TIMER0. Figure 3 shows this register and the conditions required by our application.

A bit in the PCON (Power CONtrol) register must also be set. PCON.7 (also referred to as SMOD) determines whether or not the overflow of TIMER1 will be divided by 2 or fed straight into the UART. This bit is cleared for our program.

Editor's Note: This bit was cleared upon reset.

The final step in setting up the serial port is to establish the baud rate. This is done by loading the proper timer value into the TH1 register. This is where the 11.059MHz crystal comes into play. Using this crystal value, baud rates from 1200bps to 19.2bps are easily programmed using values from the following table.

Baud Rate	SMOD Value	Reload Value
19,200	1	FD
9,600	0	FD
4,800	0	FA
2,400	0	F4
1,200	0	E8

Don't worry if you have to read through the data sheets a few times before the serial port starts to make sense. It's probably one of the tougher things to understand, but using it effectively is essential to building embedded systems that will interface to almost any sort of host computer imaginable.

Before discussing the main portion of the program, there are three subroutines that are used in most of the applications I write when the serial port is used. The first one, "recv", waits for a character to arrive at the serial port. The compliment of "recv"--"send" is used to transmit a byte out the serial port. Finally a routine called "print" makes calls to the send routine to transmit system messages to the host computer.

The pseudocode for the recv subroutine is simple:

```
"recv"
        Wait for a character to be received
        Clear the receive interrupt flag
        Copy the byte into the accumulator
Return
```

The JNB instruction causes a jump to the label (recv) specified if the bit (RI) is not set. This will cause the program to execute this statement over and over until a byte is received. Next, the RI flag is cleared so another byte can be processed by the UART. Finally, the byte received is copied into the accumulator register and the subroutine ends.

The "send" routine works in a similar fashion except that the transmit interrupt (TI) flag is checked to determine when the UART transmitter is ready to receive another character. When it is ready, the next byte is copied from the accumulator to the SBUF register.

The last subroutine implements a "print" statement on the 8031. The technique I use to store canned messages in the EPROM is to use a label for each message followed by a series of bytes whose ASCII values make up the message I'll want to transmit. The "howdy" label near the end of Listing 1 is a good example.

The ".db" assembler directive tells the PseudoSam assembler to store the bytes that follow in the assembled code. PseudoSam will translate text within double quotes into the proper values for you. The CR and LF were previously defined using the equate directive at the beginning of the program. The "print" routine relies on each distinct message being terminated with the End Of Text (EOT) character. This character could have been anything. EOT was chosen because of it's name and the fact that it is a non-printing character and will never be needed in a system message.

Keeping the message format in mind, the pseudocode for the "print" routine is as follows:

```
"print"
        Get the byte pointed to by DPTR
        If the byte equals EOT then return
        Transmit the byte
        Increment the data pointer
        Goto "print"
```

Before calling "print" the program loads the starting address of the message into the DPTR (Data PoinTeR) register. Print clears the accumulator to zero and then copies the value stored at the location pointed to by DPTR plus the value in the accumulator, into the accumulator. By always clearing the accumulator first, we are essentially copying just the byte into the accumulator.

Next, the value in the accumulator is compared to the EOT character. If the two are equal, the message is done and the subroutine returns. If the two are not equal, "send" is called to transmit the byte, the value of DPTR is incremented so that it now points to the next character and the SJMP (Short JuMP) takes us back to the beginning of the routine.

So far the serial port has been initialized and there are routines to handle sending messages and receiving bytes via the port. The only thing left is the main program that will interpret the commands and carry out your requests. Trust me, it's all down hill from here on out.

When I build a system that controls something in the real world, I usually don't like surprises. You should always include some sort of communications protocol to help your system recognize errors. If a command you send to a device gets garbled by line noise, cosmic rays, or whatever, ideally the computer should ignore it. Without any precautions the best you can hope for is that nothing will happen because the computer didn't understand. If Murphy is with you, as he most always is, your device may do something unexpected--like pour hot tea in Aunt Martha's lap instead of passing her the sugar.

The scheme I use here is to prefix each valid command with the "@" symbol. The software then requires two particular characters in sequence before any action is taken, thus avoiding unexpected results. The following com-

Figure 2: SCON Serial Port Control Register

```
+----+----+----+----+----+----+----+-------+
| SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI  | RI  |
+----+----+----+----+----+----+----+-------+
Bit 7                                   Bit 0

SM0 = 0   Mode 1, 8-bit with variable baud rate operation
SM1 = 1

SM2 = 0   Always zero for Mode 1

REN = 1   Enable the receiver

TB8 = 0   Ignore the ninth data bits (parity) for both the
RB8 = 0   transmitter and receiver

TI  = 1   Indicate the transmitter buffer is empty

RI  = 0   Indicate the receiver buffer is empty
```

Figure 3: TMOD Timer/Counter Mode Control Register

```
|          TIMER 1          |          TIMER 0          |
+------+------+------+------+------+------+------+------+
| GATE | C/*T |  M1  |  M0  | GATE | C/*T |  M1  |  M0  |
+------+------+------+------+------+------+------+------+
Bit 7                                             Bit 0

Gate = 0  Timer 1 is free running.
C/T  = 0  Selects timer mode, not counter mode.
M1   = 1  Use as an 8-bit auto reload timer. TH1 holds
M0   = 0  a value that is reloaded into TL1 each time it
           overflows.

The remaining 4 bits need only be set if your application
uses TIMER0 for some purpose.
```

Listing 1

```
; ========================================================================
; PROGRAM:      RELAY.ASM
; AUTHOR:       Tim McDonough
;               Cottage Resources Corporation
;               Suite 3-672, 1405 Stevenson Drive
;               Springfield, IL 62703
;               (217) 529-7679
; REVISION:     1.1
; DATE:         June 2, 1990
;
; PURPOSE:      This program demonstrates a method of using the 8031 serial
;               port and a simple command parser.
;
;               This code is formatted for PseudoSam Level II Version 2.2
; ========================================================================

;       Assemble the code to begin execution at memory location 0 and
;       establish some equates to make life easier on the programmer!

        .org    D'00            ;Assemble to run from location 0 (decimal)

        .equ    LF,D'10         ;ASCII Linefeed
        .equ    CR,D'13         ;ASCII Carriage Return
        .equ    EOT,D'4         ;ASCII End of Text
        .equ    relay1,P1.3     ;Control line for relay 1
        .equ    relay2,P1.4     ;Control line for relay 2

;       8031 serial port initialization
;       This device operates at 1200 baud, 8 bit, 1 stop bit, no parity
;       Data received at the serial port is NOT echoed back to the host.
;       ---------------------------------------------------------------

        mov     scon,#H'52      ;Mode 1, 8 bit operation
        setb    TR1             ;Start Timer 1
        mov     TMOD,#H'20      ;8-bit auto-reload, free running timer
        mov     TH1,#H'E8       ;1200 baud operation
```

(Listing 1 continued on next page)

(Listing 1 continued from previous page)

```
;       Display a brief message that lets the user know the serial port
;       has been initialized and the system is ready to go.

        mov     dptr,#howdy     ;Point to the welcome message
        acall   print           ;Display it to the terminal

;       Generic I/O system Command Interpreter (CI)
;       ------------------------------------------------------------

start:  acall   recv            ;Wait for a character to arrive
        cjne    A,#'@',start    ;Any valid command is preceded by ''@''
        acall   recv            ;Get another character
cmda:   cjne    A,#'A',cmdb     ;If not A, check for the B command
        clr     relay1          ;Latch relay 1
        ajmp    start
cmdb:   cjne    A,#'B',cmdc     ;If not B goto C command
        setb    relay1          ;Unlatch relay 1
        ajmp    start
cmdc:   cjne    A,#'C',cmdd     ;If not C goto D command
        clr     relay2          ;Latch relay 2
        ajmp    start
cmdd:   cjne    A,#'D',cmdh     ;If not D goto h command
        setb    relay2          ;Unlatch relay 2
        ajmp    start
cmdh:   cjne    A,#'?',err      ;If not ? goto start
        mov     dptr,#help      ;Load the address of the help text
        acall   print           ;''Print'' the message to the serial port
        ajmp    start           ;Go back to the start of the main loop
err:    mov     dptr,#error     ;Load the address of the error message
        acall   print           ;''Print'' it
        ajmp    start

;       Subroutine to wait for a byte to arrive at the serial port
;       ------------------------------------------------------------

recv:   jnb     RI,recv         ;Wait for a character to arrive
        clr     RI              ;Clear the Receive Interrupt flag
        mov     A,SBUF          ;Move the character into the A register
        ret

;       Subroutine to transmit a single byte out the serial port
;       ------------------------------------------------------------

send:   jnb     TI,send         ;Wait until the transmitter is ready
        clr     TI              ;Clear the Transmit Interrupt flag
        mov     SBUF,A          ;Move the character into the serial
buffer
        ret

;       Subroutine to send a stream of characters out the serial port
;       ------------------------------------------------------------
print:  clr     a               ;Clear the A register
        movc    a,@a+dptr       ;Get the next character
        cjne    a,#EOT,pchar    ;Print character if not EOT
        ret                     ;or return if it was ETX
pchar:  acall   send            ;Transmit the character
        inc     dptr            ;Increment DPTR
        sjmp    print           ;Do it again

;       System Messages
;       ------------------------------------------------------------
howdy:  .db     CR,LF
        .db     ''RELAY -- Version 1.1'',CR,LF
        .db     ''Type '@?' for help'',CR,LF,CR,LF
        .db     EOT

error:  .db     CR,LF,CR,LF
        .db     ''** ERROR **   Unknown Command Received'',CR,LF
help:   .db     CR,LF
        .db     ''Valid Commands:'',CR,LF,CR,LF
        .db     ''@? - Display this help message'',CR,LF
        .db     ''@A - Latch Relay #1'',CR,LF
        .db     ''@B - Unlatch Relay #1'',CR,LF
        .db     ''@C - Latch Relay #2'',CR,LF
        .db     ''@D - Unlatch Relay #2'',CR,LF
        .db     EOT

        .end
```

mands are implemented in RELAY.ASM:

@A - Latches Relay #1
@B - Unlatches Relay #1
@C - Latches Relay #2
@D - Unlatches Relay #2
@? - Displays a help screen

The pseudocode for the main program is as follows:

```
"start"
            call receive
            if byte <> "@" then goto "start"
            call receive
            if byte <> "A" then goto "cmdb"
            energize relay #1
            goto "start"
"cmdb"      if byte <> "B" then goto "cmdc"
            deenergize relay #2
            goto "start"
"cmdc"      if byte <> "C" then goto "cmdd"
            energize relay #2
            goto "start"
"cmdd"      if byte <> "D" then goto "cmdh"
            deenergize relay #2
            goto "start"
"cmdh"      if byte <> "?" then goto "err"
            print help message
            goto "start"
"err"       print error message
            goto "start"
```

In operation the "recv" routine is called and the program loops until the received byte is the "@" symbol which precedes each valid command. Next the second byte received is checked against all known commands. When a match is found, the appropriate action is carried out and the main loop starts again waiting for the next command.

The SETB and CLR instructions should be familiar to you from the previous article. The real hero of the main loop is the CJNE (Compare and Jump if Not Equal). In the RELAY.ASM code, the instruction compares the byte in the accumulator to the value of the byte that follows. If the two are not equal, the program jumps to the label indicated where execution continues.

Using the serial port for communications makes a 4 chip 8031 single board computer a powerful extension to your personal computer. Although the serial port uses two of the available I/O lines on Port 3 of the 8031 you're still left with 14 bit addressable lines that you can use for input or output. In an upcoming issue, I'll present a project that builds on the concepts of RELAY.ASM to provide a small data acquisition system that has digital inputs and outputs, as well as an analog to digital converter that will let you measure nearly anything that you can convert to a DC voltage.

If you decide to experiment more with the 8031, the basic circuit described in this article can be used over and over for a variety of projects. Some people will want to wire-wrap or point to point wire their own board to make it exactly what they want. If you don't have the time to build or would rather spend the bulk of your time coding, an assembled and tested version of this circuit that includes the circuitry for an RS232 compatible serial port is available from Cottage Resources Corporation. The Control-R I (pronounced "Controller One") is available for $39.95 and the PseudoSam 51 cross-assembler is available for $50.00 (plus $3.00 Shipping per order) from Cottage Resources Corporaion, Suite 3-672, 1405 Stevenson Drive, Springfield, IL 62703. The MAX232 IC that is required for serial port operation is optional and is available for $6.95. ●

References:

Embedded Controller Handbook, Volume I 8-bit, 1988, Intel Corporation

Mastering Digital Device Control by William Houghton, 1987, SYBEX. (Editor's Note: This book is out of print.)

Foundational Modules in Module 2

Abstract Data Types and Information Hiding

by David L. Clarke

Introduction

I have been using Modula 2 for five years now as both a teaching and a hobby language. I teach Computer Science part time at the Hartford Graduate Center. My hobby programming is done on a CP/M system with Z3PLUS. I would prefer to do the programming for my full time job in Modula 2, but it's not available on that system. I find Modula 2 to be an excellent systems programming language. It has the abstractions that one would expect from a high level language, but it is also capable of some of the low level operations that one usually writes in assembly language. On top of that, it is an ideal vehicle for such Software Engineering principles as 'Abstract Data Types' and concurrency. I hope this article will help you begin to appreciate the language as much as I do.

In Dave Moore's inaugural Modula 2 article (TCJ # 35), he presented many of the benefits of modules. These can be summarized as follows:

1. A task can be divided into smaller sections (i.e. modules), which can be written by different programmers in parallel.
2. Definition modules provide a means for each programmer to specify exactly what his module can be expected to do. Without this, it would not be possible to write a higher module until all the lower modules it calls are completed.
3. The module's internals are 'visible' to the author alone. (This is because they are kept in an 'implementation module' which the author maintains.) If the code contains an error, the author is the one who should correct it. More important, no one else should make any changes, in fact, no one else can so long as the author maintains the module.
4. As long as he doesn't change the module's definition, the author may modify or improve his code without effecting any other module.
5. The Module 2 linker makes sure that all modules refer to the latest module definitions, that is, implementation modules refer to their latest definition module, and the latest definitions of all imported modules was referenced during compilation. This enforces strong typing across modules which in turn reduces program errors.
6. Standard Libraries and user built libraries serve as a foundation for new programs and prevent us from having to reinvent the wheel over and over again.

David Clarke was originally an Electrical Engineer at Pratt & Whitney Aircraft until he discovered that it was more fun to program the data acquisition systems that he developed. He therefore became a systems programmer.

Dave is also an Adjunct Assistant Professor at the Hartford Graduate Center in Hartford CT., where he has taught courses in Systems Programming, Software Engineering, and Real Time Programming. Dave can be reached at the Graduate Center where his electronic mail address (Internet) is davec@mstr.hgc.edu. His home address for regular mail is P.O. Box 328, Tolland, CT. 06084.

One of the major concerns of programming a decade ago was 'type' compatibility. This was one of the major themes behind the Pascal language. The thought was that the compiler should be responsible for preventing the type of problems that happen when a REAL value is passed to a procedure that expects an INTEGER. This concern has continued in Modula 2, but the new compiler is also expected to furnish additional protection to the programmer. The new emphasis is called 'information hiding' by software engineers. Many of the benefits mentioned in the list above are related to information hiding. The thought here is to furnish the programmer with a measure of protection for his code. You may wonder why this protection is necessary. The following analogy may help supply an answer.

Programming twenty years ago was similar in many ways to a school locker room. Students are assigned lockers in a large 'common' area. They may use padlocks to protect their belongings, but these are not able to prevent a strong athlete from pulling the handles off whatever locker he wishes to use. In this traditional locker room, the necessary apparatus to receive the material is present, but the means of controlling its use is sadly lacking.

About ten years later, the idea was advanced of assigning specific subroutines to handle operations related to a given data type. This situation is like a locker room that has attendants hired to take charge of the equipment. Students indicate to the attendants where their lockers are. The attendant will then get the equipment from the locker and place the student's clothes in it for safe keeping. The fact that the attendants exist, does not prevent some of the unruly athletes from helping themselves to whatever they might find in someone else's locker. All they have to do is wait till the attendants are looking the other way. In this locker room, there is a well organized way to handle the lockers, but it lacks the security necessary to protect the contents. (Listing 1 shows a Turbo Pascal program that represents this situation. The hypothetical person who wrote the buffer handling routines may have done a great job, but he was unable to protect his data buffer from the mischievous writer of the main program code. The problem is that the definition of the 'LockerRoom' is completely visible to the second programmer. This is what Modula 2 attempts to prevent.)

The principle behind Modula 2 is more like a locker room that has a wall separating the locker area from the dressing area. There is a window in this wall through which the students request the attendants to get their equipment from their lockers. The wall prevents the hooligans from attacking the lockers without reducing the efficiency of the facility. In fact, it is possible to improve the system (such as replacing the lockers with easier to handle wire baskets) without effecting the students. Since the storage area is on the far side of the wall, the students never need to see how their equipment is being kept. (It may be interesting to compare the 'LockerRoom' above with the 'Sequence' which will be found in the SeqBuffer module below.)

Listing 1

```
PROGRAM LockerRoomModel;

{     D. L. Clarke     (revised) 11 June 1990                          }

{     This is a demonstration of the problems involved by using        }
{     global data as a 'private' data structure.  This program is      }
{     written in Turbo Pascal for two reasons:                         }
{           1. because Turbo adds the String extension to Pascal       }
{              (this simplifies the storage of items)                  }
{           2. to show the deficiencies of Pascal in hiding infor-     }
{              mation                                                  }

CONST
     { define the lockers in the locker room                           }
     NumberOfLockers = 3;        { this is a VERY small locker room    }
     NullLocker = 0;             { used to identify an illegal locker  }

     { define how much equipment fits in a locker                      }
     NumberOfItems = 24;         { BIG lockers in a small locker room  }

TYPE
     { define the items in a locker                                    }
     Item = String[72];                      {  rather strange item!!  }
     ItemNumber = 1 .. NumberOfItems;

     { define lockers and token values                                 }
     Locker = ARRAY [ItemNumber] OF Item;      { lockers hold items    }
     LockerNumber = 1 .. NumberOfLockers;      { i.e. legal lockers    }
     TokenValue = NullLocker .. NumberOfLockers;

VAR
    { These variables define the locker room and the tokens.           }
    { They should not be abused by the athletes.                       }
    LockerRoom: ARRAY [LockerNumber] OF Locker;      { the lockers     }
    TokenCount: TokenValue;                  { used to assign lockers  }

    { These procedures are performed by the locker room attendants.    }
    { They define the 'legal' ways of accessing the locker items.      }

FUNCTION Access: TokenValue;         { get initial locker assignment   }
     BEGIN
          IF TokenCount = NumberOfLockers THEN
               Access := NullLocker
          ELSE  BEGIN
               TokenCount := TokenCount + 1;
               Access := TokenCount
          END
     END;

PROCEDURE PutSeek (equipment: Item;    { put specific item in locker   }
                   SpecificItem: ItemNumber;
                   Locker: TokenValue);
     BEGIN
          LockerRoom[Locker, SpecificItem] := equipment
     END;

PROCEDURE GetSeek (VAR equipment: Item;    { get specific item in locker }
                   SpecificItem: ItemNumber;
                   Locker: TokenValue);
     BEGIN
          equipment := LockerRoom[Locker, SpecificItem]
     END;

{ The main program code simulates the athletes using the lockers       }

VAR Token: TokenValue;       { this is the athlete's locker token,     }
          { it is placed here to separate it from the 'private'        }
          { data above.                                                }
     Stuff: Item;            { this (may) be the correct contents      }

BEGIN
     { 'clean up the lockers' and get ready for a new semester         }
     TokenCount := 0;

     {  get locker assignment (token) at beginning of semester         }
     Token := Access;

     { this is an athlete putting an item in the locker legally        }
     PutSeek ('This is a legal insertion', 1, Token);

     { this is how an item can be stowed away illegally                }
     LockerRoom[1, 1] := 'This is an illegal insertion';

     { this is what the locker's owner gets when he tries to get his   }
     { belongings out of the locker.                                   }
     GetSeek(stuff, 1, Token);
     WriteLn(stuff)

END.
```

Having said my piece on information hiding, I will now introduce several Modula 2 modules that can serve as a foundation for many useful programs. I will not attempt to 'hide' my code, but will share it with the readers of this article. In this way I will be able to discuss some of the features of the language (as well as describe a few special techniques that can be used in Modula 2 programs). At the end of this article, I shall include a main program that demonstrates the utility of these foundational modules.

Many of the most useful modules describe an object. These modules consist of a data structure (that represents the object) and a series of procedures that control the operations that can be done on the object/data structure. Quite often the actual data structure will be defined in name alone (in the definition module), that is, it is what is known as 'opaque' to the user of the module.

I should mention at this time that the Module 2 compiler that I use at home is the *FTL Modula 2* compiler from Workman & Associates. Each compiler supplies its own variation of 'standard' library modules. If you use a different compiler, you may need to make some modifications to reflect these differences.

SeqBuffer

The first module that I will present is a buffer that consists of a sequence of items. I call it a SeqBuffer. The SeqBuffer is a fairly generic module that besides being useful in itself, is also a good foundation for several other modules. Listing 2 shows the module's definition. In Modula 2, this is called the DEFINITION MODULE. From the module user's viewpoint, this is the main definition of the object and its operations. (The TYPEs and PROCEDUREs defined in the DEFINITION MODULE correspond to the window in the wall of the final locker room model. They are the only access that the user has to the data structure.) It is worthwhile to examine the SeqBuffer definition now.

The basic object of this module is the 'Sequence.' It is only defined as a TYPE at this point—that is, it's opaque. A more complete definition will appear later in the IMPLEMENTATION MODULE.

Quite often the writer of code that uses modules like SeqBuffer will want to use more than one Sequence. To satisfy this need, a procedure such as 'Access' is provided. The user then must define a variable of the desired type by a statement like:

```
VAR  Seq1:  Sequence;
```

He would then assign the variable to a valid Sequence with an assignment statement that references the Access procedure:

```
Seq1 := Access("First Sequence");
```

Once a variable (e.g. Seq1) refers to a valid Sequence, it can be used in all of the other procedures defined in the module. (The 'Inaccessible' procedure may be used to determine if a variable has been assigned to a valid Sequence.)

Once a Sequence (buffer) has been accessed, its owner or user may place items in it, examine items that have been already placed within, or remove items from the buffer. When an item is placed into a Sequence, the user indicates where the item is to be placed in it. This is done by supplying an entry number or index value. Since a user may have access to several Sequences, the specific Sequence variable is passed as a parameter. Of course, the item itself must be one of the parameters. Since the item may be just about anything, the method of passing this parameter must be very flexible. This is one of the major strengths of Modula 2. According to the specification of Modula 2, the 'WORD' TYPE is compatible with any equally sized TYPE. Many compilers have extended the specification to make the 'BYTE' TYPE compatible with anything that takes up one byte of memory. In addition, any larger sized type is compatible with an equally sized array of bytes. The Modula 2 specification also allows procedure parameters to be defined as an 'open array.' An open array does not contain a range specifier that tells how many elements are in the array. When we put it all together, we assign the item a TYPE of 'AR-

Listing 2

```
DEFINITION MODULE SeqBuffer;
(*                              D. L. Clarke    (revised) 11 June 1990    *)

FROM  SYSTEM  IMPORT BYTE;              (* used to 'match' any type        *)

TYPE        Sequence;                   (* the sequence or table data type *)

PROCEDURE  Access                       (* access a Sequence for later use *)
          ( name: ARRAY OF CHAR)        (* sequence name -- usually ignored *)
                : Sequence;             (* the requested abstract data type *)

PROCEDURE Inaccessible                  (* see if Sequence is invalid      *)
          (   seq: Sequence)            (* the Sequence to check           *)
                : BOOLEAN;              (* TRUE if not valid               *)

PROCEDURE Include                       (* put a new entry into a sequence *)
          (     s: ARRAY OF BYTE;       (* the entry to put in the sequence *)
                                        (* cannot overwrite existing entry *)
                i: INTEGER;             (* the entry number or index       *)
          VAR seq: Sequence)            (* the sequence to place entry into *)
                : BOOLEAN;              (* FALSE if 'i' exceeds table size *)
                                        (* or was already in the sequence  *)

PROCEDURE GetSeq                        (* get an entry from a sequence    *)
          (VAR  s: ARRAY OF BYTE;       (* the returned entry              *)
                i: INTEGER;             (* the entry number (i.e. index)   *)
              seq: Sequence)            (* the sequence to be searched     *)
                : BOOLEAN;              (* TRUE if 'i' is in the sequence  *)

PROCEDURE PutSeq                        (* put an entry into a sequence    *)
          (     s: ARRAY OF BYTE;       (* the entry to put in the sequence *)
                                        (* may overwrite pre-existing entry *)
                i: INTEGER;             (* the entry number or index       *)
          VAR seq: Sequence)            (* the sequence to place entry into *)
                : BOOLEAN;              (* FALSE if 'i' exceeds table size *)

PROCEDURE Delete                        (* delete an entry from a sequence *)
          (     i: INTEGER;             (* the entry number (i.e. index)   *)
          VAR seq: Sequence)            (* the sequence to be searched     *)
                : BOOLEAN;              (* TRUE if 'i' was in the sequence *)

PROCEDURE Lowest                        (* lowest index yet in a sequence  *)
          (   seq: Sequence)            (* the sequence to be examined     *)
                : INTEGER;              (* the lowest index used so far    *)
                                        (* = + MAXINT for empty sequence   *)

PROCEDURE Highest                       (* highest index yet in a sequence *)
          (   seq: Sequence)            (* the sequence to be examined     *)
                : INTEGER;              (* the highest index used so far   *)
                                        (* = - MAXINT for empty sequence   *)

PROCEDURE Deaccess                      (* deaccess Sequence when finished *)
        (VAR seq: Sequence);            (* the sequence to be eliminated   *)

END SeqBuffer.
```

RAY OF BYTE.' This allows an item to be compatible with any TYPE.

The major procedure for putting items into a Sequence is 'PutSeq.' To place the string "This is an item" into the first entry (i.e. index # 1) of the Seq1 Sequence previously accessed, we would use the following call:

```
okay := PutSeq("This is an item", 1, Seq1);
```

Notice that most of these procedures return BOOLEAN values that indicate whether an error was detected while executing the procedure. For instance, if for some reason PutSeq was not able to put the item into the Sequence, it would return a value of FALSE. When PutSeq is requested to place an item in a particular entry of a Sequence, it will always place it there (if the entry exists). It will overwrite any data previously placed in that entry. At times we may prefer to not overwrite this older data. At those times the procedure to use is 'Include.' Include will only place items into previously empty entries. If the requested entry already contains data, then Include will return a value of FALSE to indicate an error condition.

The 'GetSeq' procedure is used to fetch information back out of a Sequence. The procedure returns a value of TRUE if anything exists at the desired entry in the Sequence, otherwise it returns FALSE to indicate an error condition.

The 'Delete' procedure will remove data from a specific entry in a Sequence. Once an entry is deleted, it is as if the entry had never been filled. GetSeq cannot retrieve anything from the entry, but Include can place another item into the entry space.

The 'Highest' and 'Lowest' procedures return the numbers of the highest and lowest entries presently containing data.

Finally, the 'Deaccess' procedure is used to retire a Sequence when we are finished with it.

An IMPLEMENTATION MODULE corresponds to the area on the far side of the wall in the final locker room model. This is where the data structure is kept safely out of the user's reach. Listing 3 shows my IMPLEMENTATION MODULE for the SeqBuffer. I used a linked list to hold the necessary data. Each entry in the Sequence is a 'Record' that contains several pieces of information. The item itself will eventually be an array of bytes. It will be pointed to by the 'data' entry of the Record. Since data of various sizes may be placed into the Sequence, the 'size' of each item is saved in the Record. Items may be placed into the Sequence in any order, there will probably be gaps in the entry indices. In order to keep everything in order, the 'entry' number of each item is also kept in the Record. The next item in the linked list is pointed to by 'next.'

In the DEFINITION MODULE, Sequence was opaque. Here, in the IMPLEMENTATION MODULE, we see that the complete definition is given as a 'POINTER TO Record'. A variable, SeqHead, is a special internally used Sequence that holds all of the valid Sequences that have been accessed by the users. As Users call 'Access', an entry is made in the SeqHead Sequence that corresponds to the pointer returned by the Access procedure. The 'Inaccessible' procedure searches through the SeqHead Sequence. If the Sequence parameter passed to Inaccessible cannot be found in SeqHead The procedure returns a value of TRUE meaning that the requested Sequence is truly inaccessible.

The module starts off with three procedures that are internal, that is they are only used within this module. The first one, SearchForEntry, will search through a linked list looking for a specific entry number. If the desired entry exists, the 'p' parameter will end up pointing to it; if it doesn't exist, then it must be placed in the gap between where 'q' and 'p' end up pointing. In the latter case, the new entry can be placed in the Sequence by calling 'MakeNewEntry'. This procedure creates the new Record, places it into the linked list, and saves the 'entry' index value. Finally, the actual item is placed into the Record (pointed to by 'p') by calling 'StoreEntry'. This procedure uses a Modula 2 built-in function, HIGH, to tell how big the 'item' is. HIGH assumes that an open array has a range like [0 .. n], that is the low end of the range is normalized to 0 and the upper end of the range becomes 'n.' This latter value is returned by HIGH. The actual number of bytes in an item (e.g. 's') is therefore HIGH(s) + 1. StoreEntry acquires enough dynamic memory to hold the item by calling ALLOCATE. The bytes in the item are then moved into this memory area one byte at a time.

The IMPLEMENTATION MODULE continues with the executable code for all of the procedures defined in the DEFINITION MODULE. It can be seen that 'Access' acquires enough memory for a Record by calling NEW. This Record is placed in the SeqHead Sequence and a pointer to it is returned to the caller of this procedure. In addition, Access will initialize the highest and lowest entry values to unique values that indicate an empty Sequence. (You'll have to admit that setting the lowest value to the largest possible integer value is rather unique.) As items are entered into the Sequence, StoreEntry will check to see if the requested entry index is smaller that the current lowest value; if it is, then it replaces the current value. (Note -- this will always happen when the first entry is placed into the Sequence.) A similar check is made on the highest value.

As mentioned above, 'Inaccessible' checks to see if the indicated Sequence is missing from the SeqHead list.

Listing 3

```
IMPLEMENTATION MODULE SeqBuffer;

(*         David L. Clarke         (revised)  11 June 1990          *)

  FROM  Storage    IMPORT ALLOCATE, DEALLOCATE;

  TYPE      Sequence = POINTER TO Record;
            Record = RECORD
                entry: INTEGER;
                size:  INTEGER;
                data:  POINTER TO ARRAY [0..256] OF BYTE;
                next:  Sequence
            END;

  VAR       SeqHead:   Sequence;

PROCEDURE SearchForEntry(i: INTEGER; VAR seq, p, q: Sequence);
BEGIN
    p := seq^.next;    q := NIL;
    WHILE (p <> NIL) AND (p^.entry < i) DO
        q := p;
        p := p^.next
    END
END SearchForEntry;

PROCEDURE MakeNewEntry(i: INTEGER; VAR seq, p, q: Sequence);
BEGIN
    NEW(p);
    p^.entry := i;
    IF q = NIL THEN
        p^.next  := seq^.next;
        seq^.next := p
    ELSE
        p^.next := q^.next;
        q^.next := p
    END
END MakeNewEntry;

PROCEDURE StoreEntry(s: ARRAY OF BYTE; i: INTEGER; VAR seq,
                p: Sequence);
    VAR    j: CARDINAL;
BEGIN
    p^.size := HIGH(s) + 1;
    ALLOCATE(p^.data, p^.size);
    FOR j := 0 TO HIGH(s) DO
        p^.data^[j] := s[j]
    END;
    IF i > seq^.entry THEN
        seq^.entry := i       (* update highest *)
    END;
    IF i < seq^.size THEN
        seq^.size := i        (* update lowest  *)
    END
END StoreEntry;

PROCEDURE  Access(name: ARRAY OF CHAR): Sequence;
    VAR    seq, p: Sequence;
           j:      CARDINAL;
BEGIN
    NEW(seq);
    seq^.entry := - MAX(INTEGER);
    seq^.size  := + MAX(INTEGER);
    seq^.next := NIL;
    NEW(p);
    p^.next := SeqHead;     SeqHead := p;
    p^.entry := INTEGER(CARDINAL(seq));
    ALLOCATE(p^.data, HIGH(name)+1);
    FOR j := 0 TO HIGH(name) DO
        p^.data^[j] := BYTE(name[j])
    END;
    p^.size := HIGH(name) + 1;
    RETURN seq
END  Access;

PROCEDURE Inaccessible(seq: Sequence): BOOLEAN;
    VAR    p: Sequence;
BEGIN
    p := SeqHead;
    LOOP
        IF p = NIL THEN EXIT END;
        IF p^.entry = INTEGER(CARDINAL(seq)) THEN EXIT END;
        p := p^.next
    END;
    RETURN  p = NIL
END  Inaccessible;

PROCEDURE GetSeq(VAR s: ARRAY OF BYTE; i: INTEGER; seq: Sequence):
BOOLEAN;
    VAR    p, q: Sequence;
           j:    CARDINAL;
BEGIN
    IF Inaccessible(seq) THEN RETURN FALSE END;
    SearchForEntry(i, seq, p, q);
    IF (p = NIL) OR (p^.entry > i) THEN RETURN  FALSE END;
    IF p^.size # HIGH(s) + 1 THEN RETURN FALSE END;
    FOR j := 0 TO HIGH(s) DO
        s[j] := p^.data^[j]
    END;
    RETURN TRUE
END  GetSeq;

PROCEDURE PutSeq(s: ARRAY OF BYTE; i: INTEGER; VAR seq: Sequence):
                BOOLEAN;
    VAR   p, q: Sequence;
BEGIN
    IF Inaccessible(seq) THEN RETURN FALSE END;
    SearchForEntry(i, seq, p, q);
    IF (p = NIL) OR (p^.entry <> i) THEN
        MakeNewEntry(i, seq, p, q)
    ELSE
        DEALLOCATE(p^.data, p^.size)
    END;
    StoreEntry(s, i, seq, p);
    RETURN TRUE
END  PutSeq;

PROCEDURE Include(s: ARRAY OF BYTE;  i: INTEGER; VAR seq: Sequence):
                BOOLEAN;
    VAR   p, q: Sequence;
BEGIN
    IF Inaccessible(seq) THEN RETURN FALSE END;
    SearchForEntry(i, seq, p, q);
    IF (p # NIL) AND (p^.entry = i) THEN
        RETURN FALSE
    ELSE
        MakeNewEntry(i, seq, p, q);
        StoreEntry(s, i, seq, p)
        RETURN TRUE
    END;
END  Include;

PROCEDURE Delete(i: INTEGER; VAR seq: Sequence): BOOLEAN;
    VAR   p, q: Sequence;
          j:    CARDINAL;
BEGIN
    IF Inaccessible(seq) THEN RETURN FALSE END;
    SearchForEntry(i, seq, p, q);
    IF (p = NIL) OR (p^.entry > i) THEN
        RETURN  FALSE
    END;
    IF q = NIL THEN
        seq^.next := p^.next
    ELSE
        q^.next := p^.next
    END;
    IF seq^.entry = i THEN
        IF q = NIL THEN
            seq^.entry := - MAX(INTEGER)
        ELSE
            seq^.entry := q^.entry
        END
    END;
    IF seq^.size = i THEN
        IF p^.next = NIL THEN
            seq^.size := + MAX(INTEGER)
        ELSE
            seq^.size := p^.next^.entry
        END
    END;
    DEALLOCATE(p^.data, p^.size);
    DISPOSE(p);
    RETURN TRUE
END  Delete;

PROCEDURE Highest(seq: Sequence): INTEGER;
BEGIN
    IF Inaccessible(seq) THEN RETURN - MAX(INTEGER) END;
    RETURN seq^.entry
END  Highest;

PROCEDURE Lowest(seq: Sequence): INTEGER;
BEGIN
    IF Inaccessible(seq) THEN RETURN + MAX(INTEGER) END;
    RETURN seq^.size
END  Lowest;

PROCEDURE  Deaccess(VAR seq: Sequence);
    VAR   p, q: Sequence;
BEGIN
    p := SeqHead;    q := NIL;
    LOOP
        IF p = NIL THEN EXIT END;
        IF p^.entry = INTEGER(CARDINAL(seq)) THEN EXIT END;
        q := p;     p := p^.next
    END;
    IF p # NIL THEN
        IF q = NIL THEN
            SeqHead := p^.next
```

(Listing 3 continued on next page)

(Listing 3 continued from previous page)

```
        ELSE
            q^.next := p^.next
        END;
        DEALLOCATE(p^.data, p^.size);
        p := seq^.next;
        WHILE p # NIL DO
            q := p;
            p := q^.next;
            DEALLOCATE(q^.data, q^.size);
            DISPOSE(q)
        END;
        DISPOSE(seq)
    END
END  Deaccess;

BEGIN
    SeqHead := NIL
END SeqBuffer.
```

'GetSeq' checks to see if the desired entry is in the Sequence, and is the same size as the 'item' parameter being passed to it. If it passes all these tests, the bytes are copied to the indicated 'item' one byte at a time.

'PutSeq' also checks to see if the indicated entry is already in the Sequence. If it is, its data is removed by calling DEALLOCATE, otherwise a new Record is created. In either case, the item is stored in the resulting free Record by calling StoreEntry.

'Include' also checks to see if the entry is already in the Sequence, however if it is, the procedure returns an error indication (i.e. FALSE).

It may seem that deleting an item should be simple, but a glance at 'Delete' shows that this is not so. The major problem here is to make sure that the current highest and lowest values for the Sequence are maintained properly. For instance, if the item with the highest entry number is deleted, the current highest value must be updated to equal the next highest entry in use.

The 'Highest' and 'Lowest' procedures merely return the corresponding value currently maintained for the Sequence. The exception is an Inaccessible Sequence. In this case, the unique values used for an empty Sequence are returned. Supposedly an invalid Sequence is empty.

The purpose of the 'Deaccess' procedure is to reclaim all of the dynamic memory that was acquired for the Sequence. This includes All of the 'item' arrays, the entry Records in the Sequence, and the main Sequence in SeqHead itself.

Modula 2 allows each module to contain some initialization code. The initializing code is included at the end of the module. In this case it is a single statement that makes SeqHead point to nothing (i.e. NIL). This means that its linked list is empty. The Modula 2 linker will create code that executes all of the initializing code before it starts executing the main program.

StackADT

We've just examined the most complex module. SeqBuffer can be used to build several other abstract data types (ADT's) such as stacks and queues. Listing 4 shows the DEFINITION MODULE for the stackADT.

A TYPE definition is given for the 'stack.' Instead of being opaque you will notice that it is visibly defined as a Sequence. This is how we build upon earlier modules.

Just as there was a need to access and deaccess a Sequence, the stackADT has procedures to 'Define' and 'Destroy' a stack. A stack can be cleared at any time by calling 'MakeEmpty'. It is highly recommended that stacks be cleared initially after being defining.

The stackADT defines the normal stack operations of 'Push' and 'Pop'. You will notice that this module also returns Boolean values to report error conditions such as popping data off an empty stack (or pushing it onto a full stack).

There is also a procedure defined that allows you to check to see if a stack is 'Empty.'

The IMPLEMENTATION MODULE for the stackADT is given in Listing 5. You will notice that it's all done with Sequences. Define calls Access to obtain a Sequence which masquerades as a stack. The Highest procedure imported from the SeqBuffer is used to control the push and pop operations. Items are pushed into entry Highest(stk)+1 using the PutSeq procedure. Likewise, items are popped off the stack by calling GetSeq to extract the Highest entry and then calling Delete to remove the entry from the Sequence/stack. The unique values assigned to Highest and Lowest for an empty Sequence are used to determine when a stack is empty. The MakeEmpty procedure clears a stack by repeatedly deleting the Lowest entry until the stack is empty. The stack is destroyed by first clearing itself using MakeEmpty and then calling the Deaccess procedure from SeqBuffer.

Convert

The FTL Modula 2 compiler offers a module named Conversions which converts numeric information in a computer into strings of digits (such as "1234"). These strings can then be output to the terminal so that we humans can read the numbers. I found that this module did not go far enough for my needs. For one thing, I wanted to also be able to convert strings of digits into numeric values. Therefore I developed the 'Convert' module.

The DEFINITION MODULE for Convert is shown in Listing 6. With this module it is possible to represent numeric information as a number in any radix from 2 (binary) to 16 (hexadecimal). This is primarily done by the procedure 'NumToStr.' Decimal numbers are more easily done by 'IntToStr' and 'CardToStr.' (In Modula 2, integers are signed, while cardinals are positive only.) This module also has procedures to convert from strings of digits in any radix into numeric data.

Listing 7 is the IMPLEMENTATION MODULE for Convert. To simplify the conversion to strings, two procedures are imported from the original Conversions module. Why reinvent the wheel? I did find it necessary to left justify the output from the Conversions

Listing 4

```
DEFINITION MODULE stackADT;

(*            David L. Clarke        (revised)  11 June 1990           *)

FROM  SeqBuffer  IMPORT  Sequence;     (* inherit type from sequence      *)
FROM  SYSTEM     IMPORT  BYTE;         (* used to 'match' any type        *)

TYPE       stack = Sequence;           (* the 'stack' data type itself    *)

PROCEDURE  Define                      (* define or initialise a stack    *)
           (VAR    q: stack)           (* the stack variable being defined *)
                    : BOOLEAN;         (* FALSE if no stack is available  *)
                                       (* MakeEmpty should be called next *)

PROCEDURE  MakeEmpty                   (* remove all elements from stack  *)
           (VAR    q: stack)           (* stack to be emptied             *)
                    : BOOLEAN;         (* FALSE if stack is invalid       *)

PROCEDURE  Push                        (* push element onto top of stack  *)
           (VAR    q: stack;           (* stack that receives the data    *)
                elem: ARRAY OF BYTE)   (* the element to be inserted      *)
                    : BOOLEAN;         (* FALSE if stack is full          *)

PROCEDURE  Pop                         (* pop element from top of stack   *)
           (VAR    q: stack;           (* stack that supplies the data    *)
            VAR elem: ARRAY OF BYTE)   (* where the element gets stored   *)
                    : BOOLEAN;         (* FALSE if stack is empty         *)

PROCEDURE  Empty                       (* test the status of a stack      *)
           (       q: stack)           (* stack to be tested              *)
                    : BOOLEAN;         (* TRUE if the stack is empty      *)

PROCEDURE  Destroy                     (* send stack to Never-never Land  *)
           (VAR    q: stack)           (* the unfortunate victim          *)
                    : BOOLEAN;         (* FALSE if stack is invalid       *)

END stackADT.
```

Listing 5

```
IMPLEMENTATION MODULE stackADT;

(*           David L. Clarke         (revised)    11 June 1990          *)

   FROM  SeqBuffer IMPORT  Access, Inaccessible, Deaccess,
                           GetSeq, PutSeq, Delete, Lowest, Highest;

   VAR     i:  INTEGER;
       done:  BOOLEAN;

   PROCEDURE  Define(VAR stk: stack): BOOLEAN;
   BEGIN
       stk := Access("stackhead");
       RETURN NOT Inaccessible(stk)
   END  Define;

   PROCEDURE  MakeEmpty(VAR stk: stack): BOOLEAN;
   BEGIN
       done := NOT Inaccessible(stk);
       WHILE done AND NOT Empty(stk) DO
           done := Delete(Lowest(stk), stk)
       END;
       RETURN done
   END  MakeEmpty;

   PROCEDURE  Push(VAR stk: stack; elem: ARRAY OF BYTE): BOOLEAN;
   BEGIN
       i := Highest(stk);
       IF i < 0 THEN i := 0 END;
       RETURN PutSeq(elem, i+1, stk)
   END  Push;

   PROCEDURE  Pop(VAR stk: stack; VAR elem: ARRAY OF BYTE): BOOLEAN;
   BEGIN
       done :=  GetSeq(elem, Highest(stk), stk);
       RETURN done AND Delete(Highest(stk), stk)
   END  Pop;

   PROCEDURE  Empty(stk: stack): BOOLEAN;
   BEGIN
       RETURN Highest(stk) < Lowest(stk)
   END  Empty;

   PROCEDURE  Destroy(VAR stk: stack): BOOLEAN;
   BEGIN
       done := MakeEmpty(stk);
       Deaccess(stk);
       RETURN done
   END  Destroy;

END stackADT.
```

Listing 6

```
DEFINITION MODULE Convert;

(*            David L. Clarke        (revised)    11 June 1990              *)

(* convert INTEGERs, CARDINALs, or numbers in some base radix to strings    *)

PROCEDURE IntToStr (      int      : INTEGER;          (* INTEGER to convert   *)
                      VAR str      : ARRAY OF CHAR;    (* destination string   *)
                          width    : CARDINAL;         (* (min) characters     *)
                      VAR success  : BOOLEAN);         (* TRUE if converted    *)

PROCEDURE CardToStr(      card     : CARDINAL;         (* CARDINAL to convert  *)
                      VAR str      : ARRAY OF CHAR;    (* destination string   *)
                          width    : CARDINAL;         (* (min) characters     *)
                      VAR success  : BOOLEAN);         (* TRUE if converted    *)

PROCEDURE NumToStr (      num      : CARDINAL;         (* number to convert    *)
                      VAR str      : ARRAY OF CHAR;    (* destination string   *)
                          base     : CARDINAL;         (* base/radix [2..16]   *)
                          width    : CARDINAL;         (* (min) characters     *)
                      VAR success  : BOOLEAN);         (* TRUE if converted    *)

(* convert strings to INTEGERs, CARDINALs, or numbers in some base radix    *)

PROCEDURE StrToInt (      str      : ARRAY OF CHAR;    (* string to convert    *)
                      VAR int      : INTEGER;          (* target INTEGER       *)
                      VAR success  : BOOLEAN);         (* TRUE if converted    *)

PROCEDURE StrToCard(      str      : ARRAY OF CHAR;    (* string to convert    *)
                      VAR card     : CARDINAL;         (* target CARDINAL      *)
                      VAR success  : BOOLEAN);         (* TRUE if converted    *)

PROCEDURE StrToNum (      str      : ARRAY OF CHAR;    (* string to convert    *)
                      VAR num      : CARDINAL;         (* target CARDINAL      *)
                          base     : CARDINAL;         (* base/radix [2..16]   *)
                      VAR success  : BOOLEAN);         (* TRUE if converted    *)

END Convert.
```

procedures. This required the creation of the internal LeftJust procedure. Likewise I had to write the routines that translate from strings to numeric data. These are loosely based on code that was used in the InOut module supplied with the FTL compiler. A string search was added to assist in converting individual digit characters (including the Hex digits of A - F) into numeric values.

Calc—An RPN Calculator

I will close this article with a practical example that makes use of the modules developed in the earlier material. The goal is to create a Reverse Polish Notation (RPN) calculator. One of the basic components of an RPN calculator is a stack. This we already have. We also need a way to convert strings of digits to numbers and numbers to strings of digits. This we have also.

Besides numbers, we also have to input operations such as '+' for addition, '-' for subtraction, and the like. We will also input certain letters to provide the calculator with specific commands such as 'C' for clear. Since this calculator will be used in a programming environment, it seems reasonable to allow it to operate in decimal, hex, octal, and even binary. We need an easy way to switch between modes and to show what our current mode is. These commands are summarized as follows:

+, -, *, /, %	Addition, subtraction, multiplication, division, and 'mod' respectively.
B, D, H, O	Enter binary, decimal, hex, or octal modes respectively.
C	Clear the display.
E	Exit the program.

Two of the commands ('B' and 'D') are also hex digits. In order to avoid confusion, all numeric data, even hex numbers, must begin with a digit between 0 and 9. If a 'B' or 'D' command is placed after a number it must be separated from the number by a space.

The program listing given in Listing 8 shows that the calculator operates quite simply. It reads in a line from the terminal keyboard. It then parses the line for strings of digits and operators. Normally they are separated by spaces (or carriage returns) but in some cases the space is not required, depending on whether the result is ambiguous or not. (Refer to the discussion on 'B' and 'D' above.) As strings of digits are encountered, they are converted into numbers and pushed onto the stack. As arithmetic operators are detected, the top two numbers are popped off the stack, the operation is performed on them, and the result is pushed back onto the stack. Some of the other operators will modify the radix mode. This controls the way strings are converted to and from numbers. Another operator (i.e. 'C') will clear the stack by calling MakeEmpty. The final operator, 'E,' will cause an exit from the program.

As each numeric string or operator is handled, the current value at the top of the stack is displayed. The number is converted into a string representing the number in the current radix or mode. The string is then written to the terminal followed by a space and a single character that indicates the current radix mode.

Reverse Polish Notation is a method of expressing mathematical equations in an unambiguous way without parentheses.

Basically, everything is expressed as triplets consisting of two operands and one operation. Now it gets a bit confusing. Either operand may in turn be expressed as a triplet. A simple equation like

```
(a + b) * (a - b)
```

Listing 7

```
IMPLEMENTATION MODULE Convert;

(*          David L. Clarke        (revised)   11 June 1990          *)

FROM Conversions IMPORT CardToString, IntToString;

    VAR
        digit: ARRAY[0..15] OF CHAR;

    PROCEDURE max(x, y: CARDINAL): CARDINAL;
    BEGIN
        IF x > y THEN RETURN x ELSE RETURN y END
    END max;


    PROCEDURE LeftJust( used: CARDINAL;  VAR     str: ARRAY OF CHAR;
                       width: CARDINAL;  VAR success: BOOLEAN);
        VAR i, j: CARDINAL;
    BEGIN
        j := max(width, used);
        IF j > HIGH(str)+1 THEN
            success := FALSE
        ELSE
            FOR i := HIGH(str)+1 - j TO HIGH(str) DO
                str[i + j - (HIGH(str)+1)] := str[i]
            END;
            IF j <= HIGH(str) THEN str[j] := 0c END;
            success := TRUE
        END
    END LeftJust;


    PROCEDURE IntToStr(  int: INTEGER;   VAR     str: ARRAY OF CHAR;
                       width: CARDINAL;  VAR success: BOOLEAN);
        VAR used:  INTEGER;
    BEGIN
        IntToString(int, 10, str, used);
        LeftJust(used, str, width, success)
    END IntToStr;


    PROCEDURE CardToStr(card: CARDINAL;  VAR     str: ARRAY OF CHAR;
                       width: CARDINAL;  VAR success: BOOLEAN);
        VAR used:  INTEGER;
    BEGIN
        CardToString(card, 10, str, used);
        LeftJust(used, str, width, success)
    END CardToStr;


    PROCEDURE NumToStr(  num: CARDINAL;  VAR     str: ARRAY OF CHAR;
                        base: CARDINAL;
                       width: CARDINAL;  VAR success: BOOLEAN);
        VAR used:  INTEGER;
    BEGIN
        IF (base < 2) OR (base > 16) THEN
            success := FALSE
        ELSE
            CardToString(num, base, str, used);
            LeftJust(used, str, width, success)
        END
    END NumToStr;


    PROCEDURE StrToInt(  str: ARRAY OF CHAR;  VAR  int: INTEGER;
                                          VAR success: BOOLEAN);
        VAR i, j: CARDINAL;
            neg:  BOOLEAN;
    BEGIN
        i := 0;     int := 0;      success := FALSE;       neg :=
FALSE;
        WHILE (i < HIGH(str)) AND (str[i] <= ' ') DO INC(i) END;
        IF str[i] = '-' THEN neg := TRUE END;
        IF (str[i] = '+') OR (str[i] = '-') THEN INC(i) END;
        success := TRUE;
        LOOP
            IF (i > HIGH(str)) OR (str[i] <= ' ') THEN EXIT END;
            IF (str[i] < '0') OR (str[i] > '9') THEN
                success := FALSE;  EXIT
            END;
            int := int * 10 + ORD(str[i]) - ORD('0');
            INC(i)
        END;
        IF neg THEN int := - int END
    END StrToInt;


    PROCEDURE StrToCard( str: ARRAY OF CHAR;  VAR card: CARDINAL;
                                          VAR success: BOOLEAN);
    BEGIN
        StrToNum(str, card, 10, success);
    END StrToCard;


    PROCEDURE StrToNum(  str: ARRAY OF CHAR;  VAR  num: CARDINAL;
                        base: CARDINAL;    VAR success: BOOLEAN);
        VAR i, j: CARDINAL;
    BEGIN
        i := 0;     num := 0;      success := FALSE;
        WHILE (i < HIGH(str)) AND (str[i] <= ' ') DO INC(i) END;
        success := TRUE;
        LOOP
            IF (i > HIGH(str)) OR (str[i] <= ' ') THEN EXIT END;
            j := 0;
            LOOP
                IF CAP(str[i]) = digit[j] THEN EXIT END;
                INC(j);
                IF (j = base) OR (j = 16) THEN EXIT END
            END;
            IF (j = base) OR (j = 16) THEN
                success := FALSE;   EXIT
            END;
            num := num * base + j;
            INC(i)
        END
    END StrToNum;

BEGIN
    digit := "0123456789ABCDEF";
END Convert.
```

is expressed in RPN as

```
a b + a b - *
```

where “a b +” is one triplet corresponding to “(a + b)”. Likewise “a b -” is a triplet that corresponds to “(a - b)”. These two triplets are the operands for the third triplet whose operation is “*”. To solve this equation for a=10 and b=5, and then display the result in hex, octal and binary (in addition to the default decimal), one would type in the following line to ‘calc’.

```
10 5 + 10 5 - * H O B
```

The calculator would list several lines for the partial calculations and then end with the following:

```
     75 d
     4B h
    113 o
1001011 b
```

The terminal letters on each line indicate that the displays are in decimal, hex, octal, and binary respectively.

Conclusion

In this article I have presented several Modula 2 modules that should be useful in future programs. I have also demonstrated their utility in the creation of a calculator program. Hopefully this may have whetted your appetite for more information on the language. In my next article I intend to make the ZCPR connection. A special module will be introduced that allows a Modula 2 programmer access to the Z3 environment. Tune in, it should be interesting. ●

Listing 8

```
MODULE calc;

(*        David L. Clarke        (revised)   11 June 1990         *)

(*  This is an RPN calculator that works in decimal, hexadecimal,  *)
(*  octal, or binary mode.  Use D, H, O, and B commands to switch  *)
(*  to the desired mode respectively.  Use C to clear the calcu-   *)
(*  lator.  Use E to exit from the calculator.  The arithmetic     *)
(*  operations of add, subtract, multiply and divide are selected  *)
(*  by the +, -, *, and / keys respectively.  In addition the mod  *)
(*  operation may is selected by using the % key (this symbol was  *)
(*  borrowed from the C language).                                 *)

FROM Convert   IMPORT StrToNum, NumToStr, IntToStr;
FROM InOut     IMPORT Done, WriteString, WriteLn, WriteInt, Write,
                      AlwaysBuffer, ReadInt, Read;
FROM SeqBuffer IMPORT Highest, GetSeq;
FROM stackADT  IMPORT stack, Define, MakeEmpty, Push, Pop, Empty,
                      Destroy;

    TYPE
        Symbol = (null, oper, number);

    VAR
        stk:        stack;
        arg1, arg2: INTEGER;
        value:      CARDINAL;
        base:       INTEGER;
        op:         CHAR;
        ok:         BOOLEAN;
        read_buff:  CHAR;
        token:      ARRAY [0..20] OF CHAR;
        sym:        Symbol;

    PROCEDURE ReadCh(VAR ch: CHAR);
    BEGIN
        IF read_buff # 0c THEN
            ch := read_buff;
            read_buff := 0c
        ELSE
            Read(ch)
        END
    END ReadCh;

    PROCEDURE ReadAgain(ch: CHAR);
    BEGIN
        read_buff := ch
    END ReadAgain;


    PROCEDURE read_token(VAR token: ARRAY OF CHAR;
                         VAR sym:   Symbol);
        VAR
            ch: CHAR;
            i:  CARDINAL;
    BEGIN
        REPEAT
            REPEAT ReadCh(ch) UNTIL ch > ' ';     ch := CAP(ch);
            CASE ch OF
                '+', '-', '*', '/', '%',
                'B', 'C', 'D', 'E', 'H', 'O':
                    token[0] := ch;    token[1] := 0c;
                    sym := oper
            |   '0' .. '9':          (* even hex must start with 0..9 *)
                    i := 0;
                    REPEAT
                        IF i <= HIGH(token) THEN
                            token[i] := ch;
                            INC(i)
                        END;
                        ReadCh(ch);    ch := CAP(ch)
                    UNTIL NOT ((ch >= '0') AND (ch <= '9')
                           OR  (ch >= 'A') AND (ch <= 'F'));
                    ReadAgain(ch);
                    IF i <= HIGH(token) THEN token[i] := 0c END;
                    sym := number
            ELSE
                    WriteString("Bad entry");   WriteLn;
                    sym := null
            END  (* case *)
        UNTIL sym # null
    END read_token;


    PROCEDURE pop_args(VAR arg1, arg2: INTEGER): BOOLEAN;
    BEGIN
        IF Pop(stk, arg2) AND Pop(stk, arg1) THEN
            RETURN TRUE
        ELSE
            WriteString("Stack underflow");  WriteLn;
            RETURN FALSE
        END
    END pop_args;

BEGIN
    read_buff := 0c;    AlwaysBuffer := TRUE;    base := 10;
    IF NOT (Define(stk) AND MakeEmpty(stk)) THEN
        WriteString("Cannot define stack");   WriteLn
    ELSE
        LOOP
            read_token(token, sym);
            IF sym = number THEN
                StrToNum(token, value, base, ok);
                IF NOT ok THEN
                    WriteString("Bad number");      WriteLn
                END;
                ok := Push(stk, value);
                IF NOT ok THEN
                    WriteString("Stack overflow");  WriteLn
                END
            ELSE
                CASE token[0] OF
                    '+':  IF pop_args(arg1, arg2) THEN
                              arg1 := arg1 + arg2;
                              ok := Push(stk, arg1)
                          END
                |   '-':  IF pop_args(arg1, arg2) THEN
                              arg1 := arg1 - arg2;
                              ok := Push(stk, arg1)
                          END
                |   '*':  IF pop_args(arg1, arg2) THEN
                              arg1 := arg1 * arg2;
                              ok := Push(stk, arg1)
                          END
                |   '/':  IF pop_args(arg1, arg2) THEN
                              arg1 := arg1 DIV arg2;
                              ok := Push(stk, arg1)
                          END
                |   '%':  IF pop_args(arg1, arg2) THEN
                              arg1 := arg1 MOD arg2;
                              ok := Push(stk, arg1)
                          END
                |   'B':  base := 2
                |   'C':  ok := MakeEmpty(stk)
                |   'D':  base := 10
                |   'E':  EXIT (* from loop *)
                |   'H':  base := 16
                |   'O':  base := 8
                END
            END;
            IF NOT Empty(stk) THEN
                ok := GetSeq(arg1, Highest(stk), stk);    WriteLn;
                IF base = 10 THEN
                    IntToStr(arg1, token,       6, ok)
                ELSE
                    NumToStr(arg1, token, base, 6, ok)
                END;
                WriteString(token)
            ELSE                             WriteLn;
                NumToStr(0, token, base, 6, ok);
                WriteString(token)
            END;                             Write(' ');
            CASE base OF
                2: Write('b')      |   8: Write('o')
            |  10: Write('d')      |  16: Write('h')
            END;  WriteLn
        END (* loop *)
    END;
    IF NOT Destroy(stk) THEN
        WriteString("Cannot destroy stack");  WriteLn
    END
END calc.
```

The Z-System Corner

by Jay Sage

Although I have not yet finished the treatment of MEX, I am going to start a new subject this time: the ZMATE macro text editor. During the past two months I have been working on a number of code patches to MEX-Plus to fix some problems and to add some new features that I wanted or needed. That work is not complete, so I have decided to hold off on a MEX update until next time. As usual, I do have a few miscellaneous items to bring to your attention.

Pieces of Eight

First, I would like to put in a plug for the "Pieces of Eight" magazine (POE) from the Connecticut CP/M Users' Group (CCP/M). CCP/M recently decided to begin addressing a national audience and not just their local members. Even if you cannot attend their meetings, the subscription to POE that your $15 annual dues brings you is alone worth the price.

POE is a very nice complement to TCJ. I don't think I will offend CCP/M by saying that their magazine is far less serious than this one. There is some solid technical content, but the emphasis is definitely on the human side of computing. It is really fun to read, and not just by us computer nuts but by our entire families as well.

The July, 1990, issue has a feature article on the Trenton Computer Festival held in April. On the cover is a picture taken there showing me, Bridger Mitchell, Al Hawley, and Cam Cotrill. (In case you might be questioning my motives, their flattering me by putting my picture on the cover provided only a fraction of the inspiration for this plug!)

Inside are more pictures: Rob Friefeld (LSH, SALIAS), Carson Wilson (ZDE, ZSDOS), Hal Bower (ZSDOS), Bruce Morgen (MEX+2Z and lots of program patches), Howard Goldstein (our alpha tester and bug catcher and fixer extraordinaire), and quite a few others. As you can see, Trenton drew Z-Team members and enthusiasts from all over the country! If you want to learn more about the festival, sign up for POE. Send dues to Tom Veile, 26 Slater Ave., Norwich, CT 06360.

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER point-and-shoot shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (MABOS on PC-Pursuit, pw=DDT). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Jay is now also the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail or chatted with live at the Wednesday real-time conferences (10pm Eastern time).

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog computation to solve problems in signal, image, and information processing. His recent interests include artificial neural networks and superconducting electronics. He can be reached at work via Internet as SAGE@LL.LL.MIT.EDU.

A Patch for The Word Plus

Some time ago I published here a set of ARUNZ aliases for automating the use of The Word Plus spell checker. Well, Richard Swift liked them just fine, but it then annoyed him that he still had to hit a carriage return to get past TW's prompt about whether the configuration was correct. He wanted TW to get right to work.

At first I didn't really see why he was making such a fuss about such a little thing. Then it began to eat at me, too. This one little thing was standing in the way of complete automation.

Well, it took a good bit of poking around in the TW.COM code, but in the end it was quite easy to patch around this annoying prompt. First I located where the code that put up the prompt began, and then I found where things picked up again after it. A simple jump instruction at the beginning to skip over it should do the trick, I thought.

Unfortunately, it was not quite that simple. As Bruce Morgen had described earlier in an issue of his NAOG newsletter, the programs in The Word Plus suite perform some simple internal checking to make sure the file is not corrupted and has loaded successfully. Nice of those folks, but after I put in my patch, the code looked corrupted. I could have figured out the new checksum value and stuck it into the testing code, but it was easier just to bypass the checking entirely.

At first I put the changes into a patch file that would be overlaid onto the original code. Then, however, I decided that there was no real need to make the change permanently. When running TW manually, one would probably want the prompt to appear so that one would have the option of changing the setup. So, my solution was the old GET/POKE/GO technique introduced by Bruce Morgen (boy does that name keep coming up!).

My original ARUNZ alias had a command of the form

```
tw:tw <file> <dictionary>
```

I just replaced that by

```
/TWPAT <file> <dictionary>
```

and wrote the new alias TWPAT with the command lines

```
get 100 tw:tw.com   load TW.COM
poke 103 c3 3b 01   patch to jump over code test
poke 395 c3 2a 04   patch to jump over prompt
go $*                             run the patched code
```

Now I could invoke the patched TW whenever I wanted by using the command TWPAT instead.

The ZMATE Text Editor

Now for the main topic of this column, the first in a series of articles on ZMATE. This one will be just an introduction and will cover only its design philosophy and mode of operation. Next time I will start to describe its language in detail.

Interpreters and Compilers

A casual user would classify ZMATE as an application program, and more precisely as a text editor or wordprocessor. In its soul, however, it is really a high-level programming language. In some ways it is similar to the familiar BASIC interpreter.

Like almost all the programming languages most people work with, BASIC is oriented toward numerical computation. For example, at the system prompt one can enter a command such as

```
print ( n1 + n2 ) * n3
```

BASIC will then retrieve the values associated with the variables N1, N2, and N3, substitute them into the mathematical expression, evaluate the expression, and print the result to the screen.

BASIC also allows one to write programs comprising a series of numbered statements such as:

```
100 n1 = 10
110 n2 = 5
120 n3 = 3
130 print ( n1 + n2 ) * n3
```

When the immediate command "RUN" is entered, the entire sequence of commands is carried out, and the number 45 appears on the screen.

One could write a program to do the same thing using assembly language, the native language of a computer. However, a high-level language like BASIC makes it far easier to generate the required instructions. This is especially true when we are dealing with floating point numbers, or when we are using array variables or advanced mathematical (trig and log) functions.

When the BASIC interpreter we described above is told to "RUN", it processes the program statements one at a time. First it analyzes a statement to determine the procedures required to perform the specified function. Then it calls routines that execute those procedures. This means that when a BASIC statement appears in a loop, the analysis has to be repeated each time the statement is executed.

A compiler provides an alternative approach. The compiler can be thought of as an automatic assembly language program writer. You write your program using the commands of the high-level language, and then the compiler converts them into an assembly language program for you.

Some compilers generate actual assembly language source code that you then have to assemble. The PASCAL Z compiler, for example, worked this way. This approach makes program development slower but allows you to fine-tune the code if you so desire. Other compilers, such as Turbo Pascal, generate only the machine code (COM) files. Some compilers, such as BDS C, follow a two-step process, but the intermediate code is not standard assembly code.

A compiler, as you might guess, has the advantage of execution speed, since the high-level language statements have to be analyzed and converted into machine code only once, even when they are executed repeatedly in a loop. Also, more complex programs that need more working memory can be accommodated, since the code that figures out how to process the high-level language statements does not have to be in memory when the final program is run.

On the other hand, an interpreter offers many advantages that may make it well worth giving up some speed. Programs are much easier to develop with an interpreter for several reasons. First, you can execute them immediately, without having to go through the extra step of compilation (and possibly assembly and linkage) before execution. Second, the programs can be run line by line, and you can watch what is happening and catch errors more easily.

There are also some things that an interpreter can do that a compiler generally cannot. For example, suppose you are working with an array variable (a variable that holds a collection of values, not just a single value). With a compiler, you would have to specify the size—or at least a maximum size—of the array at the time the program is compiled so that the compiler can allocate enough memory for it. With an interpreter, this is not necessary. It does not have to allocate the memory until the variable is first referenced. As a result, it is quite acceptable for its size to be determined by computations performed earlier in the program.

ZMATE as Interpreter

ZMATE is, in a way, like the BASIC interpreter, except that its intrinsic high-level language functions (we will call these 'primitives') are aimed at text processing rather than number processing. Just as BASIC has some text-processing primitives (e.g., string variables and functions), so ZMATE has some numerical functions, but it is the text-manipulation primitives that are emphasized and richly developed.

If your past experience has been confined to the usual programming languages—BASIC, FORTRAN, PASCAL, C, etc.—you probably have trouble picturing what a text-processing language would look like. Here are some examples to help convey the concept.

While most variables in BASIC contain either single numbers or arrays of numbers, ZMATE has 'variables' called buffers that contain pieces of text. Primitives allow reading disk files into these buffers or writing text from the buffers out to files.

Each buffer has two pointers. One is called the cursor. It is where most ZMATE primitives perform their operation. The other pointer is called a tag, and together with the cursor it defines a block of text for some block-operation primitives.

A whole set of ZMATE primitives deals with cursor motion. The cursor can be moved forward and backward in the buffer by units of characters, words, paragraphs, or the whole buffer. For example, you can tell the cursor to back up by three words or go forward two paragraphs.

This highlights the difference between a number-processing and a text processing language. BASIC supports string variables that can contain a line of text, but it does not know about words and paragraphs. The user would have to write complex code to deal with these text concepts. As a text-processing language, ZMATE provides the code for this as part of the language primitives.

Other ZMATE primitives search for strings and compare strings or characters. Text can be inserted and deleted. Blocks of text can be moved between buffers for cutting and pasting operations. All the usual control primitives are provided to allow testing, conditional operations, and looping.

There are also special facilities for handling text formatting and text input from the keyboard. Soft carriage returns can be placed into text automatically, and various kinds of indentation and margin control are provided. These functions make it easy to write a wordprocessor in the ZMATE language.

How the ZMATE Language Is Used

In our examples above, we saw that a BASIC statement can be entered for immediate execution. ZMATE, too, allows this. We also saw that BASIC programs containing a sequence of statements can be prepared for later execution. The same is true of ZMATE. In fact, ZMATE can have a number of programs loaded and ready for execution at the same time, and one program can call another as a subroutine.

ZMATE allows its language to be used in one other very special way. Programs that are permanently stored in the ZMATE COM file can be bound to a key or sequence of keys. Then when that key sequence is typed at the keyboard, the program is automatically executed. ZMATE commands executed this way are called "instant commands."

As an example, suppose we write this little ZMATE program:

```
100 put the tag where the cursor is now
110 move the cursor forward one word
120 delete the block (tag-to-cursor)
130 stop
```

[I am using a BASIC-like pseudo-language for this example. The actual ZMATE language, which we will get into next time, is not at all like this.] If we now bind this program to the "^T" (control-T) key, we will have implemented the WordStar delete-word function.

This should give you a sense now of how ZMATE can be used to implement a text editor or wordprocessor. Although ZMATE comes with some standard programs and key bindings, you can change the standard programs, can attach your own new programs, and can change the key bindings. Thus you have extensive control over the way ZMATE works and can add any functions you like to it.

The ZMATE Screen

The normal appearance of the screen while ZMATE is running is shown in Figure 1. In fact, I captured this screen using the BGii 'screen' command while writing this article. I have made a few changes to adapt it to the TCJ format. The real screen is the full width of the terminal, usually 80 characters, and the full length, usually 24 lines. I have reduced both of these sizes.

All but the top three lines are used for the display of text. In the original PMATE, only one buffer could be viewed. With ZMATE, Bridger Mitchell made it possible to look at two buffers or at two sections of one buffer at the same time. By the way, the '<' characters at the ends of some lines in Figure 1 indicate hard carriage returns. The other lines end with soft returns. If one changes the margins, the text instantly readjusts.

At the left of the top line, ZMATE shows the currently logged directory, the file that is open for input, and the file that is open for output. In this case, the output file is a temporary file, TCJ:TCJ46.$$$. When one closes the edit file, the input file will be given a file type of BAK, while the temporary output file name will be changed to the original input file name.

In the center of the top line, two status variables are displayed. The first tells us which buffer is currently being edited (there are 12 of them); the second is a numerical value returned by the last ZMATE command that was performed. That value can convey information to the user or can be used for testing in a program.

At the right edge of the screen, three other status variables are displayed. The position of the cursor is given as a column and line number. The third value tells how much free memory is available for additional text.

The second line in Figure 1 shows the mode status "INSERT MODE". ZMATE can run in three modes: insert, overtype, and command. In command mode, the second line is where the user enters ZMATE program statements for immediate execution. After a command is entered, it is executed by pressing the escape key (ESC).

The most recently entered command remains on the command line and can be executed again by pressing ESC again. Other instant command functions can be executed in between. This gives ZMATE wonderful power. It is one of the things that the author of Vedit—which began, I believe, as a PMATE clone—never understood and is one of the reasons why I have always found Vedit unacceptable as an editor.

Here is an example of how this facility can be used. Suppose we want to change a number of words to upper case. Assuming this is not already defined as a built-in editor function, we write a command line with code that changes all letters of the word containing the cursor to upper case. Then we press ESC, and the current word is converted. Suppose the next word we want to convert is down two lines and over three words from where we are now. Assuming WordStar-like bindings, we could press "^X^X^F^F^F". Then we can press ESC again to convert that word. In a sense, ZMATE commands typed on the command line become bound temporarily as an instant command on the ESC key.

In insert mode, we are effectively running a ZMATE program that asks the user to press keys, which are then inserted into the

```
TCJ: TCJ:TCJ46.WS,TCJ:TCJ46.$$$   buf=T  arg=0          |col = 18
INSERT MODE                                             |line= 204
--------------------------------------------------------|free= 13454
        100 put the tag where the cursor is now<
        110 move the cursor to the next word<
        120 delete the block (tag-to-cursor)<
        130 stop<
<
[I am using a BASIC-like pseudo-language for this example.  The
actual ZMATE language, which we will get to next time, is not at
all like this.]  If we now bind this program to the ^T key, we
will have implemented the WordStar delete-word function.<
<
<
The ZMATE Screen<
<
   The normal appearance of the screen while ZMATE is running is
shown in Fig. 1.  In fact, I captured this screen using the BGii
`screen' command while writing this article.  I have made a few
changes to adapt it to the TCJ format.  The real screen is the
```

Figure 1. This is a snapshot of the ZMATE screen approximately as it appeared while I was writing this column.

text. Overtype mode is the same except that the new characters replace the ones previously under the cursor. In both insert and overtype mode, instant commands operate just as in command mode. That is, key sequence binding are still fully in effect.

Key Bindings

This is a good time to make the role of key bindings more explicit. With ZMATE, one should think of no keys as producing direct input to the editor. All keys have to be bound to some function if they are to have any effect at all.

ZMATE has three sources for the functions that are bound to the keys. One of these comprises functions that produce ASCII characters. Most people would take it for granted that pressing the 'A' key would produce an 'A', but this is not necessarily so in ZMATE. This makes it quite easy to implement a non-standard keyboard layout, such as the Dvorak layout.

The bindings, moreover, are not one-to-one. You can have a number of different key sequences bound to the same function. So, if you want to have two ESC keys, you can bind a second keyboard key to the "produce-an-ESC-character" function as well. And I want to emphasize that these bindings are of sequences of one or more keys (up to some configurable maximum number) to any single function.

The key bindings are defined in a table with the following structure. Each entry, except the last, comprises a byte with a function number followed by the sequence of ASCII key codes bound to that function. The sequences are all exactly the maximum length specified in the configuration. If the defined sequence is shorter than this, null bytes (value 0) are used as filler. The end of the table is indicated by a value of FF hex in the function-number position.

The character-producing functions have numbers from from 1 to 127 inclusive. I am not sure about function 0. Putting a null into text is generally not allowed, as null is used to separate the buffers. If no explicit binding is specified for a single ASCII character in the range 1 to 127, it is by default bound to the function that produces that character. Thus the key sequence 'A' (a single press of the 'A' key) is bound to the "produce-an-A" function if it does not appear in the key binding table.

This direct mapping of ASCII characters is not, as I said above, required. For example, I use the tilde and back-apostrophe as lead-in keys to other sequences (some people would call these keys 'meta' keys). In order to be able to enter these two characters easily into text, I bind the sequence "~~" (two tildes in a row) to the "produce-a-tilde" function and "``" to the "produce-a-back-apostrophe" function.

The second set of functions, numbered from 128 to 191, is implemented in ZMATE's internal code. However, all but a few of them are in fact performed by macro statements in the standard ZMATE language. In PMATE there was no way to modify these; in ZMATE, they have been placed at the end of the code and referenced in a way that allows the overlay configuration patch to redefine these functions freely.

By my count, of the 64 functions of this type, all but 12 are defined by macro program statements. In some cases it is obvious why some are not. For example, there is a function for setting a repeat count that applies to the next command entered. There is also a function that aborts the execution of any macro. These functions would not make sense in the macro language itself.

For some functions it is not so clear why they are not implemented as macros. For example, there is a function to pop from the "garbage stack" the most recently deleted block of text. This is something that cannot presently be done in the command language, but I don't see why it couldn't or shouldn't be.

Then there are several functions for which there exist macro commands that perform the function. Switching to insert, overtype, or command mode are examples. I don't know why they are implemented directly in code rather than in the macro language.

The final set of functions is numbered from 192 to 254. A hexadecimal FF (255 decimal) is used to mark the end of the binding table, so this function number is not allowed. These functions are associated with what is called the "permanent macro area" or PMA in ZMATE.

The PMA is a text block that is permanently stored along with the ZMATE code and can be moved to and from editing buffers. It contains a series of macro definitions, each one introduced by a control-X character followed by the one-character name for the macro and then the program. Functions 192 to 254 correspond to macros whose one-character name is 160 less than the function number, i.e., from space (32) to caret (94). Because the PMA can be edited from within ZMATE, these instant-command functions can be modified quite easily. It might even be possible for one of these macros to be modified by another macro!

Permanent macros are not limited to the names that can be bound to key sequences. The maximum number of permanent macros would be 256 (0 to 255). However, (1) the value 0 is not allowed, (2) upper-case and lower-case letters are equivalent, and (3) not all characters with the high bit set are distinct from the same character without the high bit set (though some are different). In all, by my count there are 160 possible permanent macro names, of which 63, as mentioned earlier, can be bound to keys. The others can be invoked from the command line or from other macros.

Well, this completes the discussion of ZMATE for this time. Next time I will present its command language in detail. •

Animation with Turbo C Ver. 2.0

Part 3: Text in the Graphics Mode

by Clem Pepper

There are many ways in which text contributes to our screen action programs. The first coming to mind is almost certain to be scoring. In part 2 we wrote a simple game, TANK_WAR. Now we will learn how to keep track of the game action with a performance display. For this we need to know how text is provided and printed in the graphics mode.

The graphics library includes a default 8 x 8 bit-mapped font. In addition several stroked fonts are provided. Bit-mapped font characters are defined in a matrix. Stroked fonts are defined by instruction vectors directing the graphics system in their construction. This enables us to magnify the stroked characters while retaining good quality and resolution in their appearance. This will stimulate our creativity in game title and instruction screen designs.

The text mode printing functions, printf(), puts(), etc., do not behave as we would like when in the graphics mode. To our good fortune there is a way to get around much of the limitation.

We begin with an overview of the text functions available from the graphics library. We continue with applications of the text functions in scoring our games. We will then learn how to use printf(), puts(), and other text mode printing functions in our graphics programs.

Next we will see how to display and make use of the library fonts in the design of unique information and title screens for the games we write.

Using outtext() and outtextxy() With Our Graphics

Table 1 summarizes the text functions available to us in the graphics library. As we see, there are not too many, with only two that output text to the screen, outtext() and outtextxy(). We'll gain experience with these in the example programs to follow.

These two functions are severely limited in their capabilities when compared to the text mode printf() family. The primary lack is of any formatting provision within the functions. Formatting is possible by way of sprintf() however.

On the other hand the variety of text fonts, horizontal or vertical printing, and other supporting functions offer features not available in the text mode. So we gain in some respects while losing in another.

It turns out there is a unique requirement for the viewport should there be a need to revise our text. This because we cannot make a revision to existing text printed with either version of outtext by simply writing over it. We end up with a blurry smudge as the new material writes across the original.

We can see this for ourselves by running GRAFTXT1.C (Listing 1). In this program we first print the string "Hi Y'all!", insert a briefdelay, then overwrite it with another: "How Y'do'in?" The second string simply merges with the first creating an unreadable mush.

A solution is to include the text in a viewport. The viewport is cleared whenever new text is to be written at the same location. Note the necessity of setting the clipping variable to zero if text is to be displayed elsewhere on the screen.

Note the use of textwidth() and getmaxx() for text centering. Vertical and horizontal text positioning is available with the function settextjustify(). This could have been used for the centering. Table 2 summarizes the options provided for our text.

Formatting is possible, but in a roundabout way. It requires the use of sprintf() with a character buffer. sprintf() is similar to printf() in its features. The syntax for sprintf() is:

```
int sprintf(char *p_string, const char *format_string,...);
```

Table 1. A summary of the graphic library functions for text output in the graphics mode.

outtext	sends a string to the screen at the current position
outtextxy	sends a string to the screen at the specified position
settextjustify	sets text justification values used by outtext and outtextxy
settextstyle	sets the current text font, style, and character magnification factor
setusercharsize ...	sets width and height ratios for stroked fonts
textheight	returns the height of a string in pixels
textwidth	returns the width of a string in pixels
gettextsettings ...	returns current text font, direction, size, and justification
registerbgifont ...	registers a linked-in or user loaded font

Table 2. Text justification constants for use with settextjustify().

Direction	Constant	Description
Horizontal	LEFT_TEXT	The text left edge is flush with respect to a vertical line drawn through the current position (vertical reference line).
	CENTER_TEXT	The midpoint of the string is aligned with the vertical reference line.
	RIGHT_TEXT	The text right edge is flush with respect to the vertical reference line.
Vertical	BOTTOM_TEXT	The bottom of the lowest character is flush with a horizontal line drawn through the current position (horizontal reference line).
	CENTER_TEXT	The horizontal reference line passes through the center of the characters in the string.
	TOP_TEXT	The top of the highest character is flush with the horizontal reference line.

Listing 1. Illustrating graphics text in a viewport.

```
/* GRAFTXT1.C
** Illustrating viewport text display with outtextxy().
*/
#include <stdio.h>
#include <graphics.h>
#include "grap_enb.h"

int score = 0, score_flg = 1;

/* == Begin program == */
main()
{
int palette = 2;
int tx, twx, i = 10;
char score_buf[80];
  enable_graph(palette);

  /* ** writing over an existing outtextxy() line ** */
    twx = textwidth("Hi Y'all!");
     tx = getmaxx()/2-twx/2;
     outtextxy(tx,20,"Hi Y'all!");
      delay(1000);
       outtextxy(tx,20,"How Y'do'in?");

  /* ** define viewport ** */
   while(i--) {
    if(score_flg == 1)    /* flag is set if score updated */
     setviewport(0,40,319,50,0);             /* restore port */
      clearviewport();              /* clear for score update */
       sprintf(score_buf,"You have %d points.",score);
      outtextxy(10,0,score_buf);  /* display updated score */
     score_flg = 0;  score_up();        /* update the score */
    delay(50);
                  }
     moveto(10,30);  setcolor(2);
     outtext("Press any key to exit.");
  /* ** this message will not print with a 1 for the  ** */
  /* ** clip 0 in setviewport(0,40,319,50,0). Try it! ** */

  /* ** exit the program ** */
     getch(); closegraph(); exit(0);
}

/* == update score == */
score_up()
{
    score += 100; score_flg = 1;
}
```

This function accepts a variable number of arguments, converts their integer values to characters, and stores the characters in a buffer pointed to by *p_string. The primary difference between printf() and sprintf() is that sprintf() sends its output to a buffer. One other distinction is that sprintf() does not respond to the newline character, "\n," when in the graphics mode.

Coding for a typical formatted string is found in the while() loop of Listing 1. In this example we are simply incrementing a mythical score a few times. Note the clearing and re-printing of the entire text with each update.

We will find many applications for outtext() and outtextxy() in our programs. Through their use we can display text vertically as well as horizontally. We can take advantage of the several available fonts and those of our own making. But for scoring our games it is hard to do better than printf(). We will see why that is next.

Using the Text Mode Print Functions With Our Graphics

Ordinarily we cannot use the text mode string functions printf(), puts(), scanf() and others when in the graphics mode. The DOS_UTIL.H, (Listing 2 from "C and the MS-DOS Screen, issue 42), function pos_cur(col,row) makes its use possible. The call to this function places a 2 in register AX (set cursor position). It also enters the column and row values in registers DH and DL. Selecting coordinate values that are a multiple of eight will place the cursor at its text mode location.

Through experiment we find that printf(), puts(), and scanf() perform as in the text mode. There are limitations—only the default font can be used and the text color cannot be set. The highest color in the palette in use becomes the text color. For scoring purposes printf() is the ideal function to use. The default font is compact and readable and the color is a lesser concern.

Program GRAFTXT2.C (Listing 2) illustrates uses of printf(), puts(), and scanf(). puts() duplicates the GRAFTXT1 experiment with outtextxy() in which the second string was simply laid over the first. Observe that no such problem appears with puts(). The program also includes keyboard input with scanf(). There is no equivalent keyboard input function in the graphics library.

It is essential to call cur_pos() before any call to puts() and the other functions. If printf() includes a newline character, (\n), the line will move down. Also the line moves down following a call to puts() as this function always terminates with a newline.

As with GRAFTXT1 the scoring is incremented in a while() loop, lines 32 - 41. Note that a new call is made for restoring the cursor position on each pass through. To see the necessity for this try commenting it out. Also to verify that our computer is truly operating on a row and column basis we request a cursor position report, lines 44 - 46. The position report is followed with a request for keyboard input using scanf(), lines 49 - 52.

Adding Scoring To the TANK_WAR Game

Scoring is best added to a game's source code as a final step. This turns out to be a really easy task when compared to writing the game logic. We will see this from the ease by which scoring is added to TANK_WAR.C. The majority of revisions are made in this module, which we will now address. It will be helpful to have the game listings from the previous article (Part 2 of this series) on your screen as you read through the revisions.

As a first step add #include "dos_util.h" to the #include list. In lieu of this add the pos_cur(col,row) function to the program. With this addition we can take advantage of printf() for game performance display. If you add the utility file to the #include list the function rd_nonasky() should be deleted.

The next step consists of additions to the list of global declarations. Of these, two are flags, shl_fir_flg and tnk_hit_flg. The first flag is set when a shell is fired. The second when a hit on the tank is made. The purpose of the flags is to save program time. There is no point in updating the displayed scores if no change has occurred. So we use a flag only when there is a change to enable access to the updating statements.

The full list of the additions follows:

```
    /* ** scoring additions ** */
extern int shl_fir_flg;
extern int shl_bal;
extern int tnk_hit;
extern int tnk_hit_flg;
extern int score;
extern int pla_hit;
extern char *rank;
```

In this game we begin with the rank of PRIVATE and work up to GENERAL as our hit score accumulates.

Lines 38 and 39 are deleted since we are going to use printf(). These are:

```
    /* ** set viewport for scoring ** */
     /* This will be added later */
```

The line number for scoring insertion is changed to a more favorable location. The code block for maintaining the game performance is inserted following existing line 55, drop_bombs(). This code is as follows:

```
/* ** maintain play scoring ** */
   if(tnk_hit_flg == 1 || shl_fir_flg == 1) {
```

Listing 2. Illustrating use of text mode print functions in the graphics mode.

```
/* GRAFTXT2.C
** A test program for using DOS_UTIL.H text mode functions
** with the graphics mode.
*/
#include <stdio.h>
#include <graphics.h>
#include "grap_enb.h"
#include "dos_util.h"

#define BYE "Press any key to exit."
extern int page, row_no, col_no;
int score_up(void);
unsigned score = 1000;
int score_flg = 1;

/* == Begin program == */
main()
{
int flg_trk = 0, i = 10, palette = 2;
char name[25];
  enable_graph(palette);

  /* ** writing over an existing text using puts() ** */
    col = 10, row = 0; /* page = 0; in dos_util.h */
    pos_cur(col,row,page);
     puts("Hi Y'All!");
      delay(1000);
       pos_cur(col,row,page);
        puts("How Y' do'in?");

  /* ** repeat the score exercise using printf() ** */
     while(i--) {
      if(score_flg == 1) {
         pos_cur(0,3,0); /* OK to omit col, row, page */
        printf("You have %u points.",score);
                          }
       pos_cur(0,4,0);
        printf("flg_trk = %d.",flg_trk++);
         score_flg = 0;  score_up();
          delay(50);
                 }
   /* ** read and display the current cursor position ** */
   /* ** in column and row coordinates.               ** */
    rd_cur_pos(0);
   pos_cur(5,4,0);
  printf("\nrow number = %d, column number is %d.\n\n",\
   row_no,col_no);

  /* ** display scanf() input ability with name query ** */
   printf("Please tell us your first name: ");
    scanf("%s",&name);
     printf("What a great name! %s, I envy you.\n\n",name);
      puts(BYE);

   /* ** exit the program ** */
     getch();  closegraph();  exit(0);
}

/* == update the score == */
score_up()
{
  score += 100; score_flg = 1;
}

```

```
    pos_cur(0,0);
     printf(''TANKS left %d: SHELLS left %d: HITS %d  '',\
      tnk_hit,shl_bal,pla_hit);
    pos_cur(2,1,0);
     printf(''Your SCORE is %d, %s'',score,rank);
      tnk_hit_flg = 0; shl_fir_flg = 0;      }
  if(tnk_hit == 0 || shl_bal == 0) { i = 0; break; }
```

The two spaces following HITS %d are needed to accommodate for changing line length as scores build up. Without the spaces strange numbers make appearance. Note when all the tanks have been bombed or all the shells fired the game is over. At this time we need to return to the text mode, but not to exit the game yet. The code for a performance summary is inserted between the closegraph() of line 79 and the exit(0) following.

```
/* ** closing messages ** */
 printf(''\n\n\n\n\n\n\n\n'');
  if(tnk_hit == 0)
   printf(''        That was your last tank, %s\n'',rank);
 else
   printf(''        That was your last shell, %s\n'',rank);
  if(tnk_hit != 0)
   printf(''        Congratulations on saving %d of your\
25 tanks.\n'',tnk_hit);
   printf(''        You fired %d shells for %d plane hits.\n'',\
    100 - shl_bal,pla_hit);
   printf(''        Your percentage of hits is %d.'',\
    (pla_hit*100)/(100-shl_bal));
```

This completes the revisions to the TANK_WAR.C module. The next changes are made to the BOMBS.C module. The global declarations of lines 13 and 14 were included initially in anticipation of scoring. Three global additions are now required, actually transfers from local to global. The three static declarations of lines 19, 21, and 22 are globals following lines 13 and 14. The new configuration follows:

```
int tnk_hit = 25;
int tnk_hit_flg = 0;
int drp1_flg = 0, drp2_flg = 0, drp3_flg = 0;
int xbmb1, xbmb2, xbmb3;
int ybmb1 = 65, ybmb2 = 69, ybmb3 = 73;

/* == begin program == */
drop_bombs()
{
static int xa = 0, ya = 0;
static int bom1_flg = 1, bom2_flg = 1, bom3_flg = 1;
```

This is all that is needed with this module, as lines 86 and 87 were initially included in anticipation.

The remaining affected module is SHELL.C. The first addition follows line 17 as:

```
extern char *rank;
```

recalling that char *rank was declared in TANK_WAR.C.

One extern is also added for scoring:

```
extern int fir1_flg, fir2_flg, fir3_flg;
```

Lines 21 - 25 were included initially as anticipation. These had to be global as they are used in other modules as well. Existing lines 74, 86, and 98 are:

```
return;     }}
```

Insert a function call, rank_tst(); ahead of return; in each of these lines. Then attach the following function code to the program's end.

```
/* == test for rank promotion, make award ** */
rank_tst()
{
        if(score < 500)  rank = ''PRIVATE    '';
   else if(score < 1000) rank = ''CORPORAL   '';
   else if(score < 2000) rank = ''SERGEANT   '';
   else if(score < 3000) rank = ''LIEUTENANT'';
   else if(score < 4000) rank = ''CAPTAIN    '';
   else if(score < 5000) rank = ''MAJOR      '';
   else if(score < 6000) rank = ''COLONEL    '';
   else if(score < 7000) rank = ''GENERAL    '';
}
```

Applying the Graphics Library Character Fonts

A dictionary definition of font is "A complete set of type of one size and face." The word is derived from the old French fondre, meaning to melt or cast. Luckily for us we can side-step the melting and casting.

The graphics library provides a bit-mapped default font plus four stroked fonts. We just experienced the default through the scoring of TANK_WAR. The four stroked fonts are named Triplex, Small, SansSerif, and Gothic. Their file names are TRIP.CHR, LITT.CHR, SANS.CHR, and GOTH.CHR. Each of the fonts, including the default, can be magnified over a range of

Listing 3. Comparing the five library fonts.

```
/* FONTS.C
** A program illustrating the Turbo C fonts.
*/
#include <stdio.h>
#include <alloc.h>
#include <graphics.h>
#include "grap_enb.h"

/* == Begin program == */
main()
{
int tx, ty = 10, twx, palette = 2;
  /* ** register the fonts ** */
 if(registerbgifont(triplex_font) < 0)   exit(1);
 if(registerbgifont(small_font) < 0)     exit(1);
 if(registerbgifont(sansserif_font) < 0) exit(1);
 if(registerbgifont(gothic_font) < 0)    exit(1);
  /* ** set the graphics mode ** */
  enable_graph(palette);
    setcolor(2);  /* the font color */
  /* ** define the font, direction and size ** */
   settextstyle(TRIPLEX_FONT,HORIZ_DIR,2);
  /* ** dimension the text ** */
     twx = textwidth("Hi Y' All!");
     tx = getmaxx()/2-twx/2;
    outtextxy(tx,ty,"Hi Y' All!");

   settextstyle(SMALL_FONT,HORIZ_DIR,8);
     twx = textwidth("Hi Y' All!");
     tx = getmaxx()/2-twx/2;
      ty += 30;
    outtextxy(tx,ty,"Hi Y' All!");

   settextstyle(SANS_SERIF_FONT,HORIZ_DIR,2);
     twx = textwidth("Hi Y' All!");
     tx = getmaxx()/2-twx/2;
      ty += 30;
    outtextxy(tx,ty,"Hi Y' All!");

   settextstyle(GOTHIC_FONT,HORIZ_DIR,4);
     twx = textwidth("Hi Y' All!");
     tx = getmaxx()/2-twx/2;
      ty += 30;
    outtextxy(tx,ty,"Hi Y' All!");

   settextstyle(DEFAULT_FONT,HORIZ_DIR,3);
     twx = textwidth("Hi Y' All!");
     tx = getmaxx()/2-twx/2;
      ty += 50;
    outtextxy(tx,ty,"Hi Y' All!");
  /* ** return to the text mode and exit ** */
     getch(); closegraph(); exit(0);
}
```

one to ten. These are summarized in Table 3.

While the .CHR files can be used directly it is advantageous to convert them to object (.OBJ) files. This is readily accomplished using the BGIOBJ utility.

Suppose we take a look at the stroked fonts to best learn how to enter them into our programs.

The Stroked Fonts

These differ from the bit-mapped in their construction as lines rather than set bits. The line segments are called "strokes." In general the stroked fonts provide higher quality text. The program FONTS.C (Listing 3) shows us how to make use the fonts. It also gives us a means of comparing the fonts through a sampling of each on our screen. The font characters conform to the ASCII numbering code; i.e., A = 65, etc. We'll see this when we look at the program code later.

Table 3. Turbo C fonts currently available for use with settextstyle().

Font Name	Value	Description	File Name
DEFAULT_FONT	0	8x8 bit-mappped (default)	---------
TRIPLEX_FONT	1	Stroked triplex font	TRIP.CHR
SMALL_FONT	2	Stroked small font	LITT.CHR
SANS_SERIF_FONT	3	Sansserif font	SANS.CHR
GOTHIC_FONT	4	Gothic font	GOTH.CHR

Listing 4. A display of an entire font character set.

```
/* FONTDISP.C
** A program for displaying the Turbo C fonts in sequence.
*/
#include <stdio.h>
#include <alloc.h>
#include <graphics.h>
#include "grap_enb.h"

#define TRIP "1 = triplex font"
#define SMAL "2 = small font"
#define GOTH "3 = gothic font"
#define SANS "4 = sansserif font"

/* == Begin program == */
main()
{
int tx, ty, i = 65, last = 123, palette = 2;
char entry, alpha, al_buf[4];

  /* ** display font selection query ** */
   puts(TRIP); puts(SMAL); puts(GOTH); puts(SANS);
   printf("Enter the number for the font to be displayed: ");
   scanf("%c",&entry);
   clrscr();

  /* ** register the four stroked fonts ** */
    if(registerbgifont(triplex_font) < 0)   exit(1);
    if(registerbgifont(small_font) < 0)     exit(1);
    if(registerbgifont(gothic_font) < 0)    exit(1);
    if(registerbgifont(sansserif_font) < 0) exit(1);

  /* ** set the graphics mode ** */
    enable_graph(palette);
     setcolor(2);  /* the font color */

  /* ** define the font, direction and size ** */
     if(entry == '1')
      settextstyle(TRIPLEX_FONT,HORIZ_DIR,1);
     else if(entry == '2')
      settextstyle(SMALL_FONT,HORIZ_DIR,4);
     else if(entry == '3')
      settextstyle(GOTHIC_FONT,HORIZ_DIR,1);
     else if(entry == '4')
      settextstyle(SANS_SERIF_FONT,HORIZ_DIR,1);

  /* ** display the character set ** */
    tx = 10; ty = 1;
     while(i <= last)  {
      alpha = (char *)i;
       sprintf(al_buf,"%c",alpha);
        outtextxy(tx,ty,al_buf);
         tx += 23; i++;
          if(i == 91) i = 97;
         else if(i == 123) { i = 48; last = 64; }
        else if(i == 65) { i = 33; last = 47; }
       else if(i == 48) { i = 91; last = 96; }
      else if(i == 97) { i = 123; last = 126; }
     if(tx >= 300) { tx = 10; ty += 25; }
                         }
  /* ** return to text mode and exit ** */
     getch(); closegraph(); exit(0);
}
```

Each font, including the default, is magnifiable over a range of one to ten. The smallest of the fonts, appropriately named SMALL, is barely readable in its minimum size. The fanciest, GOTHIC, is virtually unreadable in any size. The stroked fonts retain a better resolution and appearance than the bit-mapped with magnification. The program FONTDISP.C(Listing 4) displays the entire character set on screen for a requested font. The magnification is an indication of relative size: note SMALL with a factor of eight is about equal to SANSSERIF which is only doubled.

Using the Fonts In Our Programs

The fonts as provided in the library have the file extension .CHR. Anoption available to us is to use them with this extension. Another is to do a file conversion to object form. The converted file, now having the extension .OBJ, may be linked into

Listing 5. An information screen for the TANK_WAR action game.

```
/* TANKPLAY.C
** Demo program for a TANK WAR game play information screen.
*/
#include <stdio.h>
#include <graphics.h>
#include "grap_enb.h"

/* == Begin program == */
main()
{
int palette = 2;
int bx = 30, by = 64, i = 4;
  /* ** firing key data ** */
int key_box[] = { 0,0, 24,0, 24,24, 0,24 };
int key_box_pts = sizeof(key_box)/(2*sizeof(int));
int tx, ty = 10, twx;
  /* ** direction key data ** */
int left_arrow[] = { 4,11, 8,7, 8,11, 19,11, 8,11, 8,15 };
int left_arrow_pts = sizeof(left_arrow)/(2*sizeof(int));
int up_arrow[] = { 8,8, 12,4, 16,8, 12,8, 12,20, 12,8 };
int up_arrow_pts = sizeof(up_arrow)/(2*sizeof(int));
int right_arrow[] = { 4,11, 15,11, 15,7, 19,11, 15,15,
    15,11 };
int right_arrow_pts = sizeof(right_arrow)/(2*sizeof(int));

  enable_graph(palette);

  /* ** register the font ** */
   if(registerbgifont(small_font) < 0) exit(1);
   if(registerbgifont(sansserif_font) < 0) exit(1);

  /* ** display screen title line ** */
    settextstyle(SMALL_FONT,HORIZ_DIR,5);
     setcolor(2);
     twx = textwidth("HERE'S HOW TO PLAY THE TANK WAR GAME");
     tx = getmaxx()/2-twx/2;
     outtextxy(tx,ty,"HERE'S HOW TO PLAY THE TANK WAR GAME");

  /* ** print firing key instructions ** */
    setcolor(1);
    settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
      outtextxy(15,110,"Use these keys ");
      outtextxy(15,120," for shell firing.");
      outtextxy(15,130,"Each key fires at ");
      outtextxy(15,140," a different angle.");
  /* ** print arrow key instructions ** */
      outtextxy(185,110,"Use these keys ");
      outtextxy(185,120," to re-direct ");
      outtextxy(185,130,"  the tank. ");
      outtextxy(185,140," UP is hold. ");
  /* ** print closing message ** */
      outtextxy(30,160,"Have fun now, y' hear?");
      outtextxy(35,170,"Press any key to continue.");

  /* ** draw key outlines ** */
    settextstyle(SANS_SERIF_FONT,HORIZ_DIR,2);
     setcolor(3);
      setfillstyle(1,1);
       while(i--) {
      setviewport(bx,by,bx+24,by+24,1);
     fillpoly(key_box_pts,key_box);
     bx += 32;
    if(i == 3) outtextxy(7,1,"A");
    else if(i == 2)  outtextxy(7,1,"S");
    else if(i == 1)  outtextxy(7,1,"D");
    else if(i == 0)  outtextxy(7,1,"F");
                   }
     bx = 200;  i = 3;

       while(i--) {
     setviewport(bx,by,bx+24,by+24,1);
      setfillstyle(1,1);
       fillpoly(key_box_pts,key_box);
        setfillstyle(1,3);
     if(i == 2) {
       fillpoly(left_arrow_pts,left_arrow);
        bx = 232; by = 36;
                 }
     else if(i == 1) {
       fillpoly(up_arrow_pts,up_arrow);
        bx = 264; by = 64;
                      }
     else if(i == 0) {
       fillpoly(right_arrow_pts,right_arrow);
                      }
                     }
  /* ** return to text mode and exit ** */
   getch(); closegraph(); exit(0);
}
```

our program. It may also be added to a library, such as GRAPHICS.LIB, using the utility TLIB.EXE.

To use the file with its original CHR extension we simply copy it to our working disk. This because in this mode the file is loaded at run time on the call to settextstyle().

As an alternative we can convert the files to object code using the utility BGIOBJ.EXE. To make the conversion simply enter:

```
BGIOBJ.EXE <font name>
```

on the command line. Redirection may be employed if the object file is to be on a different drive. Do not include the .CHR extension; doing so yields an error message.

The object file can then be linked in with the program using TLINK. Or, as mentioned, simply add it to a library file. The addition to GRAPHICS.LIB is a simple procedure easily carried out. Once done there need be no further concern in this regard. To perform the addition enter

```
TLIB GRAPHICS +TRIP +LITT +SMAL +GOTH
```

The original GRAPHICS.LIB is retained with the extension .BAK. One consequence is an increase in file size from 29K to 51K.

In either of these, linking or library addition, the font or fonts to be used, other than the default, must be registered in the using program prior to calling setextstyle(). The registering function is:

```
registerbgifont(void(*font)(void));
```

or, when required by insufficient memory

```
registerfarbgifont(void(*font)(void));.
```

Lines 13 - 16 of program FONTS.C (Listing 3) illustrate the procedure. Note it is not necessary to register the default. The registration incorporates a test for its success.

Before using a specific font it is called by settextstyle(int font, int direction, int charsize);. Note that the call for sansserif includes an underscore between sans and serif. Without the underscore the compiler reports an error. This is NOT shown in the Reference Guide, by the way. I learned it the hard way.

Although the same string, "Hi Y'all!" is repeated for each font a new width and starting column must be calculated. This because the text is centered on the screen for each font.

It would be a great if we could look at the full character set for any font we might have in mind to use. The program FONTDISP.C (Listing 4) does just that for us. When run the program displays the entire set of font characters. The SMALL_FONT charsize is expanded to four for best visibility, the remainder are size one.

The program begins in the text mode with a display of the four font options and a query as to which is to be viewed. On entering a selection number and pressing return the program transitions to the graphic mode and draws the display.

It is our good fortune the fonts all make use of the standard ASCII designations. The entire display is performed with a single while() loop. Because of the way the assignments are made the loop will appear confusing. Upper case alpha chars have the decimal range of 65 for 'A' through 90 for 'Z.' There is a jump then to 97 for the beginning of the lower case letters. These extend to 122 for 'z.' The upper and lower case letters are displayed in this sequence with 13 letters on each line.

The third line begins with the ten numerals 0 - 9. These have the decimal assignment range 48 - 57. Punctuation is distributed over the four ranges of 33 - 47, 58 - 64, 91 - 96, and 123 - 126. An integer variable, last, simplifies the loop. Range limits are detected by if() statements. In these last is assigned the final variable of the new range.

The loop code, abstracted from FONTDISP.C, is:

```
/* ** display the character set ** */
  tx = 10; ty = 10;
   while(i <= last)  {
    alpha = (char *)i;
     sprintf(al_buf,''%c'',alpha);
      outtextxy(tx,ty,al_buf);
       tx += 23; i++;
        if(i == 91) i = 97;
       else if(i == 123) { i = 48; last = 64; }
      else if(i == 65) { i = 33; last = 47; }
     else if(i == 48) { i = 91; last = 96; }
    else if(i == 97) { i = 123; last = 126; }
   if(tx >= 300) { tx = 10; ty += 25; }
                        }
```

Now suppose we apply our new knowledge of fonts and how to use them to the creation of an information screen for TANK_WAR, TANKPLAY.C (Listing 5). The screen displays the two sets of keys, A - F and the three cursor arrow keys for tank direction, used by the player with instructions for their use. A difficulty we have to live with when using the CGA adapter is the forty column screen. This forces us into use of the default font in many situations where we would prefer another but run out of column width in the effort.

Summary

We have learned much of both the capabilities and limitations of the text options available to us in the graphics mode. It is our good fortune to be able to take advantage of the text mode puts(), printf(), and scanf() functions through direct register communication.

Five text character fonts are available through the compiler graphics library. One, the bit-mapped default, is always available for our use. The four stroked style must be registered and defined prior to their use. All the fonts may be magnified over a 1 to 10 range. Though the number of fonts currently available through the compiler are limited we can make significant use of them in game information and title screens. ●

Z80 Communications Gateway

Part 2: The Z80 CTC

by Art Carlson

We covered the RS-232 basics in the previous issue, and will continue in this issue with using a CTC to establish the baud rate. The Z80 CPU provides address and data output lines, but it lacks the internal timers and I/O lines which are found in control oriented processors such as the 8051. For the Z80 CPU these functions are provided by the Z80 CTC (Counter/Timer Circuit) and the Z80 SIO (Serial Input/Output).

Prototyping

I have been prototyping small projects on a solderless breadboard. This has been satisfactory for simple tests such as to check the currents in a transistor driven LED, but when I assembled a Z80 communications prototype on the breadboard it was so flakey that I decided to go back to square one. I've ordered perfboard and wire wrap sockets to rebuild the communications gateway, and will use either wire wrap or point-to-point wiring in the future for similar projects.

Prototyping is sometimes a time consuming nuisance, but it's absolutely necessary—it's the only way that you learn. As long as you limit yourself to the 5 or 12 volts in the logic circuits you won't electrocute yourself, and logic chips are cheap enough so that it won't hurt too much if you smoke a few. In their book, *Interfacing Microcomputers to the Real World*, Sargent and Shoemaker said, "Experience is directly proportional to the amount of equipment ruined." With TTL logic circuits a lot of failures do not actually ruin anything, so I'll restate that as "Experience is directly proportional to the number of experiments which fail." Their book is a goldmine of information, but unfortunately it is out of print.

Counter/Timers

Most assembly language books demonstrate the use of software loops to obtain time delays. For example, with the Z80 running at 4 MHz you can obtain a one second delay using the routine shown in Listing 1 which produces a 'beep' every second. In order to write this you have to look up the number of "T states" for each command, determine how many times to go through the loop, etc. For example, LD takes 7 T states, DEC takes 4 , and JR NZ takes 12 if true or 7 if not true. It gets rather time consuming and confusing—I'm not sure if I remembered to include the time for the BEEP routine. Many high level language implementations include a "sleep" or "wait" function which relieves you from the drudgery of counting the clock cycles.

The primary objection to using software loops for timing is that it keeps the processor occupied full time, and there is no time for it to do other work. The solution to the problem is to use a hardware C/T (Counter/Timer). The IBM PC uses an 8253, which on my '286, provides System Timer, Refresh Request, Speaker Tone, and Mode Control. The Ampro Z80 Little Board uses two channels of a Z80 CTC for generating the serial port baud rates, and the other two channels are available for user programming.

The IBM PC System Timer generates an interrupt 18.206482 times a second (approximately every 55 milliseconds). Your software can count these interrupts, and be doing other tasks except during the brief interrupt service routine. The high level language routines may or may not tie up the processor depending on whether there is a source of interrupts and if they use it.

Embedded controllers almost always include some sort of a hardware Counter/Timer, in fact most controller processors include several Counter/Timers. When the processor does not provide a C/T, or when you need even more channels, you can use peripheral chips such as the Z80 CTC or the 8253. For our Z80 communications system we'll use the Z80 CTC because it is designed to work with the Z80 CPU and the Z80 SIO.

The Z80 CTC

The Z80 CTC contains four Counter/Timer channels. When used as a counter, they count pulses on the appropriate CLK/TRG pin. When used as a timer, they count cycles on the system clock after the clock has been divided by either a 16 or 256 prescaler. The four channels can be programmed independently, and three channels have ZC/TO outputs capable of driving Darlington transistors. The counters decrement the count until it reaches zero, and reloaded automatically. Since the counters are eight bit, the maximum time constant value is 256 (which is prescaled by 16 or 256 in the timer mode), but more than one counter can be cascaded for counts greater than 256.

"Experience is directly proportional to the number of experiments which fail"

Generating a beep every second as we did in Listing 1 would be an example of using a CTC in a Z80 controller. Using a 4 MHz clock and the 256 prescaler would give a prescaler output of 15.625 kHz. A time constant value of 256 (for the longest time) would give 61.035 per second. The longest time is less than one second, so we'll have to either count pulses in software or take the output and feed it into another channel set up as a counter. Since the longest time is too short, we might as well use a nice round decimal value, such as 0.010 second which also provides greater timing resolution if needed for other purposes. Dividing 15.625 kHz by 100 give a a time constant value (commonly referred to as

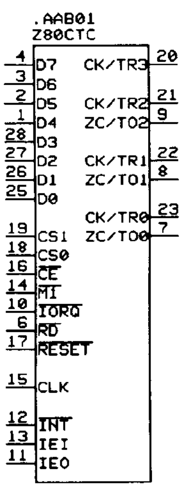


Figure 1: The Z80 CTC pin functions.

Listing 1: Z80 timing loop.

```
; A program to test Z80 timing loop--desired time 1 second.
; VERSION 1.0  RAC 8/3/90 - Written for SLR Z80ASM
; Clock frequency 4 MHz

        ORG     0100H

START   CALL    BEEP

        CALL    DLYA

        JP      START

DLYA    LD      A,4
DLYB    LD      B,0
DLYC    LD      C,0
LOOP    DEC     C
        JR      NZ,LOOP
        DEC     B
        JR      NZ,DLYC
        DEC     A
        JR      NZ,DLYB
        RET

BEEP    LD      C,2
        LD      E,7
        CALL    5
        RET

        END     0100H
```

the reload value) of 156.25. There is a problem here because the 8 bit reload register only accepts integer whole numbers, so we can't hit exactly 100 counts per second using a 4 MHz clock. With a 3.6864 MHz clock the reload value is a nice integer 144—now you know why people sometimes use such odd clock frequencies.

Assuming that the 100.16 counts per second is close enough for our purposes, we can set channel 0 as a timer with a prescaler of 256 and a reload value of 144, and feed the output pulse from TO0 into channel 1 set up as a counter (remember no prescaler when used as a counter) with a reload value of 100. Then the output pulse from ZC1 can be used with a Darlington to pulse a piezo beeper. Once the Z80 CPU transmits the few bytes required to configure the CTC, the CTC will keep generating one second beeps with absolute no further action from the CPU

So, there are three different ways to generate time delays with the Z80. 1) Use software loops which prevent the CPU from doing anything else. 2) Use a CTC to generate interrupts and count the interrupts in software, which only ties up the CPU for a small portion of the time. 3) Configure a CTC and let it do the whole job.

We're going to use the third method because the Z80 CPU is busy enough when preforming high speed communications without the added burden of generating the baud rates.

Generating Baud Rates for the Z80 SIO

The Z80 SIO requires RXC (Receiver Clock) and TXC (Transmitter Clock) signals of 1, 16, 32, or 64 times the data rate. These could be provided by a fixed clock, but I am using the CTC in order to provide a programmable baud rate. The baud rates in common use today are 300, 1200, 2400, 4800, 9600, and 19,200, but you may find rates as low as 50 in older industrial TTY installations. These accepted standard values are convenient for communications between off-the-shelf hardware, but any in between values can be used if you can custom program both ends. If your routines choke on 19.2K baud, but can run faster than 9600, you might find that some non-standard value, perhaps something odd such as 13,800, will provide the highest speed data transfer. But remember, non-standard baud rates only work where you control both the receiver and the transmitter--such as intra-processor communications in multiprocessor embedded systems.

The Z80 CTC is programmed by writing 8-bit words to the Channel Control Word and the Time Constant Word registers. If we were to use the CTC generated interrupts, we would also have to write to the Interrupt Vector Word register. The four channels each have their own channel control and time constant registers which are addressed during programming by the CS0 and CS1 control lines as follows:

Channel	CS1	CS0
0	0	0
1	0	1
2	1	0
3	1	1

The Channel Control Word, shown in Figure 2, configures the channel. To generate 9600 baud for our example we will load it with 17H (00010111 binary) as follows:

Bit	Value	Action
7	0	Disable interrupts.
6	0	Select timer mode.
5	0	Prescaler value of 16.
4	1	Trigger on rising edge.
3	0	Automatic trigger when time constant is loaded.
2	1	Time constant follows.
1	1	Software reset.
0	1	Control word.

The Time constant word is calculated from the clock frequency, the CTC prescaler value, the SIO clock mode, and the baud rate. For 9600 baud with a 4 MHz system clock, a CTC prescaler of 16, and an SIO clock mode of 1, the time constant is 26. The actual baud rate is 9615 instead of the desired 9600, but it is close enough to work. The 3.6864 MHz clock with a time constant of 24 would provide the desired 9600. The time constant is calculated as follows:

```
BIT     ACTION
 7      INTERRUPT
             0 Disables interrupt
             1 Enables interrupt
 6      MODE
             0 Selects timer mode
             1 Selects counter mode
 5      PRESCALER VALUE (Timer mode only)
             0 Value of 16
             1 Value of 256
 4      CLK/TRG EDGE SELECTION
             0 Selects falling edge
             1 Selects rising edge
 3      TIMER TRIGGER (Timer mode only)
             0 Automatic trigger when time constant
                 is loaded
             1 CLK/TRG pulse starts timer
 2      TIME CONSTANT
             0 No time constant follows
             1 Time constant follows
 1      RESET
             0 Continued operation
             1 Software reset
 0      CONTROL OR VECTOR
             0 Vector
             1 Control Word
```

Figrue 2: Z80 CTC Control Word.

```
                  Clock frequency X SIO clock mode
Time constant=-------------------------------------
                        Baud X Prescaler

              4 MHz(1)
            =-----------
              9600(16)

            = 26.04
```

The CTC prescaler, SIO clock mode, CTC Clock/Timer mode, and the clock frequency may all have to be changed in order to cover the full baud range. The Ampro Little board uses a 16 MHz clock which is divided to 4 MHz for the Z80 CPU system clock, and divided to 2 MHz for the Z80 CTC CLK/TRG input for the 110 to 9600 baud range. A 615.385 kHz clock (16 MHz divided by 26) is used for the 9600 to 38.4k baud high range on channel A only. The high frequency clock is applied directly to the SIO without going through the CTC.

Addressing the CTC

The Z80 CPU has separate memory and I/O ports (but you can still use memory mapped I/O if you desire). The instruction LD A,(40H) which moves the contents of memory location 40H to the accumulator, and the instruction IN A,(40H) which moves the contents of input port 40H to the accumulator do not address the same 40H location. The memory and port locations are different even though they have the same Hex address. The Z80 CPU uses the MREQ* line to signify that it is addressing a memory location or the IORQ* line to signify that it is addressing an I/O port.

As an example of addressing, the Ampro Little Board uses address line 4 and 5 to select CS0 and CS1 (see chart above) for channel addressing, and address line 6 to select the CTC. Address lines 6, and 7 go to one half of a 74LS139 dual 2 to 4 decoder (demultiplexer). If bit 7 is 0 and bit 6 is 1, the CTC is selected by pulling the CTC CE* line low. The chart in Figure 3 shows the Little Board addressing.

ADDRESS	BITS 7	6	5	4	CTC CHANNEL SELECT CS0	CS1	CHANNEL
40H	0	1	0	0	0	0	0
50H	0	1	0	1	1	0	1
60H	0	1	1	0	0	1	2
70H	0	1	1	1	1	1	3

Figure 3: Ampro Little Board CTC addressing.

Editor's Note: Since we can not typeset an overbar, we use an asterisk to indicate active low i.e., we set $\overline{CE}$ as CE.*

Configuring the CTC

Once the Channel Control and Time Constant words have been determined, and the CTC addresses have been established, the CTC can be configured. Using the word values for 9600 baud above, the code is as follows:

```
LD A,26         ;Load the channel control word
OUT (40H),A     ;Send it to the CTC
LD A,17H        ;Load time constant
OUT (40H),A     ;Send it to the CTC
```

You will find many uses for Counter/Timers, and should spend some time becoming familiar with the Z80 CTC, the 8253, 8254, and the C/Ts in the various controller processors.

Next Time

I am never sure how much detail to include with a project. I used more space for the CTC than I intended, but I felt that it was important to describe how the CTC works rather than just to say, "Configure the CTC for 9600 baud." Our readers have a wide range of experience and background, and we can't write at exactly the right level for every individual, but I do need your feedback. Tell me if you want lots of details on how and why things work, or just a minimum of information. Should we include tutorial information with our projects, and even some simple starter hardware projects? Do you want more information on addressing, multiplexing and demultiplexing, logic chips, linear devices, etc?

Next time we'll cover the SIO--and it will involve even more than the CTC. I should have my prototype running by that time so that I can include actual communications routines. Send your questions, suggestions, corrections, and articles so that they can be included. ●

THE COMPUTER JOURNAL

Back Issues

Special Close Out Sale on these back issues only

Issues—1, 2, 3, 4 and 8
3 or more, $1.50 each postpaid in the U.S.
Outside of the U.S., 3 or more, $2.50 each postpaid surface.
Other back issues are available at the regular price.

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 8:

- Build VIC-20 EPROM Programmer.
- Multi-User: CP/Net.
- Build High Resolution S-100 Graphics Board: Part 3.
- System Integration, Part 3: CP/M 3.0.
- Linear Optimization with Micros.

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System.
- Database: A continuation of the data base primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking executive.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.
- The Computer Corner

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8088 software to produce modifiable assem. source code.
- Real Computing: The NS32032.
- S-100: EPROM Burner project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System. Part 1: Selecting your assembler, linker and debugger.
- The Computer Corner

Issue Number 36:

- Information Engineering: Introduction.
- Modula-2: A list of reference books.
- Temperature Measurement & Control: Agricultural computer application.
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE.
- Real Computing: NS32032 hardware for experimenter, CPUs in series, software options.
- SPRINT: A review.
- REL-Style Assembly Language for CP/M & ZSystems, part 2.
- Advanced CP/M: Environmental programming.
- The Computer Corner.

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers.
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER.
- Information Engineering: Basic Concepts: fields, field definition, client worksheets.
- Shells: Using ZCPR3 named shell variables to store date variables.
- Resident Programs: A detailed look at TSRs & how they can lead to chaos.
- Advanced CP/M: Raw and cooked console I/O.
- Real Computing: The NS 32000.
- ZSDOS: Anatomy of an Operating System: Part 1.
- The Computer Corner.

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS: Anatomy of an Operating System, Part 2.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:

- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0—The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

Issue Number 41:

- Forth Column: ADTs, Object Oriented Concepts.
- Improving the Ampro LB: Overcoming the 88Mb hard drive limit.
- How to add Data Structures in Forth
- Advanced CP/M: CP/M is hacker's haven, and Z-System Command Scheduler.
- The Z-System Corner: Extended Multiple Command Line, and aliases.
- Programming disk and printer functions with C.
- LINKPRL: Making RSXes easy.
- SCOPY: Copying a series of unrelated files.
- The Computer Corner.

Issue Number 42:

- Dynamic Memory Allocation: Allocating memory at runtime with examples in Forth.
- Using BYE with NZCOM.
- C and the MS-DOS Screen Character Attributes.
- Forth Column: Lists and object oriented Forth.
- The Z-System Corner: Genie, BDS Z and Z-System Fundamentals.
- 68705 Embedded Controller Application: An example of a single-chip microcontroller application.
- Advanced CP/M: PluPerfect Writer and using BDS C with REL files.
- Real Computing: The NS 32000.
- The Computer Corner

Issue Number 43:

- Standardize Your Floppy Disk Drives.
- A New History Shell for ZSystem.
- Heath's HDOS, Then and Now.
- The ZSystem Corner: Software update service, and customizing NZCOM.
- Graphics Programming With C: Graphics routines for the IBM PC, and the Turbo C graphics library.
- Lazy Evaluation: End the evaluation as soon as the result is known.
- S-100: There's still life in the old bus.
- Advanced CP/M: Passing parameters, and complex error recovery.
- Real Computing: The NS32000.
- The Computer Corner.

Issue Number 44:

- Animation with Turbo C Part 1: The Basic Tools.
- Multitasking in Forth: New Micros F68FC11 and Max Forth.
- Mysteries of PC Floppy Disks Revealed: FM, MFM, and the twisted cable.
- DosDisk: MS-DOS disk format emulator for CP/M.
- Advanced CP/M: ZMATE and using lookup and dispatch for passing parameters.
- Real Computing: The NS32000.
- Forth Column: Handling Strings.
- Z-System Corner: MEX and telecommunications.
- The Computer Corner

Issue Number 45:

- Embedded Systems for the Tenderfoot: Getting started with the 8031.
- The Z-System Corner: Using scripts with MEX.
- The Z-System and Turbo Pascal: Patching TURBO.COM to access the Z-System.
- Embedded Applications: Designing a Z80 RS-232 communications gateway, part 1.
- Advanced CP/M: String searches and tuning Jetfind.
- Animation with Turbo C: Part 2, screen interactions.
- Real Computing: The NS32000.
- The Computer Corner.

Subscriptions	**U.S.**	**Foreign (Surface)**	**Total**
1year (6 issues)	$18.00	$24.00	
2 years (12 issues)	$32.00	$46.00	________
Air Mail rates on request.			
Back Issues			
16 thru #43	$3.50 ea.	$4.50 ea.	
6 or more	$3.00 ea.	$4.00 ea.	
#44 and up	$4.50 ea.	$5.50 ea.	
6 or more	$4.00 ea.	$5.00 ea.	________

Issue #s ordered________________________________

Subscription Total ________

Back Issues Total ________

Total Enclosed ________

All funds must be in U.S. dollars on a U.S. bank

Name________________________________

Address________________________________

____

____

☐ Check ☐ VISA ☐ MasterCard Exp. Date____

Card #________________________________

Signature________________________________

The Computer Journal
190 Sullivan Crossroad, Columbia Falls, MT 59912
Phone (406) 257-9119 Mountain Time Zone

Real Computing

The National Semiconductor NS320XX

by Richard Rodman

The New Fax and Laser Processors

National Semiconductor has introduced three new versions of its processors which address the fax machine and laser printer markets. These are the 32FX16, the 32CG160 and the 32GX320.

The 32FX16 is a 32CG16 (graphics processor) with an added vector multiplier (multiplier-accumulator) device added on-chip. This device has a 384-byte scratchpad static RAM, and is intended to perform DSP tasks required by a Group III fax modem. It's not supposed to replace DSP devices in predominantly signal-processing applications, however. Because the multiplier-accumulator instructions are built into the instruction set, it will be possible to build fax machines around the 32FX16 with a very small number of external components. The chip is pin-compatible with the 32CG16 and sells in the $40 range.

Interestingly, other DSP tone-decoding applications, such as DTMF detectors, may be possible using the multiplier-accumulator. If you get a chance to check this out, let me know.

The 32CG160 is a 32CG16 with a fast 16x16 multiplier, a 2-channel DMA controller, an interrupt controller and a bit-blit controller added. This multiplier has faster performance than the original one, but is built into the same instruction opcode. It is intended to increase the performance of Postscript interpreters. The part is intended for laser printers and other graphics devices.

The 32GX320 (formerly the "Barracuda") is similar to the 32CG160, but with a high-performance 32GX32 core instead of a 32CG16 one. Remember that the 32GX32 is basically a 32532 with the MMU and cache coherency logic stripped out. The 32GX320 CPU adds a 32-bit hardware multiplier. The bit instructions (set, clear, and test bit) and Index instructions, which were seldom used because they were slower than regular instructions, have all been improved. Further, four new complex arithmetic instructions were added, again, primarily for the purpose of providing enough DSP capability to make a 9600 baud fax modem in software possible. There is also the 2-channel DMA controller, interrupt controller, and 3 counter-timers. Alas, no blitter.

Personally, I don't find fax all that interesting, except as a possible gateway into a future of standardized, but feature-rich, electronic mail which encompasses device-independent graphics. The possibilities of the built-in DSP capabilities are intriguing, of course--and I'm a big fan of making laser printers more powerful and inexpensive.

Basically, National is making a strong push for dominance in what is becoming a high-volume market for low-cost, high-resolution graphics devices. We have also seen Motorola's new 68302, which is a 68000 with a number of serial ports on chip and a very bizarre RISC CPU which controls the serial ports.

My personal opinion is that the fad for specialized parts is good only to a point. When a special processor instruction set is needed for a part, or when a part is not available without complicated functional blocks that are not usable because they are too highly specialized, things have gone too far. Remember the video game chips of yore? Fortunately, both National and Motorola seem to be taking care not to get too specialized.

Rumor Mill

National's announcements also added fuel to another rumor that National is working on a package similar to TI's TIGA, but built around the 32CG16. This will be a high-resolution graphics standard which will support X Window primitives directly. It will avoid TI's astronomical fees, and be cheaper and better.

Presently, X terminals are mostly built around Motorola CPUs, some with 34010s as well. It appears that a bandwagon is building to make Ethernet and TCP/IP the "next RS-232", but costs are still too high to get the Big Mo.

Futurebus: Is It Going Places, or Taking People for a Ride?

Futurebus was a poorly chosen name: once it's here, will we change it to Presentbus? The original standard has now evolved into what is called Futurebus+ (Futurebus Plus), and several major vendors have announced chip sets, backplanes, and card cages meeting the interim specification. Industry consensus is beginning to gel that Futurebus+ is indeed better than present-day busses.

However, the VME and Multibus-II camps each have existing inventory and development costs to amortize, and thus are trying to direct their adherents into an "evolutionary" pathway which will ultimately lead to Futurebus+. It's a shame that this wasn't done in the past. For example, suppose that S-100 boards had been available with a PC-bus connector on the other side, so that you could connect PC-bus boards into an S-100 system. Then users could have gradually "evolved" into the PC bus without junking any S-100 hardware, right?

ECHH!

Well, while the VME people appear to have real plans to put Futurebus signals on the P2 connector, the Multibus-II people have taken a more SAA-like stance (i.e. posturing without any real activity). They have made statements along the line of Futurebus being "strategic for the future".

Let's face it, all of these old boards are going to be junked eventually anyway. If you're going to make a transition to a new bus, do so cleanly—cold turkey.

So, just what are the advantages of Futurebus+? First and foremost, it's fast. BTL (Backplane Transceiver Logic) drives the backplane at only a 3-volt swing to allow higher slew rates (dV/dt--

reduce dV, and you can reduce dt). The chips being developed have propagation delays of 2 to 3 nanoseconds. There's a packet mode in which boards can exchange data at 100 megabytes per second.

Now remember that it takes electricity about 8 nanoseconds to go a foot through copper, so a 2 nanosecond delay is about 3 inches of PC board trace. This means that traces not only have to be short, but all data signals must travel through traces that are equal in length. Care must be taken when going around corners.

The boards are the same size as VME bus boards, but the data bus can be 32, 64, 128, or even 256 bits wide. At 256 bits wide, speeds of up to 1 gigabyte per second are possible. But with so little room for logic, and with such high speeds necessitating short traces, surface-mount technology will be needed. Traces will have to be modeled as transmission lines. Power consumption will be high--bus drivers have to be capable of driving 100 milliamps. The dense boards will generate lots of heat.

Yes, Futurebus+ boards will be real works of art--and priced like them. There is a high-end board market which is not cost-sensitive, but this market is only a tiny fraction of the overall board market which extends down to the low-cost STD and G64 bus cards. And the high end is the hunting ground of proprietary buses such as XMI. For this reason, it's hard at this time to foresee a bright Futurebus+.

The CD Conspiracy Continued

It was pointed out to me that the Notch copy-protection scheme used in CDs, to prevent them being copied by DATs, can be easily bypassed by injecting a small amount of energy at precisely 15 kHz. It's a shame that the distortion to the original signal can't be corrected so easily. Once again, everyone suffers but the pirate. There is **no copy-protection scheme which does not have this characteristic.** When will people learn?

Minix Miscellany

Recent news on the Minix front: the software package is available in bookstores now, version 1.3 for PCs. From Prentice-Hall, a version is available for the Atari ST. Versions for the Macintosh and Amiga will be available shortly. The next big distribution, supposedly this fall, will be version 1.5. Version 1.5 is in the hands of a lot of people now, because it has been distributed in the form of "cdiffs" (context diffs) from version 1.3.

The important point to note is that while you get source code and can port it to any machine you like, Minix is **not** public domain. While they have fumbled around a lot on their way into the software distribution business, Prentice-Hall seems to be getting their act together lately, and Andy Tanenbaum (the author of Minix) is struggling to make the various 68000 Minixes binary- and media-compatible. Think of it! The three mass-market 68000 PCs actually able to run the same code! It boggles the mind. ●

Computer Corner

(Continued from page 39)

it can perform parallel instruction operations. The only flaw in the design was setting one bit aside for return flag. This makes the 16 bit device actually 15 bit addressing or 32K memory pages, with a max of 512K possible (16 pages of 32K each). In Forth it is important to point out that very little memory is needed for complex programs. I have a NOVIX CAD program that runs very well in 32K of memory (including the operating system).

By having the ability to perform more than one operation per clock cycle, it means a 8 MHZ cpu could reach 16MIPS (million instructions per second) speed. I believe the 20 MHZ 386 is around 4 or 5 MIPS at best condition. Some work has been done with the NOVIX which showed that sustained throughput is possible near twice the clock speed. How fast above the clock speed is based on compiler optimization. I need to point out that the NOVIX had only 40,000 transistors, while a 386 is pushing a million transistors. The RTX and NOVIX are simple designs with better reliability and ease of programming. Chip design bugs would be easy to find in the RTX, where as the 386 could have design problems that may take years to appear because of the complexity.

Lastly is the getting information for my test project. I want to drive some LEDs and can't seem to find out just what the current ability of the device is. In reviewing the books, I find little if anything about hardware connections. This appears to be a big mistake on Harris' part, as a large part of the project will be interfacing to the real world. There is nothing in the book about how you are suppose to connect the ASIC bus to devices. I want to know does it work like the Noivx which could load up to 30MA, be latched high or low and kept that way till toggled off. The Novix could be used as an I/O device, but the book really doesn't explain if the RTX can. Guess I will be calling Harris on this one.

Till Later

Well I think I have said enough about the hardware and design of the RTX, what is needed next is some code comparisons. I have done a few small things lately on other devices and will find the old code next. I did an industrial product and may try porting it to the RTX to just see the difference. Since I am getting ready to move again, the term 'finding' has developed a new meaning. In any case, for next time, programming the RTX. •

Editor

(Continued from page 2)

(Needham Electronics), EPROM eraser (Ultra Violet Products), and a bar code reader (Adaptive Technologies). I've also stocked up on processor chips, logic chips, EPROMs, SRAMs, LEDs, transistors, motors, shaft encoders, plus resistors, caps, sockets and all the other necessary odds and ends.

I originally intended to build my first projects around the Z80, but as I became more familiar with the 8051 family I switched. Working on hardware projects with the Z80 took too long and required too many chips. With the 8051 tools listed above I can go downstairs after supper, write a small test program, assemble it, run it on the simulator, burn an EPROM, and run it on the development board.

Burning, testing, and erasing EPROMs is a bit of a nuisance, but it is not quite as bad as it first appeared. It takes about 15 minutes to erase an EPROM, but 2764s only cost about $4.00 and I have enough of them so that I don't have to wait until the first one is erased before I program the next one. I am planning a RAM/ROM, but it will have to wait till I get to it.

So far, the programs have been short samples to get used to the 8051 command set and to gain experience with interfacing to logic chips, buffers, transistors, etc. I have an overabundance of projects waiting for development. One of the projects is using sonic and ultrasonic transmissions from a piezo transducer to control pests and varmints. One application will be to keep the birds out of a fruit tree so that they don't harvest the crop before I do. Another application will be to keep certain insects out of the garden. Mole and gopher repellers have been done before, but I also want to be able to control them. Each of these uses involves a different frequency spectrum, and researching the sound pattern to use for the different species requires much more time and effort than the programming and software design..

I am very interested in controlling motion with motors. Designing and building maze running robots is a popular application, but I am more interested in numeric machining and model train control. Not everyone has a machine tool to control, but many of you have (or can more readily obtain) a small model train layout. I'd like to hear from any of you who are interested in the area of hobby type robotics.

Some of the other projects in the queue are mouse drivers to control motor driven devices from a small embedded controller (NOT involving a desktop computer), and using a laser diode interferometer to measure small distances for numeric machining and robot control. Again, I'd like to hear your ideas and suggestions.

Controller Market Activity

The controller market is much different than the consumer desktop computer market. The desktop market changes quite rapidly, and the results are quite visible. It is apparent that the two important chip families are the 80X86 and the 680X0. The 8088 (PCs and XTs) is fading very rapidly, and the '286 (AT's) has peaked. The '386 and '386SX are hot right now, and there is tremendous interest in the '486. I don't follow the 68000 very closely, but the activity is moving from the 68020 to the '030 and '040. Everything is speeding up in the desktop arena and it appears that a CPU will be hot for about 2-3 years before it will be obsoleted. This makes for a lot of upgrade activity, and doesn't allow software people time to become proficient with the current version before it is replaced with the newer version. The user is faced with the financial and training problems of constant hardware, operating system, and software revisions. This constant churning is one of the reasons that I don't participate in this field.

The controller market is driven by price, and pennies are important. If you are going to produce 100,000 microwave ovens, and a 4-bit processor is sufficient, you use the 4-bitter instead of the latest 16-bit wonders. On the other hand, if your product is a real time control system, only the fastest 16-bit (or even 32-bit) processor may suffice--I have talked to several people who are using the 680X0 for embedded applications.

Designs favor multitasking on fast powerful processors where cost and board size are critical, but I tend to favor distributed processing on smaller multiprocessor systems when ever possible. Making the single/multi processor design is one of the subjects that we will cover.

As always, your feedback is important. Take the time to let us know what you are doing—even better write an article to share your ideas. •

Computer Corner

(Continued from page 40)

give up a really marketable product. I know I would not.

The problem that got me was time. When it arrived I discovered the deadline was June 8, or 45 days later. Having since changed jobs, and other projects taking priority, my spare time has become quite sparse. What made it more of a problem was the hardware itself. I would need to add more memory to the development board for my big project and they didn't lay out the circuit for it. I think it was a little short sighted on their part to not include the needed printed circuit work for the extra memory. Their book explains how and provides a schematic, but you only get a bread board area. So to meet my deadline I would also have to hand wire in two high speed memory chips. At that point I gave up trying to develop my product for the contest.

The Hardware

If you were an early winner you received the development board and a very good manual. The shipping invoice stated it had a market price of $100 (75 for the board, 25 for the book). Many Forth users are now trying to get Harris to sell them the board at that price (I will let you know if they succeed). I found the book to be very good if you know Forth, and they give you other sources to seek out if you are a novice at Forth. It consists of a USERS GUIDE, HARDWARE REFERENCE MANUAL, CONTEST PROGRAMMERS REFERENCE MANUAL, and EBFORTH SOFTWARE USER'S MANUAL. In all the manual has about 400 pages including, sample code, schematics, and several glossaries.

I had to make an adapter from the 9 pin PC type serial plug supplied to a regular DB25 so I could use my modem program to talk to it. The power supply is a 4 'AA' battery holder and plug. You are suppose to get 3 or more hours of use on the battery pack. They supply an extra two pin plug so you can connect to a regular 5 volt supply. They say the supply should be capable of 150MA (not very much of a current draw). Once the power and serial line is connect to your computer it is ready to run. I often forget to hit the 'B' key, which checks for baud rate, and end up with nothing or very unreadable output. Just push the onboard reset button and hit 'B' once and the sign on message should appear.

The board is 4 by 6.3 inches in size. Half the board is 0.1 inch holes for bread boarding. The other half contains the RTX2001AX chip, the MAP chip, an RS232 interface chip, an 74AC00, a 16MHZ crystal oscillator, and numerous caps and sip resistors. All components except the MAP are surface mounted, MAP is in a socket. The prototype area is not solder through holes, but holes with a thin cross hatch trace running on both sides of the board. The grid or cross hatch provides power on the top and ground on the bottom. Personally I am a bit skeptical of the grid, as it looks too easy to get both the top and bottom soldered together, otherwise know as shorting out the power. A very fine touch with the soldering iron will be needed here.

Harris does produce a more expensive proto-type and development board for the RTX series. This contest board was designed to be cheap and allow for some minor development work. It does give you a chance to test and experiment with some simple operations. Next issue I will give you a sample simple program to test it out, but this time I want to review mostly the hardware, especially the MAP chip.

The main chip is of course the RTX processor, but it would not be usable without the EBFORTH in ROM and some extra RAM. All that is supplied in a special chip called the MAP. The MAP168-55 is made by Waferscale Integration, Inc. and consists of 16K EPROM and 4K RAM. The MAP name comes from the PAD or programmable address decoder, which allows the user to program the location of the memory (map to a location of your choice). Gated buffers are included making for a two chip computer (CPU and MAP). The RTX is a 16 bit wide device and thus the MAP is set up for 16 bit data transfers (one of its options). That means 2K of words storage and after EBFORTH sets up its own tables, about 1.6K of words for dictionary is left (actually 3228 bytes).

It appears that the Forth is also special and not their full development program. This is where another problem exists, the book is just full of information. So much information is provided you quickly get confused and overloaded. The programmers reference manual starts you out with block diagrams of the RTX core and then drops off the deep end with internal register usage followed by detailed instruction definitions and samples. I am sure if you really start using the chip, you will be very glad of the information provided, but for a quick project just too much information is provided. A number of smaller chapters do provide some "how to" information and samples, but just not enough for my tastes. I feel most users will need a little more hand holding to get started than the manual provides.

RTX2001AX

The RTX is an outgrowth of the NOVIX chip. As the Harris people are happy to say "a NOVIX with things done right!" Harris has also added to the NOVIX core with some of their own ASIC devices. What they did was use the NOVIX with minor changes, added timers, interrupt controller, stack controller and stack RAM, ASIC bus interface (a Harris standard interface), optional multiplier, memory control devices, and other needed glue chips. All this and more can be had in an RTX product. The idea is to provide as many optional 'STANDARD' products as possible. That allows the customer to pick and choose functions as needed. All developed by using their regular products during the development stage and only making one chip when the idea is proven to work.

The RTX2000 series of chips has been delayed as I understand, because Harris had such a large special order from one company, they had little time (or incentive) to produce some standard products. For low quantity use standard products are needed and thus the RTX2000. As it stands now, I believe there are only the two products available, a RTX2000 and RTX2001. The 2001 is cheaper with no multiplier (math processor), less stack memory on board (64 bytes not 256) and a couple of internal registers being different. I seem to remember reading (although I can not find it now) some references to minor other difference between the standard RTX2001A and the evaluation RTX2001AX used in the contest board.

The hard part is describing the internal operation of the RTX. The core is composed of some 24 (23 in RTX2000) registers, arranged to provide the functional equivalent operation in hardware of what has been a software architecture. The Forth architecture is based on a two stack system (a data stack, and a return stack). These two stacks allow for separate operations on the data or program direction. In the case of the RTX it allows for single cycle return instructions.

A very large number of the high level Forth words are directly convertible to the RTX machine instructions. In some cases just setting a bit of the instruction word will cause the Forth operation. This means

(Continued on page 38)

The Computer Corner

by Bill Kibler

I have two topics to cover, one strictly software, the other is a start on the Harris RTX system.

By this time everyone should have received their copy of the new WINDOW 3 package. We were able to get one at work before they ran out. Seems Microsoft was getting over 6000 calls a day. It will be interesting to see how fast selling tapers off. There are plenty of reviews elsewhere so I will only touch on a few things that interested me.

I like the new screen display, it is much better than their WINDOW 2, but still not as good as 68000 based window programs. The use of memory seems to be getting better, but unless you have a 386 with about 4 megabytes total memory most of the features are not usable. Our biggest problem is the absence of programmer support. Microsoft's SDK (software developers kit) is not supposed to be out until a month after WINDOW 3 release. We got the product to see if our resident programs can co-exist with the product.

Our products use special key operations to switch between our products and the PC. It has been interesting to see just how many programs go in and steal keystrokes. The IBM LAN manager takes key strokes before the handler, which is exactly what all their documentation says you are not to do. It seems most of our problems stem from Microsoft and IBM not following their own rules. Windows now requires a special program as they steal all the keystrokes and don't give you any way of getting them. To write a special program you need the SDK and of course no SDK package is available yet.

My personal position on WINDOWs is that it is a nice product to try, but most users will remove the program from their hard disk (all 4.5 megabytes) after trying it a few times. At a local computer club meeting, several had already tested it, and found it running slower than WINDOW 2. They said about 50% speed on a 286, while it runs at 75% of the speed with a 386 and 3 megs of extended memory. My test show it runs far too slow no matter what machine you use. Personally, the features it provides are useless to me and as such I have no use for it at all. It may at some point be needed however, as many companies are planning on using it as a graphical interface to their next version. The new GUI (graphic user interface) will make interfacing programs to it easier (once they release the SDK). With that ability I expect more companies to use it. My concern is how much memory and hardware these new programs will need.

A Post Script Driver

A friend of mine just got himself a new 24 pin Epson printer, and it came with a discount offer on a postscript driver package. He didn't have 3 inch 1.44 meg drives so he ask me to move it to 780K 3 inch drives. In doing that I tried the program on my printer, an non-epson compatible device. Well needless to say I didn't get a usable output. The point of interest was how the package was set up.

Let's start from the top of the problems list. First off it came only on 1.44 meg disks. If you are going to offer a package on 3 inch disk, 780K is the standard format right now. It was zipped with ZOO as two files. I used PKZIP and put it on three, 780K disks. With the cost of disks these days, three 780K are about the same price as two 1.44 meggers. No saving on that, just user problems.

Let's talk programs next. I must give them credit for seeing the need for several different programs. The smallest running program however does require at least 400K of extended memory. If you want to run the program with other programs (I.E. a word processor) you will need 1 to 2 megs of extended memory. A math coprocessor also would help. I know that post script is a math intensive program, but when I ran their one line post script sample it took from 5 to 10 minutes before being ready to output to a printer. The variations were due to which printer version I installed it for.

The point I am making about both WINDOWs and the post script driver is their need for specific hardware. The supposed selling point of the PC was it's hardware independence. Programs were supposed to work regardless of your hardware configuration. First we had graphic programs that started writing directly to certain video ram, but not all. We had always been free of disk format problems, everybody just used 360K disks. Now with programs requiring 4 to 8 megs of hard disk, disk formats are becoming an issue. It is starting to be necessary as well, that you have a 20/25 MHZ 386 with 4 to 8 megs of memory to run most programs.

What I question is which are they selling, software or hardware? It seems more lately hardware. For my money 68000 based machines are looking better and better all the time. They do everything the new programs can do with less hardware. Personally, it is starting to make me feel like letting others try all the new programs, while I stick with my smaller and faster programs running on cheap hardware.

Harris RTX

Speaking of faster and cheaper solutions, I have promised a review on the RTX2001. Harris and *Embedded System Magazine*, sponsored a contest on using the RTX as an embedded controller. I sent in my request very early, but didn't get a board until early MAY. You see the idea was to have you submit an idea and flow chart of your project. I wanted to do an AX.25 (amateur radio X.25 packet system) on a single card. They would select winners, well good ideas versus bad ideas, and send the winners a card with an RTX2001A on board. It was suppose to be a development system, and would allow you to develop your product(contest product that is).

I can see from the letters that arrived after my board that a number of problems developed. The big problem was a reluctance to develop anything marketable, as Harris retained all right to code. They of course want to have plenty of application notes to help sell the chips. I think it is a good way of doing it, it is just most will not

(Continued on page 39)