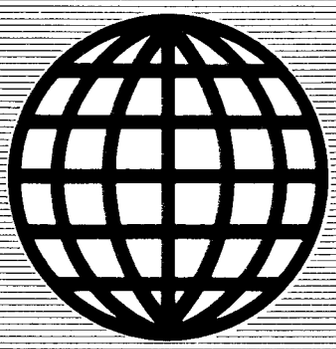


The COMPUTER JOURNAL

Programming - User Support
Applications

Issue Number 47 November / December 1990 $3.95

Controlling Stepper Motors with the F68HC11

The Z-System Corner

Using 8031 Interrupts

T-1 Telecomunications Standard

Modula-2 Makes the Z-System Connection

Interfacing the 68HC705 to LCDs

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

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.

Registered Trademarks

The COMPUTER JOURNAL

Issue Number 47 November / December 1990

Editor's Page

Congratulations!!

Matthew Mercaldo, author of *Multitasking in Forth* in issue #44, and the stepper motor series starting in this issue, was the second runner for the software entry in the Harris RTX contest. His entry was an application note on the use of expert systems for stepper motor control.

We are fortunate in having people of this caliber writing for TCJ, and are looking forward to his next article in the stepper motor series. Matthew, like so many other of our authors, is very busy, but perhaps he'll also find time to tell us about how the RTX can be applied to AI in the real world.

Harris was swamped with entries, and the competition was very stiff, so Matthew deserves to be proud of his accomplishment.

ZSDOS Programmer's Manual

We covered ZSDOS, which is a CP/M BDOS replacement written in the spirit of the ZCPR command processor, in issues #37 and #38. It is a good way to wring more performance from the 8-bit systems, and corrects many of CP/M's shortcomings. Some of its features are file datestamping (built-in or modular), Public and Path file access, automatic login of changed diskettes, and fast relog of fixed disks.

Now, Carson Wilson, coauthor of ZSDOS, author of ZDE, and Sysop of Antelope Freeway (modem: Antelope Freeway RAS, 312/764-5126, Chicago) has published the manual on programming for ZSDOS. It is available for $10 from Carson, Plu*Perfect Systems, or Sage Microsystems East. It is hard to find books on eight-bit systems and many of them are never reprinted, so I suggest that you order yours now so that you won't be disappointed.

RAM, ROM, and R?M

We are all familiar with ROM which is read only, RAM which we can read and write, and EPROM (plus EEPROM) which we can program and then read. But there is another variation, which can cause some strange debugging problems where it is difficult to determine if the problem is software or hardware.

I was talking to someone who had run across a strange effect with static memory devices. While working with embedded Z80 controllers connected to peripheral chips (I forgot whether it was the 8253 Programmable Interval Timer or the 8255 Programmable Peripheral I/O) the device would not always come up correctly.

They finally discovered that, after holding a value for some time, the Z80 and peripheral chip registers would often retain their previous value after being powered down and restarted. This can be a nebulous trap if you fail to initialize all registers on reset. If the registers always came up with random garbage it would be easy to find the problem. But when it sometimes comes up OK, it makes the solution more difficult.

I don't know how long the effect lasts, what percentage of the chips react this way, or if some chips will reliably exhibit this characteristic. But, if you lose power in the midst of an operation, you may be able to recover some of the data by powering up with out re-initializing and saving the registers. I plan on setting up a small SBC to evaluate this effect. Has anyone else noticed this? Does it also hold true for any dynamic RAM devices? I would appreciate hearing from you.

WordStar Strikes Again

One of my long delayed projects is a file printing utility which will output what I want the way I want it. The WordStar 4.0 print command bit me again, so I rotated the print utility back to the top of the stack. My project stack (actually it's a pile) is definitely not FIFO (First In First Out) or FILO First In Last Out). Sometime I think it's FINO (First In Never Out), and when it avalanches to the floor it gets rearranged in random order. There are some interesting results when a spill results in the contents of two project folders being intermingled – perhaps that is serendipity (an aptitude for making desirable discoveries by accident) at work.

While most printer drivers are concentrating on WYSIWYG (What You See Is What You Get) for typographic and graphic page preparation, I want YGWRT (You Get What's Really There). I'll use PageMaker for graphic page preparation, but I want a sterile print driver for programming and text files.

The WordStar driver considers any line which begins with a period as a comment or command, and does not print the line. Our authors frequently refer to file extensions, such as COM or PRN, which should properly have a preceding period. WordStar sets the text OK if the file extension is within the line but will lose it if reformatting puts the file extension at a beginning of a line. It would be better if WordStar only considered lines starting after a hard return, but it also ignores lines which it reformats with soft returns. There is the same problem with decimal numbers. My current fix is to make sure that nothing starts with a period by dropping the period

(Continued on page 33)

Controlling Stepper Motors with the F68HC11

by Matthew Mercaldo

When I was a little boy, machinery fascinated me. I was obsessed with the science of movement—as far as a little boy can be obsessed. I would develop the stuff of star drive and robot with my tinker toy set and my imagination. On Christmas morning, through hastily flung wrappings, these machines would come to life from the building blocks that Santa left. After completing any one of my machinations I'd tinker, ponder, then contemplate the what if: "What if I could make it move on its own?".

This and the following series of articles will teach the theory of stepper motor control and demonstrate principles through application. The motors will be spun with a NewMicros F68HC11 based processor and motor driver board. These articles will not concern themselves with the physics of magnet and coil, but will instead focus on the control of steppers. This control includes the acceleration, deceleration, starting, stopping, and position tracking issues along with error detection and correction which are associated with stepper motor control. This first article will delve into stepper motor control theory. The second article will apply the theory to one stepper motor. The third article will build on the first two and explain how to control two motors simultaneously, as well as examine all the issues associated with this motor asynchronisity. Now for some theory.

If the acceleration, sustaining of constant velocity and deceleration of a motor were viewed on a graph with coordinates of time vs. velocity, a trapezoid is seen. Positive acceleration is the upward sloping line, constant velocity is the plateau, and deceleration is the downward sloping line. Figure 1 illustrates this trapezoid.

There are three guiding variables in the stepper motor control model described herein: the accumulator, the rate counter, and the acceleration counter. The accumulator is a counter which is decremented by the rate counter on every periodic interrupt. When the accumulator's value reaches zero, a new phase is sent to the stepper motor windings. The rate counter determines the velocity of the stepper motor if kept constant. On acceleration and deceleration, the rate counter is updated by the acceleration counter. The acceleration counter determines the acceleration or deceleration of the stepper motor. Upon acceleration, this counter is set with the acceleration constant, a positive number. Upon deceleration, this counter is set with the deceleration constant, a negative number.

There are four critical constants in this stepper motor model: step count, acceleration count, deceleration count, and regular interrupt period. Step count is a constant used to refresh the accumulator. This constant, in conjunction with the rate count counter, determines the motor's minimum velocity. The acceleration constant is a positive value which determines the motor's acceleration. The constant for deceleration is a negative value which determines the motor's deceleration. The regular interrupt period determines the maximum motor velocity. This model is assumed to be driven by a regular interrupt.

When a motor start is initiated, the accumulator is set to the constant step count and the rate counter is set to the acceleration constant's value. Figure 2 shows an initial setup for the counters.

Upon interrupt, the rate counter is subtracted from the accumulator. This happens until the accumulator reaches zero. When the accumulator reaches zero, three things occur: first, the accumulator is reset with the constant step count; second, an acceleration constant is added to the rate counter via the acceleration counter; third, a new step command is sent to the hardware. This cycle continues until the rate counter either accumulates a value equal to the step count or the rate counter decreases in value to zero.

On an acceleration cycle the rate counter is initialized to the acceleration constant's value. As cycles progress the rate counter's value increases. This accumulation of the rate counter progresses until the rate counter's value is equal to or greater than step count; the motor is at its maximum speed. Figure 3 illustrates this acceleration.

The speed attained by acceleration can be sustained by the state which writes a new step to the stepper motor on each periodic interrupt over a specified period of time. Figure 4 illustrates the constant velocity plateau.

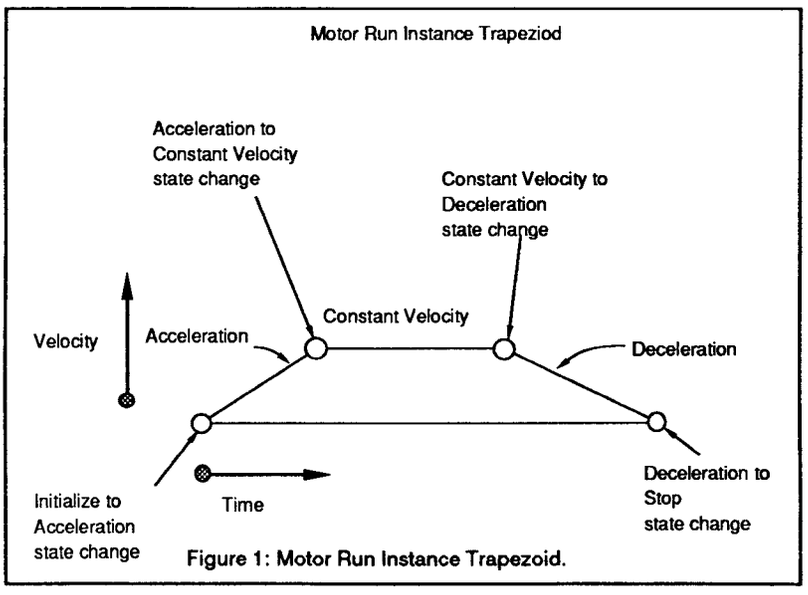


Figure 1: Motor Run Instance Trapezoid.

```
Initial Values of Constants:

Step Count = 10
Acceleration Count = 2
Deceleration Count = -1

Initial Values of Variables:

Accumulator = 10
Rate Counter = 2
Acceleration Counter = 2
```

Figure 2.Initial Variable Settings for a Stepper Run.

After this time the deceleration of the motor is initiated. Upon initiation of the deceleration cycle, the rate counter's value is equal to the value of step count. The deceleration constant is used, via the acceleration counter, to update the rate counter in subsequent deceleration cycles. In order to decelerate the motor, the rate counter must be decreased. Since the deceleration constant is a negative number, addition to the rate counter decreases the rate counter's value. This decreasing of the rate counter causes the frequency of writes to the stepper motor windings to decrease. The motor decelerates to a stop. Figure 5 illustrates motor deceleration.

From the above illustrations one can see the concept behind the acceleration and deceleration of a stepper motor. Acceleration and deceleration of a stepper is required to maximize stepper motor function and usage. Next we need to explore the concepts behind tracking and controlling stepper motors.

A stepper motor is usually found in a closed system. This means that the start and stop points as well as individual steps are all monitored by switches and encoder devices. When the motor reaches some "home" position a switch is actuated. When the motor reaches some maximum limit another switch is actuated. The software looks at these switches to see if they are open or closed depending on where it thinks the motor is in the system. Limit switches work in conjunction with encoder devices in the typical robotic scenario. An encoder is a counting device that when turned in a specific direction generates clock pulses. From these pulses the speed and direction as well as distance traveled can be ascertained. When the motor spins in one direction, counts are accumulated in the forward counter; when the motor spins in the opposite direction, counts are accumulated in the reverse counter. An encoder has a higher resolution than the stepper motor's step. This means that there are many encoder "ticks" for each motor step.

Let us assume a stepper motor system where each motor step is five encoder ticks. We send the software a command to step three steps. Several actions are taken to step the stepper three steps. The steps are converted into encoder ticks (there are 15 ticks in 3 steps), and the current encoder position is retained by the software. Fifteen is then added to the retained encoder value (this new variable is the Target). The motor is stepped three times, and the new encoder count is compared with the Target count. With these counts being equal, we know we have reached our destination!

One can guess that it never works that smoothly, and would be correct. Assuming thus let's cover two more concepts: settle time and boost.

When a stepper motor steps, a certain amount of energy is developed in the mechanical portion of the system. When the new step is complete, inertia wants to continue the step's movement. The magnets in the motor want to stop the step. A kind of "rubber banding" effect occurs in which the motor's magnets and the system's inertia fight until they both tire out. The armature "wiggles" about the new step. The time for this oscillation to stop is called settle time. Let's call the forward "wiggles" forward counts and the back "wiggles" back counts. Depending on the encoder's resolution, these forward and back counts can make the encoder count one or two ticks in each direction. Since the "rubber banding" happens more than once, erroneous counts are accumulated. Usually the back counts are symmetric with the forward counts. In this case erroneous accumulation of counts can be accounted for in software by subtracting the forward counter from the reverse counter. The symmetric counts are eliminated thus giving an absolute distance in encoder ticks.

Boost is a stepper driving technique which increases the stepper motor's initial torque. A higher voltage is applied to the motor coils under software control. This "boost" is applied for only part of the motor step cycle—only a limited amount of regular interrupts as described in the acceleration model above. Boost is typically used in high torque applications where more power is required to initially "start the wheel moving."

Interrupt	Accumulator	Rate Count	Acceleration Count	
Reset values	10	2	2	New Step
1	8	2	2	
2	6	2	2	
3	4	2	2	
4	2	2	2	
5	0	2	2	New Step
Reset values	10	4	2	
6	6	4	2	
7	2	4	2	
8	-2	4	2	New Step
Reset values	10	6	2	
9	4	6	2	
10	-2	6	2	New Step
Reset values	10	8	2	
11	2	8	2	
12	-6	8	2	New Step
Reset valuees	10	10	2	
13	0	10	2	New Step

Stepper Acceleration Complete

Figure 3.Counter Activity During the Acceleration of a Stepper.

Interrupt	Accumulator	Rate Count	Acceleration Count	
Reset values	10	10	2	
14	0	10	2	New Step
Reset values	10	10	2	
15	0	10	2	New Step
Reset values	10	10	2	
16	0	10	2	New Step

Constant Velocity Complete

Figure 4.Counter Activity During Constant Velocity of Stepper Run.

Figure 5.Counter Activity During Deceleration of a Stepper.

Interrupt	Accumulator	Rate Count	Acceleration Count	
Reset values	10	10	-1	
17	0	10	-1	New Step
Reset values	10	9	-1	
18	1	9	-1	
19	-8	9	-1	New Step
Reset values	10	8	-1	
20	2	8	-1	
21	-6	8	-1	New Step
Reset values	10	7	-1	
22	3	7	-1	
23	-4	7	-1	New Step
Reset values	10	6	-1	
24	4	6	-1	
25	-2	6	-1	New Step
Reset values	10	5	-1	
26	5	5	-1	
27	0	5	-1	New Step
Reset values	10	4	-1	
28	6	4	-1	
29	1	4	-1	
30	-3	4	-1	New Step
Reset values	10	3	-1	
31	7	3	-1	
32	4	3	-1	
33	1	3	-1	
34	-2	3	-1	New Step
Reset values	10	2	-1	
35	8	2	-1	
36	6	2	-1	
37	4	2	-1	
38	2	2	-1	
39	0	2	-1	New Step
Reset values	10	1	-1	
40	9	1	-1	
41	8	1	-1	
42	7	1	-1	
43	6	1	-1	
44	5	1	-1	
45	4	1	-1	
46	3	1	-1	
47	2	1	-1	
48	1	1	-1	
49	0	1	-1	New Step
Reset values	10	0	-1	
Deceleration Complete				

In review, within the typical stepper system, a step accounts for a certain distance. The mechanical part of the system is adjusted accordingly. (One doesn't want gizmo's arm to move too far when told to pick up the block.) Steps are correlated to numbers of encoder ticks. Encoder ticks, along with limit switches, let the software track the motor's position. The motor can start moving by the combination of boost and a slow acceleration if more torque is required, it can maintain velocity, and it can slow down by slowing down the step rate in the deceleration state.

The next article is this series will apply some of this theory to spin a motor and track it using the NewMicros F68HC11 and the NewMicros Stepper Driver board. Until then, take care and have fun with Forth. ●

The Z-System Corner

by Jay Sage

Last time I presented an overview of the philosophy behind the design of the ZMATE macro text editor and wordprocessor. In particular, I described the approach that allows the user of the program to implement his/her own text processing functions and to bind them to arbitrary sequences of keystrokes. In other words, you can design your own wordprocessor!

This time I am going to begin a description of the macro command language that ZMATE uses. For this column we will start with relatively simple macros; in future columns we'll begin to display some of the fancy things that ZMATE can do. Even if you don't own or use ZMATE (yet), I hope you will find it interesting to learn about this approach to the implementation of a text editor.

For those of you who remember my promises from two issues back, I'm afraid that my computer has still not been restored to full operation since the hard disk drive gave me problems. I sent the drive out to be repaired, and the technicians could not find anything wrong with it. Since I did not want any data to be destroyed, they did not try reformatting it, but they told me that they had no trouble reading data from the tracks.

I just have not had the time to reinstall that drive. For the moment I am still running on a replacement drive with only the most basic software (and I mean basic -- I don't even have BGii on it yet!). It may be that my house has been afflicted by malicious gremlins. My modem failed at about the same time as the hard disk. I finally negotiated with the manufacturer the terms under which I could return it for repair. Before doing so, however, I decided to give it one more try. It worked perfectly! Are there problems that go away when an instrument is powered down for a week or two that would not go away in one day?!

During the time of these troubles, Murphy really had the upper hand. Although there were some files on the hard disk that I should have backed up but did not, there were some very important files related to these TCJ columns that I had backed up to a floppy. However, when the hard disk failed, the backup floppy vanished. Now, even after a month, it has still not turned up. If I'm lucky, the hard disk will work when I reinstall it, and the floppy will then reappear, in plain sight on my desk where I know I kept it. If all that happens in the next two months, I hope to have a further installment on my efforts to patch up MEX-Plus to add some new features and correct some bugs. Cross your fingers for me!

Jay Sage has been an avid ZCPR proponent since the beginning and is best known as the author of the current version of the ZCPR3 command processor and of the ARUNZ alias processor. He has been running Z-Node #3 (617-965-7259, MABOS on PC-Pursuit, pw=DDT) since 1983. Jay is also the Z-System sysop on GEnie, where his mail address is JAY.SAGE. Stop by and chat live at the Wednesday real-time conferences (10pm Eastern time), especially at the first one each month, which Jay hosts.

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 and image processing. He can be reached via Internet as SAGE@LL.LL.MIT.EDU.

You can also write to him at the SME address or, if you prefer voice contact, you are welcome to try 617-965-3552 (10:30-11:15 at night is most likely to find him at home), but please don't call Friday evening or Saturday.

Recapitulation

I'd like to start the ZMATE discussion by reminding you of the four ways in which macro commands can be used, as described in my previous column.

The most common way to execute a macro is by pressing one of the so-called instant-command editing keys, such as the keys that move the cursor left, right, up, and down by various amounts. These keys are bound to a set of ZMATE internal functions, most of which are implemented in the macro language. In the original PMATE editor, from which ZMATE was derived, these functions could not be changed; in ZMATE the user can patch in new macro functions for the internal commands. Source code is provided in the file INTMACRO.Z80, and it is a relatively simple procedure to edit it and patch it into the distribution version of ZMATE.COM.

An additional set of macro functions can be defined in the "permanent macro area" or PMA. There, macro strings are associated with single-character names. These macros can be invoked by name using other commands in the macro language, and those with a specific range of names can be bound to keystroke sequences.

Third, when the editor is in command mode, the user can enter a temporary macro sequence directly on the command line at the top of the screen. The user's command line is stored in a special text buffer called the command buffer.

Finally, the contents of any of the ten numbered editing buffers can be interpreted as a macro sequence and executed. In other words, any of the auxiliary editing buffers can function like the command buffer.

ZMATE Macro Commands

Now let's look at some of the macro commands that ZMATE recognizes. We can't cover all of them this time. We will start with some of the simpler ones and then will cover a few of the more sophisticated ones. We will then look at some of ZMATE's built-in functions so you can see how macros are used to implement them.

The ZMATE language is very compact to save space, typing, and time. Most of the commands use only a single letter; some are two characters; a few are three-characters long. To the extent possible, the letters for the commands are chosen to be mnemonic in some way. There is a bit of a learning curve, but it does not take

too long to get the hang of them. I made a two-page crib sheet that I occasionally consult to remind myself of some of the more obscure ones.

Cursor Motion

The most basic commands are those that move the cursor around within the text in an editing buffer.

There are four macros that move the cursor in units: 'M' moves by characters; 'W', by words; 'P', by paragraphs; and 'L', by lines. It is pretty clear what a character is. The one thing that might not be obvious is that hard carriage returns are treated as a character in the text. The cursor can be positioned on one, and it can be deleted to close up two lines. ZMATE does not generally use linefeeds. Carriage returns in the editing buffer are converted to carriage-return/linefeed pairs when the file is written out to disk or printed.

Words are contiguous groups of letters and numbers. Special characters – such as periods, quotes, asterisks, dollar signs, etc. – and control characters – including spaces, tabs, and carriage returns – separate words. Paragraphs are terminated by hard carriage returns. ZMATE supports a mode, called "format mode," in which lines wrap automatically at the right margin as with wordprocessors. The apparent carriage returns at the ends of the wrapped lines are called soft carriage returns. The "P" macro ignores those carriage returns. When ZMATE is in format mode, the hard returns are visible as '<' characters in highlighted video.

Each of these move-by-unit macros can take a signed numerical prefix. Such a prefix is one of the two ways arguments are passed to ZMATE commands. For the cursor-motion commands, if there is no prefix, "1" is assumed; if the prefix is simply "-", then "-1" is assumed.

Positive prefixes move the cursor forward (to the right and down) in the text; negative prefixes move it back (left and up). For example, "3W" moves the cursor to the beginning of the third word after the one where the cursor is now, while "-2W" moves the cursor to the beginning of the second word before the one in which the cursor is presently located.

A "0" prefix moves to the beginning of the current unit. For this purpose, word separator characters are considered to be a part of the word they follow. Suppose we have the text

```
ONE...TWO:;THREE
```

with the cursor sitting on the 'W' in 'TWO'. "-W" will put the cursor on the 'O' in 'ONE'; "0W" will put it on the 'T' in 'TWO'; and "W" will put it on the 'T' in 'THREE'. Where would the cursor have ended up if it had started on either the colon or semicolon between 'TWO' and 'THREE'? Answer: in the same places. Those word-separator characters after 'TWO' are treated as if they were a part of the word they follow.

What do you think "0M" does? Well, it does nothing to the cursor. Nevertheless, it is not at all a useless command. You see, the numerical prefix is not always given as a literal number. Sometimes it is a calculated quantity, as we will see later. If we compute how far we should move, and the answer is zero, the macro should work.

There are a few absolute (unit-less) cursor motion macro commands. The command "A" moves the cursor to the first character in the edit buffer, while "Z" moves it just past the last character, namely to the place where the next character would be inserted. ZMATE supports virtual memory in its main editing buffer, called the 'T' or text buffer. Files that are too big to fit in memory are paged in and out, either manually or automatically, as desired. The macros "UA" and "UZ" go to the beginning and end of the entire file (think of 'U' as 'UNLIMITED'). If required, the file will be paged from disk.

The "QX" command is one of a whole family of "Q" commands, a couple of which we will see this time. It takes a numerical prefix and moves the cursor to that column in the current line. The columns are numbered beginning with '0'. As is frequently the case, an absent prefix is taken as "1". Not surprisingly, the prefix must be non-negative. What happens if you violate this restriction? Does your file get trashed and your whole disk wiped out? No, ZMATE tries to interpret the number as a positive number, which, of course, is larger than the maximum column allowed (typically 250). The result is a beep from the terminal and a strange positioning of the cursor.

Perhaps this is a good time for a general comment about numbers in ZMATE. Numbers are stored as words (two bytes, or 16 bits). Such numbers can be interpreted either as positive numbers ranging from 0 to 65535 or as signed numbers with a positive range 0 to 32767 (7FFF hex) and a negative range from -1 (FFFF hex) to -32768 (8000 hex). Numbers are also used to represent Boolean logical values. False is represented by '0'; true, by '-1'. When a function only accepts Boolean values, then any number other than '0' is taken as true.

The "Q-" macro determines whether ZMATE will display numbers as signed or unsigned. Following the Boolean convention, "0Q-" turns off the display of negative numbers, while "-1Q-" or just "Q-" turns them on. If you enter a number all by itself as an interactive command line macro, its value in the current display mode will be shown after the "arg=" status message on the top line of ZMATE's screen. If you enter

```
0Q- -1$$
```

you will see "arg=65535" on the status line. If you enter

```
Q- -1$$
```

you will see "arg=-1".

Now lets see how some of ZMATE's built-in functions are defined. Look at Table 1. Functions 3 and 6 are especially interesting. Macro commands can be combined by writing them in sequence with or without spaces or tabs between the individual commands. The spaces in the two examples above were put there only to make the commands easier for you to read; ZMATE can read them just as well without any spaces.

Function 3 implements what an ordinary person would think of as "move back one word." If the cursor is currently somewhere other than the beginning of a word, then the cursor is supposed to move back to the beginning of the current word. If it is already at the beginning, then it should move back to the beginning of the previous word. As we noted earlier, "-W" would move back too far in the former case. Moving back one character and then to the beginning of the current word does the trick. See if you can figure out why – and why function 6 is implemented as it is.

Text Deletion and Insertion

There are only two deletion commands: "D" deletes characters; "K" deletes (kills) lines. They take a numerical prefix with the usual default values. Deletions to the right – positive prefixes – start with the character under the cursor. Deletions to the left – negative prefixes – start with the character to the left of the one under the cursor. Thus, the command "K" deletes all characters on a line to the right of and including the cursor, while "0K" deletes all characters on a line to the left of the cursor.

"0LK" deletes the entire current line. Line deletions to the right, by the way, include the carriage return at the end of the line.

The basic insertion command is "I". It can be used in two forms. If it has a numerical prefix, then the character with that ASCII value is inserted before the character under the cursor, and the cursor remains on the character it was on before (i.e., after the new character).

The prefix value is interpreted as a positive number modulo 256. This form of the insert command can be used to insert some characters that cannot be inserted by typing (e.g., characters with their high bit set) and some that cannot be inserted using the second form of the insert command that we will look at shortly (e.g., the escape character).

Some character values (e.g., 0 and some special values that ZMATE uses for specially formatted text) cannot be used. Characters with the high bit set can be put into a file and can be written out to disk, but when such a file is read back in, the high bits will be filtered out.

The second form of the "I" command illustrates the general syntax for string arguments in the macro language. These come after the macro command and are terminated or delimited by escape characters. The command

```
Istring of text$
```

will insert the string of characters following the 'I' and up to the escape character, which is indicated here by the dollar sign. Some commands, as we will see shortly, take more than one string argument.

Another type of insertion is replacement, which uses the "R" command. It is much like the "I" command except that the new characters replace those under the cursor and to the right. For example, "65R" changes the character under the cursor to an 'A' (ASCII value 65), and the command "Rtest$" replaces the character under the cursor with a 't', the next character with an 'e', and so on. The cursor ends up on the character after the last one replaced.

The "\" command converts its numerical prefix into the text representation for the number and inserts it into the text before the cursor. Leading zeros are not included.

It might be appropriate to mention at this point that ZMATE in not limited to working in decimal radix. There are macro commands to set a radix to values between 2 and 16. Decimal is standard and will be assumed in all our examples. However, the radix in which input numbers, such as command prefixes, are interpreted and the radix in which output numbers are displayed can be changed independently.

For example, "8QI" sets the input radix to octal. "QI" (no prefix) will always set the radix back to decimal. This is awfully handy when you don't know what the current input radix is. After all, "10QI" leaves the radix unchanged. Do you understand why? "16QO" will set the output radix to hexadecimal, provided the input radix was decimal when this command was processed. If it was octal, the output radix would become 14, the octal value of '16'.

I recommend a trick for entering constants so that expressions will not be misinterpreted if the radix is changed. We have been careful to implement all built-in functions using radix-invariant expressions. The special ZMATE operator double-quote converts the character following it to its ASCII numerical value. To set the output radix to hexadecimal, for example, we could use the command

```
"^PQO
```

Here we use '^P' to represent control-P. ZMATE allows control characters to be entered by typing a caret followed by the letter. The older PMATE editor also displayed control characters this way, but ZMATE shows just the character in highlighted video. The value of control-P is always 16, no matter what the input radix is. If you cannot enter a single character with the desired value, you can use arithmetic to get the value. We could have written the above command as

```
("Q-"A)QO
```

or simply

```
"Q-"AQO
```

since 'Q' is 16 characters higher than 'A'.

Now back to insertion macros. There is one more. "QH" inserts a block of blank spaces (perhaps the 'H' stands for 'hole') in the text before the cursor. As usual, it takes a numeric prefix. The three commands below are all equivalent.

```
10QH
I          $
" I" I" I" I" I" I" I" I" I" I
```

Besides being more compact, the "10QH" form will generally be faster. It tells ZMATE in advance how much space to open up, and the entire insertion, which may involve moving the text after the cursor, can be done in a single operation.

Search and Search-and-Replace Macros

ZMATE's string searching command illustrates a syntax in which a command takes both a numerical prefix argument and a string argument. The general form of the search command is "nSstring$". The numerical prefix can be positive or negative. With a positive value, the search is performed in the forward direction; with a negative value, the search moves back toward the beginning of the buffer.

The number tells ZMATE the maximum number of lines to search through. The current line to the left of the cursor is line 0. The current line to the right of the cursor is line 1. Thus "0Stest$" will search for 'test' in the part of the current line to the left of the cursor. "-4Stest$" will search in the current line to the left of the cursor plus the four lines before that. "1Stest$" will search the remainder of the current line beginning at the cursor and working to the right.

The default prefix values are different with the "S" command from what we have seen before. If just a sign is given without a number, then the entire remainder of the text buffer in the given direction will be searched. Thus a plus sign — or no prefix at all — defaults to the largest positive number (32767); a minus sign alone defaults to the largest negative number (-32768).

A variant of the "S" command is "US" (unlimited search). Like the commands "UA" and "UZ", it will perform scrolling of a file to and from disk in order to search the entire file. The "US" command does not accept a numerical prefix; it would not make sense, since it searches the entire file. It does accept a sign to indicate the direction of the search.

The search-and-replace, or change, commands "C" and "UC" are quite similar except that they take two string arguments. The first is the search target; the second is the text to replace the search target with. If I executed the command

Table 1. Macros used to implement some ZMATE built-in cursor motion functions. The table lists the function number and describes what the function does. If the function has a standard binding, it is shown. A caret prefix indicates a control character.

fn #	description	key	macro
1	to end of buffer		Z
2	to previous char	^G	-M
3	to previous word	^O	-M0W
4	to next character	^H	M
5	to next word	^P	W
6	up one line	^Y	-M0L

```
-Ctest$examination$
```

at this point in the text, the instance of 'test' three paragraphs back would be removed and replaced by 'examination'.

Some special characters can be used in the search string in the "S" and "C" commands. A control-E represents any character ('E' as in 'EVERY'). Thus "Ste^Et$" would find either 'test' or 'text' (or lots of other things). A control-S ('S' as in 'SPACE') represents any white-space character, namely space and tab. Control-W represents any word-separation character, so that "Sa^Wb$" would find 'a-b' or 'a b' or 'a/b' (but not 'axb' or 'a//b', which has two word-spacing characters between the 'a' and the 'b').

A control-N matches any character except the one that follows it ('N' as in 'NOT'). "Ste^Nst$" will stop on 'text' and 'tent' but not on 'test'. Just in case you need to search for one of these special characters, control-L ('L' as in 'LITERAL') causes the character following it to be treated literally. Thus "S^L^N$" will search for a control-N, and "S^L$$" will search for an escape character.

These special characters do not implement string searching as powerful as that in Unix GREP or in Bridger Mitchell's Jetfind, but they cover the most common situations. Other macro capabilities in ZMATE would make it possible to implement full GREP search rules, but such a macro would not be very fast.

Variables

A little earlier we alluded to the fact that ZMATE has numerical variables. It can perform arithmetic operations, bitwise Boolean operations, and logical comparisons with literal numbers and values of variables.

Listing all the variables would take up too much room here, so I will describe just a few of them to give you some idea of the kind of information available to a ZMATE macro.

Almost all ZMATE variables are represented by an '@' sign followed by a character that designates the variable name. None of them take any arguments, except for two that take a string argument.

Some variables tell where the cursor is located. "@C" returns the number of the character (its position in the text) counting from the first character in the buffer. "@L" gives the absolute line number (i.e., counting from the beginning of the file, even if some of it has been scrolled out to disk) of the line containing the cursor. "@X" reports the column number.

Some variables give information about the way the page is set up. "@Y" and "@W" give the left and right margins, respectively. "@Z" gives the column number of the next tab stop.

One of the most important variable commands, if not the most important, is "@T". It returns the ASCII value of the character under the cursor. This information is critical to intelligent text processing.

There are also ten user variables numbered from 0 to 9. The "V" macro is used to set values into them, and the values can be retrieved by '@' followed by the variable name. Full memory access is provided by the variable "@@", which returns the contents of memory at the address stored in user variable 9. The macro "Q!" stores the value passed as a prefix into that address. Thus "@@" is the PEEK function and "Q!" is the POKE function. "@P" returns the absolute address where the character under the cursor is presently stored in memory.

The macro command "Gprompt$" displays the prompt string on the command line and waits for the user to enter a keystroke. The macro "@K" ('K' as in 'KEYSTROKE') then returns the ASCII value of the user's response. This provides the hook for interactive operation.

As you can see, ZMATE gives you the basic tools for doing just about anything. It may take some effort, but there isn't much that is impossible. There are a few variables I can think of that are missing. For example, ZMATE can get a disk directory or ask if a file exists, but it cannot find out how much space is left on a disk (though it can find out how much memory is free). It also cannot determine the drive or user number it is logged into or that is associated with a file it is editing.

Flow Control

A programming language is pretty much useless if it has no way of making decisions. That's why the flow control package (FCP) is so important in the Z-System. We have already shown you the kind of information that is at ZMATE's disposal. Now we will show you how that information is used to make decisions.

In one way or another ZMATE implements all the major flow control forms: repetition, if-then-else, do-until, and goto. Blocks of macro code are formed by enclosing them in matching pairs of either square or curly brackets. For most purposes, the two forms of bracket are equivalent.

For the flow control formats, we will use 'n' and 'm' to represent macro expressions that return numerical values. A numerical value of '0' has a Boolean value of 'false', while a numerical value of '-1' has a Boolean value of 'true'. Three dots are used to indicate an arbitrary sequence of macro commands, which may, themselves, include flow-control constructs (nesting of flow control is allowed to 15 levels).

The general form of the repetition macro is

```
n[...m]
```

The value 'n' is the repeat count. In general, the macro commands inside the brackets will be repeated 'n' times. If 'n' is '0' or 'false', they will be skipped. If 'n' has the special value '-1', it will be interpreted as a Boolean, and the block will be executed only once. Other negative values will be interpreted as their corresponding positive values. If the prefix 'n' is omitted, the block will be repeated indefinitely (well, actually some 65534 times, but who's counting).

After each iteration of the block of macro commands and before going back to the beginning, the value of 'm' is checked. If its Boolean value is 0 (false), iteration continues; if it is nonzero (true), control passes over the ending bracket and continues with any following commands. Thus 'm' constitutes the 'until' test for the DO-UNTIL construct.

If the numerical/Boolean expression 'm' is omitted, then the value of the special ZMATE error flag is used. This flag can also be evaluated explicitly as "@E". Certain commands set and clear this flag. For example, a search command ("S") will set the error flag if it could not find the designated search string. A cursor motion command that tries to take the cursor beyond the bounds of the text will also set the flag.

There are cases where iteration loops seem to terminate prematurely. This is usually because of the default use of the error flag as the 'until' test. One way to get around the problem is to end the block with the form '0]'. This ensures a false 'until' test and continued iteration.

The general form above includes basic condition processing. Full if-then-else processing is implemented by the form

```
n[...][...]
```

If 'n' is 'false' (i.e., has a value of zero), then the first block will be skipped and the second block executed. If 'n' is 'true' (in this context, nonzero), then the first block will be executed and the second block skipped. This form is identified by the touching closing and opening brackets. If you have two repeat blocks in a row, use '] [' with a space instead of '][' to prevent ZMATE from interpreting the macro as an IF-THEN-ELSE test.

There are several special commands for terminating or moving around within an iteration block. The 'exit' macro "n_" will immediately exit the loop and continue after the next closing square bracket. The 'next' macro "n^" will immediately go back to the closest preceding opening **square** bracket and start a new iteration. This is the only case in which the kind of bracket makes a difference. Because of this difference, it is generally effective to use curly brackets for if-then-else tests and square brackets for iteration constructs.

One extra word of caution. ZMATE is not smart enough to distinguish brackets in string expressions from those used in flow control constructs. Be very careful whenever you have string expressions containing square or curly brackets; they may confuse flow control macros.

ZMATE has a goto function. The syntax is "nJx", where 'x' is a single-character label (any character can be used). If 'n' is true, ZMATE will scan the macro from the beginning for a marker of the form ":x".

Finally, if 'n' is true, the command "n%" will terminate execution of the entire macro and return control to any macro that called this one as a subroutine or to the user.

Some Final Examples

Table 2 shows some more examples of built-in functions. These macros use some of ZMATE's testing powers. Function 0 moves the cursor to the first character in the buffer unless it is already there, in which case it moves it to the bottom of the buffer. The expression "@C=0" performs a logical comparison of the value of '@C', the number of the character under the cursor, and 0. If the cursor is on the first character in the buffer (remember, numbering starts at 0), then this expression will be Boolean 'true' (arithmetic -1), and the first macro block, "Z", will be performed. Otherwise the second block, "A", will be carried out.

In looking at that macro just now, I realized that it could be shorted slightly to

```
@C{A}{Z}
```

Table 2. Macros used to implement some additional and more complex ZMATE built-in functions.

fn #	description	macro
0	toggle top/bottom of buffer	`@C=0{Z}{A}`
24	character left geometric	`@X>0{@X-1QX}`
38	toggle case of character	`@T>"@&(@T<"[){@T+" R%}` `@T>"`&(@T<"{){@T-" R%}` `M`

Here we treat "@C" as a Boolean value. If it is zero (we are at top of buffer), it will be interpreted as 'false' and "Z" will be executed. The version we used is easier to read but costs two extra characters.

Function 24 moves the cursor left one character geometrically. The command "-M" moves the cursor back one character absolute, and will back up to the previous line if the cursor is presently at the beginning of a line. The geometric motion macros work on the column number. Function 24 first checks to see what column the cursor is in presently. If the column number is greater than 0 ("@X>0"), then the macro computes the column number one to the left ("@X-1") and passes that value as a prefix argument to the command "QX".

Function 38 is still more complicated. It toggles the case of the alphabetic character under the cursor. The macro has three independent parts, the first two of which are conditionally executed. The conditionals are complex expressions involving two parts combined by a Boolean operator.

In the first one, `@T>"@` tests to see whether the character under the cursor has an ASCII value greater than that of '@', which is one less than 'A'. If ZMATE had a greater-than-or-equal-to test, we could have written something like `@T>="A`, but this is not allowed. The second part, `@T<"[`, tests to see if the character is 'Z' or less. These two tests are combined by the Boolean 'and' operator '&'. The result will be true if the character is an upper case letter.

If the result is true, the value of the space character (32 decimal, but radix-invariant when expressed this way) is added to the current value to make the corresponding lower case character. This value then replaces the existing character. Finally, "%" is executed to terminate the macro.

If the first conditional is false, the macro continues with the second one. It tests to see if the character is in the range 'a' to 'z'. If it is, 32 is subtracted from the present value to make the corresponding upper case character, which then replaces the existing character. Again, the macro is terminated with "%".

If the character is not alphabetic at all, the macro continues with the final line. This simply moves the cursor to the next character without making any change.

Next time we will continue the discussion of ZMATE's command language, and, with any luck, I will have recovered my MEX patches and will be able to present them as well. •

Embedded Systems for theTenderfoot

Using 8031 Interrupts

by Tim McDonough

The topic for this month's Embedded Systems for the Tenderfoot series is the use of interrupts with the 8031. You know what an interrupt is right? It's when you're working on your latest project and your spouse stops you to cut the grass or haul the kids to a ball game. You stop whatever your doing, jot down a quick note or two so you don't forget what you were about to do next; run the kids to their ball game, return home and pick up where you left off.

Don't laugh. If you were an 8031 your spouse's yell would have been an external interrupt. Jotting down your notes would have been saving the return address on the stack. When you ran to the car took the kids to their game, and watched them win (hopefully), you were executing the interrupt service routine whose last act was to return you home and plop you in front of your project to continue working. See, I told you knew what an interrupt was!

Now of course as with most things in life, 8031 interrupts aren't always as simple as running the kids to the ball game. For one thing there are several different types of interrupts—external interrupt, timer interrupts, serial port interrupts, etc. This time, I'll look at a simple timer interrupt. In future installments I'll present projects that use the 8031's external interrupt lines and another that uses interrupt driven serial communications.

There are a few basic things you need to know about interrupts before you can make use of them. First, the 8031 must be told to look out for the interrupt you intend to use. This is done by enabling the specific one you're interested in using in your system.

Next, you need to be aware of what the 8031 will do when the interrupt occurs. It turns out that for each particular type of interrupt, the 8031 has a special memory location that it will jump to when the interrupt occurs. This location is often called the interrupt vector address. In the 8031, and most other microcontrollers, all of the interrupt vectors are located near one another in memory. It is common practice, since little address space exists between each vector, to place a JMP instruction at this address that will jump to the actual code you want to execute; hence you will often see the group of interrupt vectors referred to as the "jump table".

The code that the jump table points to is usually called the "interrupt service routine." This, as it name implies, services or responds to the interrupt. Depending on your own application this routine may cause a counter to be incremented that is part of a clock, perform an analog to digital conversion, update a display, flash a indicator light, count an external event, read a keyboard, etc. Interrupts are extremely powerful and their use is nearly essential in a "real time" control system.

When timer interrupts are enabled, the 8031 will generate an interrupt whenever a timer/counter overflows from FFFF(Hex) to 0000(Hex). By making a calculation based on the clock speed used for the 8031 and using the resulting number as the timer reload value, it is possible to generate an interrupt at precise time intervals. If your application then counts the interrupts, you can use the timer as a system clock so you can perform certain actions every so often, regardless of what else the computer may be doing.

Figure 1 shows a minimal 8031 circuit with a transistor driver and LED added to demonstrate timer interrupts. This is the same circuit I used in the first article in the series when I presented the XOR.ASM program. (See the sidebar for a description of this interface.)

Listing 1 is the source code for FLASH.ASM. It demonstrates the use of a timer interrupt to perform a task at regular intervals. In this example an LED flashes. Other uses might be checking the status of a switch regularly, taking an analog to digital converter sample, etc.

FLASH.ASM uses equate statements to establish the location of the I/O pin used to drive the LED interface (See the interfacing sidebar) and sets the number of interrupts that must occur before the state of the LED is toggled. Next the program execution skips over the jump table for the interrupt vectors.

The example uses Timer 0 of the 8031 to generate the interrupts. Code must be located at location 0B (Hexadecimal) to handle the interrupt. Although it is not the case here, the interrupt service routine might be fairly long. It is fairly common then to place a jump at location 0B (Hexadecimal) to the address of the routine that will service the interrupt.

The setup portion of the program is slightly different than the previous programs I've presented. Timer 0 is set up to be a free running timer and turned on and the delay value is loaded into register R0 which will be used as a counter. The last two lines of code may be unfamiliar. The Timer 0 interrupt must be enabled before it will have any effect and the 8031 must also be told to pay attention to all the possible interrupts.

The main program of FLASH.ASM is an endless loop. The 8031's NOP instruction does nothing except waste some time. Your own application might perform some serial communications or whatever. The idea is that no matter what the main program is doing, a Timer 0 interrupt will suspend that action and service the interrupt before continuing.

The BLINK subroutine is where the work gets done. Each time the 8031's Timer 0 rolls over from FFFF (Hexadecimal) to 0000

Interfacing Microcontrollers

Whatever the microprocessor, the "computer" of an embedded system can range from exciting to ho-hum, depending on your point of view. But let's face it, in most embedded systems interfacing the "computer" to the real world is where the rubber meets the road, so to speak.

Several articles, indeed entire books, could be devoted (and have been) to interfacing microprocessors. I'm not going to attempt to cover the topic in depth but a few general bits and pieces of circuitry may help get your project off the ground a bit sooner. Some of these interfaces have been mentioned, but not described, in previous columns. A few others are presented due to popular request by those of you who are venturing forth into the world of embedded systems.

The following descriptions are of general purpose interfaces. No one circuit design is optimal for every solution and depending on the nature of your system and its interface to the real world, a poor design can lead to injury or even death to the user. That doesn't mean you shouldn't build embedded systems. It does mean you should work within your own limitations and abilities. Stick with applications that you understand. I doubt that anyone's first project was a missile guidance system or a heart pacemaker.

Push Button Switch

Figure A shows the electrical interface for a pushbutton switch. The switch might actually be a button that the user presses or it might be a cabinet interlock switch, burglar alarm contact closure, etc.

S1 is a momentary contact, normally open switch. The 10K ohm resistor, R1, is called a "pull up" resistor. When the switch is open, R1 pulls up the voltage on the microprocessors input pin to +5 volts indicating a value of "1" when that bit is read. When S1 is pressed, the input pin is held at ground potential and a value of "0" is represented when the bit is read. R1 keeps the 5 volt power supply from being shorted to ground when the switch is closed.

"Programming" Jumpers

In many microprocessor circuits, options that are infrequently changed are selected via programming jumpers or DIP switches. You've probably encountered these when setting up a new printer or modem. Electrically they can be the same as the pushbutton switch except that S1 is replaced with a small removable jumper or a small switch on the printed circuit board. During the initialization portion of the system software, the switch settings are read and the users preferences loaded into the processor.

There are endless uses for programming jumpers. Some often seen examples are baud rate selection for serial ports, whether or not a device should send a carriage return or a carriage return/line feed combination, etc.

Relay Output

Several readers have inquired about my "to relay #1" description in the Communicating With The Real World column. I apologize to anyone who's been chomping at the bit to try their hand at parsing commands using the system I described.

Driving a small relay is quite similar to the LED indicator except that I normally seem to end up with relays that require 12 volts DC to operate and the logic components can only provide 5 volts at a very low current.

The transistor circuit shown in Figure B will do nicely for a relay to control small wattage AC circuits. The interface circuit between the relay and the microprocessor is required because the amount of current that the 8031 and most other microprocessors can source or sink. The typical 8031 part can sink or source 4 LS series TTL loads (about 500 micro Amps.)

The PNP type transistor shown in Figure B acts as an electronic switch. When Q1's base is negative with respect to its emitter, the transistor is turned on and a path is provided for current to flow through the coil of relay K1. Note that in this particular case 12 volts is being supplied to the relay.

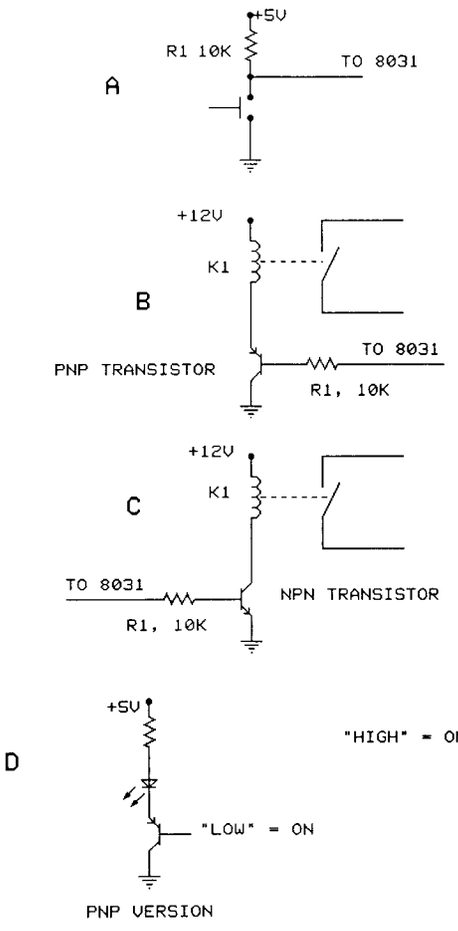
The type of transistor chosen for the interface (PNP or NPN) may be influenced by the design of your project. After reset all port latches of the 8031 have 1s written to them. In the case of Figure B this means that K1 will be de-energized after a reset since Q1 will be in the off condition. When you toggle the output bit to a "0" the relay energizes; change it back to a "1" and the relay is turned off.

Figure C shows an interface circuit using an NPN type transistor. This circuit, because an NPN transistor is turned on when the base is positive with respect to the emitter, will energize the relay as soon as the system is reset. Similarly, it would be turned off by writing a "0" to the output pin.

The choice of driver circuitry may be very important depending on your application. If the relay is controlling a piece of machinery, it may make a big difference whether or not it is powered up after the microprocessor is reset. You probably wouldn't want your microprocessor controlled table saw to start spinning without your software specifically telling it to do so now would you?

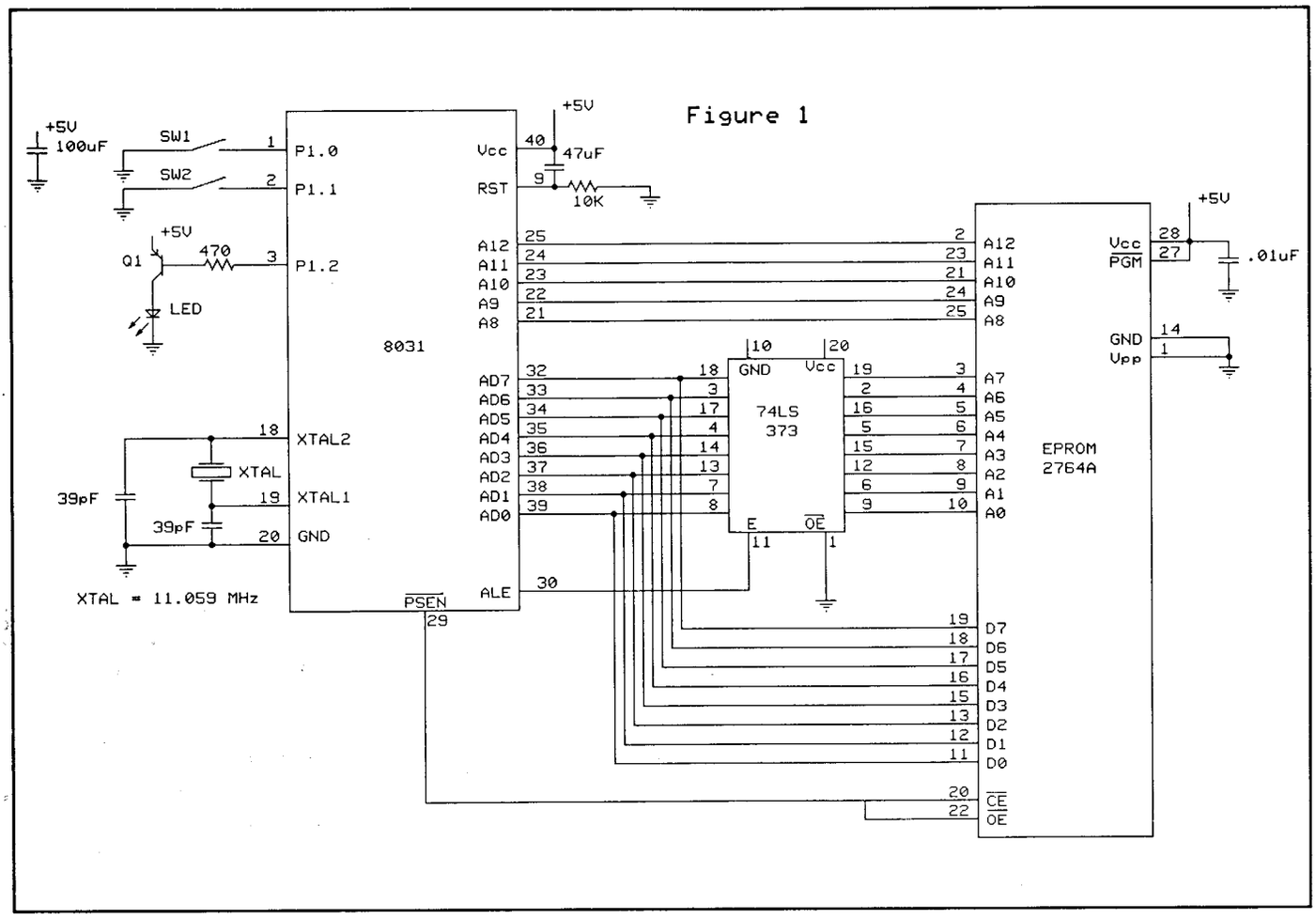
LED Interface (transistor driven)

Many embedded systems feature LEDs to indicate the status of the system. An LED is driven much the same as a small relay as shown in Figure D. In this circuit the resistor R1 limits the current through the LED to about 15 milliamperes. The value of R1 is chosen based on the recommended operating current for the LED.



Listing 1

```
; FLASH.ASM
; September, 1990

; Tim McDonough
; Cottage Resources Corporation
; Suite 3-672, 1405 Stevenson Drive
; Springfield, Illinois 62703
; (217) 529-7679

; This program demonstrates a method of using the 8031 timer interrupt
; force an event to occur at regular intervals. It was developed as a
; part of the "Embedded Systems for the Tenderfoot" series published by
; The Computer Journal.


;       'DELAY' is the number of timer interrupts that must occur before
;       the state of the output LED is toggled. The LED driver circuit
;       is attached to Port 1, bit 0 of the 8031 microcontroller.

        .EQU    DELAY,H'05      ;# of interrupts before LED toggles
        .EQU    LED,P1.3        ;LED is at Port 1, Bit 0


;       The code is assembled to be located starting at location 0. Then
;       execution jumps past the interrupt jump table.

        .ORG    H'0000
        AJMP    SETUP

;       Any time a Timer 0 interupt occurs, the program jumps to
;       location 0B (Hex). This location contains a jump to the interrupt
;       service routine that toggles the state of the output pin.

        .ORG    H'000B          ;Timer 0 interrupt vector
        AJMP    BLINK

;       The system is initialized by clearing any pending interrupts
;       and loading Register 0 with the delay value declared elsewhere
;       with an equate statement. Timer 0 high and low bytes are
;       initialized to a known state and Timer 0 is set to free run.
;       The timer and it's interupts are enabled.

SETUP:  CLR     EA              ;Clear all interrupts
        MOV     R0,#DELAY       ;Initialize Register 0
        MOV     TL0,#00         ;Initialize the low and high bytes of the
        MOV     TH0,#00         ; counter/timer
        MOV     TMOD,#H'21      ;Timer 0 mode 1, Timer 1 mode 2
        SETB    TR0             ;Turn on Timer 0
        SETB    ET0             ;Enable timer 0 interrupt
        SETB    EA              ;Enable all interrupts

;       "MAIN PROGRAM"
;       The main program in this example does nothing. Except for those times
;       when an interrupt is being serviced, the program executes a very
;       boring loop.

LOOP:   NOP                     ;Do something here if you like...
        NOP                     ;
        NOP                     ;
        NOP                     ;
        SJMP    LOOP            ;Spin forever

;       "BLINK" -> INTERRUPT SERVICE ROUTINE - TIMER 0
;       Timer 0 runs continuously and generates an interrupt each time it
;       rolls over from 0 to all 1's. This routine services that interrupt
;       by decrementing a counter (register 0) and complementing the output
;       line that drives the LED each time the counter reaches 0. If the
;       value of DELAY equals 5, then the output state will change once every
;       5th Timer 0 interrupt.

BLINK:  DJNZ    R0,NEXT         ;Jump to NEXT if R0 <> 0
        CPL     LED             ;Toggle the LED
        MOV     R0,#DELAY       ;Reload the delay value into R0
NEXT:   RETI                    ;Return from the ISR

        .END
```



(Hexadecimal) an interrupt is generated causing the AJMP instruction at location 0B (Hexadecimal) to jump to the BLINK subroutine.

The DJNZ instruction tells the 8031 to decrement the value contained in the referenced location and jump to the memory address given if the value is not zero. Since R0 has been loaded with our delay value, 5 Timer 0 interrupts must occur before the status of the LED is altered.

Once the value in R0 equals 0, the complement instruction (CPL) changes the state of the output line to the opposite of its current value. A low output will be driven high and a high output will be driven low.

Finally the subroutine ends. Notice that the Return from Interrupt (RETI) instruction is used instead of the normal Return (RET). This instruction causes program execution to be resumed after the point at which the interrupt occurred and restores the interrupt logic to accept additional interrupts.

I've intentionally cut this month's installment short so a sidebar on interfacing simple devices like relays, switches and LEDs could be presented as well. There is a lot more to 8031 interrupts than I've presented and you'll be hearing more about them in the future.

Several readers have requested a useful beginner's project that demonstrates an actual application of the 8031. I've had a "not quite finished" data acquisition system on the back burner for a couple of months now. It's been dusted off and I intend to present it as the first real project in the next issue. The system provides 8 analog to digital converter channels in a 8031 based system that communicates with a host computer via the serial port. The serial interface allows it to be used with nearly any computer so no matter whether your machine is a PC, Macintosh, CP/M box or whatever, you can add an A/D interface and collect some real world information. •

T-1

What it is and Why You Need to Know

by Richard Rodman

The little boy closely examined the metal box the old man wanted to sell. He wiped the dust from its plastic front, revealing a row of dark circles. "What is it?", the little boy said.

The old man winked. "In the old days," he said, "you used to communicate over the telephone system with an analog modem. This one was considered quite fast." He leaned forward. "This one ran *ninety-six hundred baud.*"

The little boy's face wrinkled up. "Nine point six k-baud?" Then, after a pause: "Does it have any good parts in it?"

This scenario is actually not too far off. By now, most of you have probably heard of T-1. This article will attempt to explain what T-1 is, and why it's so important.

High-Speed Digital Networks

When I first started working in the telephone business, I was astonished to learn that the United States already has a coast-to-coast network with high-speed, fully digital connections running at what were to me incredible speeds – 1.5 to 45 million bits per second.

Back in the seventies, AT&T was using microwave and fiber optics to carry telephone conversations. These were usually sent in multiplexed analog form. However, to reduce distortion and to allow greater multiplexing, Bell Labs began developing a digital transmission standard. Higher and higher speeds were attained, leading to the release of the 1.544 megabit-per-second rate in 1981. This standard has become known as T-1. Today, there is a T-1 Standards Committee, made up of telecommunications industry experts, which monitor the standard and have guided it into new technologies.

Today, T-1 and T-1-like circuits span the globe with relatively error-free multi-megabit-per-second voice and data communication over fiber-optic, microwave, satellite, and copper links.

Digital Services

T-1 is actually the second level of a four-level hierarchy of Digital Services. The first, basic level is called a DS-0, which corresponds to a single voice channel digitized at 8000 8-bit samples per second, or 64 kb/s. T-1, also called DS-1, is 24 DS-0's placed end-to-end. The package of 24 8-bit samples is preceded by a framing bit (similar to a start bit), for a total of 193 bits making up what is called a "frame." Since there are 8000 frames every second, the data rate is 1.544 mb/s.

Digital Services

DS-0: 64 kb/s
DS-1: 1.544 mb/s (24 DS-0s)
DS-2: 6.312 mb/s (4 DS-1s)
DS-3: 44.736 mb/s (28 DS-1s)

The next levels are DS-2, which carries four multiplexed DS-1s, and DS-3, which carries 28 multiplexed DS-1s. DS-3s are typically carried over long-haul fiber optics. Because the DS-1s are multiplexed rather loosely into the DS-3, the individual DS-0 channels cannot be broken out of the DS-3 directly. Instead, the DS-1s have to be demultiplexed by a M13 multiplexer. Then, the DS-1 (T-1) can be sent to a channel bank or T-1 multiplexer to connect to analog phone lines or digital equipment.

There is a movement afoot to convert the DS-3s to a synchronous form called SONET, which will allow direct conversion between DS-0 and DS-3. However, this will not be universal for some time.

Signal Levels and Clocking

Because the T-1 signal must be transmitted over long distances, it has to meet two criteria: (1) it should have little or no DC-component, to minimize power consumption, and (2) it has to be self-clocking. To satisfy these requirements, a bipolar signal (see Figure 1) was designed. The presence, in a given bit-time-cell, of a pulse indicates a "1"; the absence of a pulse indicates a "0". These pulses alternate in polarity. The timing of the pulses allows the data clock to be regenerated from the data stream.

Since a constant stream of zeros would mean that no pulses would be transmitted,

Bits and Baud

The terms "baud" and "bits per second" are not synonymous. To avoid confusion, the term "bits per second" (b/s) is used universally in telecommunications. Also, in telecommunications, and in this text, "kilo" means one thousand (not 1,024), and "mega" means one million.

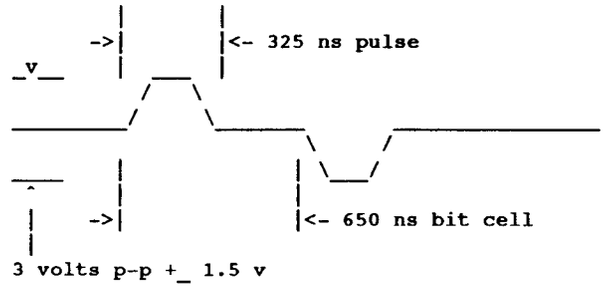


Figure 1. T-1 signal format

and the clock would get out of sync or "slip," there is a "ones-density" requirement. This is actually a two-fold requirement. First, there may be no more than 15 zeros in a row; second, there must be at least 3 pulses in any 24 time slots. This means that transmitting plain digital data through a T-1 is not directly possible.

Also, the pulses are supposed to always alternate in polarity. If they don't, then you have what is called a "bipolar violation" (BPV). While these are usually errors, they are sometimes deliberately caused to indicate special conditions.

Why is it so important that there be no DC component? Suppose RS-232 were used. RS-232 lines tend to stay in the idle or "space" state, which is represented as a -12 volt level. Zero bits are transmitted by transitions into the "mark" state, which is at a +12 volt level. Since there is always some finite resistance to ground, some current must flow to maintain the voltage level. Now consider a 10-mile-long wire, which has considerable resistance itself. Just keeping that wire in the idle state will require tremendous electrical power—to transmit no information at all. Today, fiber optic cables are used for the really long hauls, but fiber equipment is expensive, so short runs are still mostly copper. And "short," in telecommunications, is still many, many miles.

The clock is extracted by a careful and very involved procedure. The extracted clock is used only for the receive signal, of course. How can an accurate transmit signal be assured? Within a given facility, there is usually a standardized "composite clock" which is generated from one or more trusted T-1 circuits. The most accurate signals are those synchronized to a cesium frequency standard buried beneath a mountain in Colorado. Generally, you can synchronize to a T-1 that's synchronized to a T-1 that's synchronized to. . . Well, you get the picture.

Framing

I mentioned earlier that the first bit, the framing bit, was analogous to the start bit of an asynchronous data byte transmission. In normal async transmission, the start bit is always a zero. In normal synchronous transmission, there are no start bits, but sync bytes are transmitted periodically to establish synchronization. T-1 transmission is a little like both of these. The framing bit is not always 0 or 1, but goes through a sequence which allows the receiver to synchronize not only the frame itself (the one set of 192 bits) but also the frame within a sequence of twelve frames, which is called a "super-frame." Why is this necessary?

Well, a phone line carries more than just voice. It also carries indications of whether the handset is picked up ("off hook") or laying in the cradle ("on hook"), or whether the phone is ringing. This data is called "signaling." This data changes far less often than 8000 times a second, so what is done is that the least significant bit of the data is "robbed" in the sixth and twelfth frames of the superframe to carry two bits of signaling per superframe. Since the T-1 signals are going in both directions, the signaling does too. Figure 2 shows the framing bit sequence for what is called "standard superframe."

Frame	Framing bit	Signaling bit
1	1	-
2	0	-
3	0	-
4	0	-
5	1	-
6	1	A-bit
7	0	-
8	1	-
9	1	-
10	1	-
11	0	-
12	0	B-bit

Figure 2. Standard superframe framing bit and signaling

The specific meaning of the A and B bits depends on the particular interface at the end of the circuit. On long-distance telephone trunks, the B bit is always 0; the A bit is 0 for "idle" (on-hook) and 1 for "seized" (off-hook).

Actually, I didn't intend to go this far into telephony terminology, but as long as I've done so, a few more words may justify definition. Telephony is a hundred-year-old industry with a rich tradition. "Trunks" are long-distance telephone lines that run between switches, whereas "lines" are telephone lines that go from switches to "subscribers" (customers). Today, of course, trunks are usually T-1s carried within DS3s.

There is an enhanced version of standard framing available in some equipment these days which uses 24 frames to make up a superframe. This is called "extended superframe." It also offers a 6-bit CRC per superframe and a "hidden" 4kb/s data channel.

Signaling bits are not usually used for data transmissions. However, there is a switched 56 kb/s digital data service (DDS) available (more on this later). In order to guarantee the minimum ones-density as described earlier, usually, data services will force the least-significant data bit to a one and only use 7 data bits per DS0. This is what gives you the data rate of 56 kb/s.

Alarms

If a device receiving a T-1 bit stream should go out of sync, it will then collect frame after frame looking for a bit which follows the framing sequence, until synchronization is reestablished. This condition is known as "yellow alarm"—there is data, but it isn't locked in. Frames lost as it attempts to resynchronize are called "slips."

If the data signal is interrupted completely, and no data stream is being received at all, this condition is called a "red alarm." There is also a "blue alarm"—this is when all-ones are transmitted due to some equipment problem.

Remember, the T-1 signal is bidirectional. You always have data going in both directions. In the copper environment, two coaxial cables are used. So, when a piece of equipment is in an alarm condition, it may still be able to communicate with the other end and tell him something is wrong.

CEPT (European E-1)

T-1, as described in this article, is used in North and South America and most of Asia. However, the Europeans have done their usual thing and come up with *Something Better.* In this case, they have a 2.048 mb/s digital standard called CEPT (aka E-1).

CEPT's signal format is identical to T-1. However, the framing is different. There is no framing bit. Instead, there are 32 timeslots, and the first one (channel zero) is used to synchronize the frame, to carry all signaling (no "robbed bits") and to carry special flags for zero substitution. The other 31 channels are used for voice or data. Whenever any channel wants to send an all-zero byte, a different pattern is substituted and a flag is set in channel zero. This allows all-zero bytes to be transmitted easily in CEPT, which is called "clear channel capability."

Another difference in CEPT is that voice data is not companded with "mu-law" encoding, which is used in T-1, but with "A-law" encoding. (I didn't mention this earlier, but the 8-bit digitization of voice over digital services is not linear but logarithmic.)

Conversion between CEPT and T-1 is necessary for carriers

which move traffic (that is, voice and data) between Europe and the rest of the world. This conversion is performed by "gateway" devices which re-channelize the data, move the signaling bits around, and use lookup tables to convert between mu-law and A-law—for voice channels only, of course.

Clear-Channel Capability on T-1

The lack of clear 64 kb/s capability on T-1 has bothered some people. So, they've come up with some strange kludges to allow all-zero bytes to be transmitted. Usually, these schemes involve either stealing a bit from some other timeslot or frame, or causing deliberate BPVs, or other messy stuff. Some popular schemes are AMI, B8ZS and ZBTSI. None of these are universally accepted, and frankly, it isn't worth the trouble.

End-User Interfaces

There are two basic types of T-1 equipment which is used at an end-user site. These are channel banks and multiplexers. Both of these have basically the same function of creating an outgoing T-1 signal out of up to 24 data sources, and breaking the incoming T-1 signal into 24 data outputs.

The channel bank has, for each T-1, 24 channel cards. Each channel card is one of many different types. It might be an analog telephone interface, such as an FXS, FXO, SF, E&M, and so forth; it could be a synchronous data interface, such as a DSO/DP; or it could be an asynchronous data interface.

The DSO/DP card uses a synchronous data interface called DS0A. The DS0A signal is generally converted to V.35 or RS-449 using another box called a DSU. V.35 is more widely used in telecommunications than RS-449; it's synchronous and balanced, and uses a very bulky, expensive connector, but can drive very long cables.

The T-1 multiplexer usually offers more flexibility than a channel bank, but at higher cost. Some have modules which offer higher data rates than 56 kb/s, using multiple DS0s to carry the data.

A popular use of T-1 circuits in large companies is as a long-distance extension to an Ethernet network. Usually, an Ethernet bridge is used to convert Ethernet packets for the other location into a high-speed synchronous bit stream, which then is fed into a T-1 multiplexer. Another use of T-1 is for video teleconferencing.

Putting asynchronous data on a T-1 stream usually involves what's called "subrate" multiplexing. Almost always, a whole DS0 is used to carry as little as 1200 b/s of data, by sending each byte of data over and over. Some multiplexers are smarter and can fit as many as five 9600 b/s channels into a single DS0.

But What About Smaller End-Users?

Indirectly, all of us have benefited from the digital network interconnecting our cities. It is this noise- and loss-free network which has made possible our present-day high-speed modems. However, it seems a little strange to use an analog modem in a digital network. When you use a 2400 baud modem, your digital data at 2400 b/s is converted into analog sounds, which are then digitized at 64 kb/s, transmitted across the country, converted back to analog sounds, sent to the distant modem, which then converts the analog sounds back into a 2400 b/s digital data stream.

While the network is digital, though, your local telephone line is still analog. A real jump in speed requires getting away from analog circuits completely.

If you live near or in a big city, it may be economical to obtain 56 kb/s switched service (Digital Data Service, or DDS). This is a box which functions much like a really fast modem, but requires a digital connection such as V.35 or DS0A to the local telephone company office. For connections between two distant offices which are not needed around the clock, this service may be just the thing. This service is getting more affordable every day.

ISDN and Fiber to the Home

In the near future, ISDN will be available in most major cities. This is certain to happen, even though nobody is screaming for it (yet), but because the telecommunications industry has designed it as a universal interface for voice and data, and once everyone can have it, they'll realize they want it. After that, it will gradually make it out to the rurals.

ISDN actually consists of two types of interfaces. One is called the Primary Rate Interface, which consists of 23 "basic" channels carrying customer voice and data and a "data" channel carrying signaling, including dialing digits. These 24 channels (23B+D) are carried in a standard T-1. The PRI connection is for PBXs and other larger systems.

The main end-user ISDN interface is called the Basic Rate Interface (BRI). This consists of two B channels at 64 kb/s and a D channel at 16 kb/s. Typically, it is thought that this interface could carry, in a little bundle, a voice channel and a data channel. The data channel could carry limited-motion video in highly compressed form. This BRI is what may be coming soon to a telephone pole near you.

Meanwhile, in some parts of the country, housing developments have been "wired" completely with fiber optics. This means a digital 64 kb/s signal delivered right to your doorstep. If this catches on, it could mean wide acceptance of 56 kb/s switched type services. ISDN could follow shortly thereafter.

Yes, the signs are good that the days of pokey modems, and characters creeping oh-so-painfully slow across the screen, are coming to a merciful end.

You can be there to see it. Better yet, you can have a part in it. The telecommunications industry is always looking for people who can bridge the gap between the telephone and the computer. Maybe you'll find it to be your . . . calling!

References

There's only been time in this article to barely scratch the surface of the topic. However, neophytes in the telecommunications world run a serious risk of being buried under literal tons of technical specifications and jargon. I recommend the following, in descending order of readability: •

Understanding Telephone Electronics, by J. Fike and G. Friend, 1983, a Sams publication. Available at Radio Shack for the incredible bargain price of $3!

"An IXC's look at global ISDN", T. Kero, *Telephony*, April 23, 1990, p. 18. For that matter, most recent issues of *Telephony* magazine have good articles on ISDN.

Telephony's Dictionary, by G. Langley.

AT&T publication no. 41451, *1.544 mb/s Service Channel Interface Specifications*, Feb. 1974. Most AT&T and Bellcore publications tend to be readable, once you have sufficient "critical mass" to satisfy cross-references. Avoid CCITT books.

ZCPR3 and Modula Too

Modula 2 Makes the Z-System Connection

by David L. Clarke

Introduction

It's becoming quite popular lately. First we learned that one C compiler has been specially designed for the Z-System (Jay Sage's Z-System Corner, TCJ # 38, May/June 1989). Then in TCJ issue # 45, Joe Wright showed how a clever alias could patch Turbo Pascal so that it can access the Z-System environment which he defined as a set of Pascal records. Again, if you search around the Z-Nodes, you might find that someone else has found a way for Microsoft BASIC to access the Z3 environment. With all of this, it's only natural to expect Modula 2 to also make the Z-connection. Here's the surprise, Modula 2 may well have been the first.

There is a large archive of public domain files available for electronic download (i.e. FTP) from SIMTEL20 on the Internet. Some time ago I discovered a file there named Z33-TM2.LBR. This library documents how to interface Turbo Modula 2 and ZCPR3.3. The documentation was dated June 8, 1987. It seems that Steve Cohen used the fact that ZCPR3.3 passes the address of the Z3 environment in the HL register when loading and running a program. He used a small amount of assembler code to save this address for later usage by the high level code. Once the program was compiled and linked, it required a small amount of patching before it was ready to be run under ZCPR.

Although I have used Turbo Modula 2, I prefer using FTL Modula 2 from Workman & Associates. I therefore proceeded to adjust the program to my requirements. During the adjustment process, I expanded on several areas of the original approach. This article shows the final result.

The 'Implementation' Module

Modula 2 is linked so that all modules that require initialization will be executed before the main program code. The result of this linkage is that the first instruction executed (at location 100 hex) will always be a jump to the first module needing to be initialized. The trick is to make the jump go first to the code that saves the environment address and then jump to the original destination. Listing 1 shows how this is done. Instead of an implementation module written in Modula 2's high level language, this implemen-

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.

Listing 1

```
;IMPLEMENTATION MODULE Z34M2;

;(*  D. L. Clarke    for TCJ    (rev) 5 August 1990     *)

;(*  This module gets the address of the ZCPR3.3+ or    *)
;(*  Z3PLUS environment descriptor from the command     *)
;(*  processor and stores it away.  A procedure is also *)
;(*  supplied to retrieve the pointer later for use by  *)
;(*  programs written in FTL Modula 2.                  *)

;(*  This interface is based on similar code written    *)
;(*  by Steve Cohen for use with Turbo Modula 2.  In    *)
;(*  addition, improved verification of having found    *)
;(*  the Z33 or Z34 environment is based on code by     *)
;(*  Bridger Mitchell as found in The Computer Journal  *)
:(*  # 36, Jan/Feb 89.                                 *)

z3mat:  db      'Z33ENV'        ; used to find this code
                                ; with patching utility

;(*  These next instructions are the area that the      *)
;(*  program will immediately jump to (once we patch it *)
;(*  to do so).  We save the environment pointer which  *)
;(*  is provided in HL, and then jump to the procedure  *)
;(*  that the program initially went to first, before   *)
;(*  we patched it.                                     *)

patch:  ld      (z34adr),HL     ;save environment pointer
        jp      0               ;replace '0' by original
                                ;program destination

z34adr: dw      0               ;pointer to Z34ENV

        label                   ;entry to GetEnv procedure
GetEnv:                         ;PROCEDURE GetEnv(): Z34PTR;
        pop     IY              ;BEGIN (* save ret addr *)
        ld      HL,(z34adr)     ;
        push    HL              ;
        inc     HL              ;
        inc     HL              ;
        inc     HL              ;
        ld      A,'Z'           ;
        ld      (z3mat+1),A     ;  IF z34adr^.z3str = "Z3ENV"
        ld      B,5             ;
        ld      DE,z3mat+1      ;
        call    match           ;
        pop     DE              ;              AND
        ld      A,'3'           ;
        ld      (z3mat+1),A     ;
        jr      NZ,notz3        ;
        ld      HL,1Bh          ;     z34adr^.z3env = z34adr
        add     HL,DE           ;
        ld      A,(HL)          ;
        cp      E               ;  THEN
        jr      NZ,notz3        ;
        inc     HL              ;
        ld      A,(HL)          ;    RETURN z34adr
        cp      D               ;
        jr      Z,return        ;  ELSE
notz3:  ld      HL,0            ;
        jr      end             ;    RETURN NIL
return: ld      HL,(z34adr)     ;
end:    ex      (SP),HL         ;  END
        push    IY              ;       (* restore ret addr *)
        ret                     ;END  GetEnv;

match:  ld      A,(DE)          ;PROCEDURE match(B);
        cp      (HL)            ;BEGIN
        inc     HL              ;  FOR i := 0 TO B-1 DO
        inc     DE              ;    IF HL^[i] <> DE^[i] THEN
                                ;         RETURN FALSE END
        ret     NZ              ;  END;
        djnz    match           ;  RETURN TRUE
        ret                     ;END  match;

        END     ;Z34M2
```

tation is written in assembler language. The instruction at location 'patch' moves the address supplied by HL into location 'z34adr'. The next instruction is shown as 'jp 0', however, the zero will eventually be replaced by the 'original destination' mentioned above.

FTL Modula 2 comes with its own assembler. This assembler uses the pseudo-op 'label' to relate procedure names between the definition module and the assembler code. In this case, when the definition module is eventually written, it will define a procedure named 'GetEnv'. In this implementation module written in assembler language, the code for the procedure follows the 'label' statement in the listing.

There is the possibility that what was passed in the HL register was not the environment address (for instance, when the program is running under bare CP/M). To avoid the possibility of returning an invalid address value, I make two checks on the value. First, offset 3 into the (assumed) environment must contain the string "Z3ENV" and second, hex offset 1B into the environment must be the address of the environment itself. If either test fails, a zero value is returned by GetEnv. The code used for these tests is based on Bridger Mitchell's article in TCJ issue # 36.

The Definition Module

Steve Cohen's original definition module merely defined the procedure that returns the environment address. I realized that the ZCPR environment could easily be expressed using Modula 2 record structures. (As mentioned above, this same approach was recently used to define the environment for Turbo Pascal.) My Modula 2 definition module is given in Listing 2. The records are set up so that either the ZCPR3.3 version (crt1, prt1, etc.) or the newer ZCPR3.4 version (DrvVec, spar1-2, ccp, etc.) may be used. All of the separate structures in the environment are defined as individual records. The ZCPR environment itself is listed as 'Z3ENV'. (Modula 2 is case sensitive, thus 'Z3WHL' and 'z3whl' are not the same type; one is a byte and the other is a pointer to the byte.)

The 'GetEnv' procedure is also defined. It is actually a function that returns a pointer to the environment (Z3ENV).

Listing the Environment

Once we have the address on the Z-environment, the first thing we might do is display its present state. Next time I shall diagram how the environmental components are laid out in memory, but for now I shall just list some of their contents. Listing 3 is the program module for the code that lists the environment. I chose to list the "Z3ENV" string itself, the state of the wheel byte, indicate

Listing 2

```
DEFINITION MODULE z34m2;

(*      D. L. Clarke        for TCJ         5 August 1990       *)

FROM  SYSTEM  IMPORT  ADDRESS, BYTE;

TYPE
    TCAP =   RECORD                              (* TCAP definition *)
               TCName:  ARRAY [0..14] OF CHAR;
               TrmMod:  CHAR;
               UpArrow: CHAR;
               DnArrow: CHAR;
               RtArrow: CHAR;
               LfArrow: CHAR;
               ClsDly:  CHAR;
               CurDly:  CHAR;
               EOSDly:  CHAR;
               CtlStr:  ARRAY [0..104] OF CHAR
             END;

    DU =     ARRAY [1..2] OF BYTE;                (* disc, user *)

    path =   RECORD                  (* external path definition *)
               du:      ARRAY [1..5] OF DU;
               EOP:     BYTE    (* terminating null *)
             END;

    NDRrec = RECORD                  (* named directory definition *)
               du:      DU;
               Name:    ARRAY [0..7] OF CHAR;
               Pass:    ARRAY [0..7] OF CHAR
             END;
    NDR =    RECORD
               rec:     ARRAY [1..14] OF NDRrec;
               EOR:     BYTE    (* terminating null *)
             END;

    CL =     ARRAY [0..249] OF CHAR;        (* command line def *)
    MCL =    RECORD
               nxt:     POINTER TO CL;
               max, len: BYTE;
               str:     CL
             END;

    (*   various other definitions of environmental records    *)
    ShStk = ARRAY [1..4] OF ARRAY [0..31] OF CHAR;
    MSG =   ARRAY [0..79] OF CHAR;
    Z3WHL = BYTE;
    Z3ver = (z33, z34);
    zcrt =  RECORD
               cols:    BYTE;               rows:    BYTE;
               lins:    BYTE
             END;
    zptr =   RECORD
               pcol:    BYTE;               prow:    BYTE;
               plin:    BYTE;               form:    BYTE
             END;
    filename = ARRAY [0..10] OF CHAR;

    (*       definition of the Z3 environment itself       *)
    Z3PTR = POINTER TO Z3ENV;
    Z3ENV = RECORD
               wrmboot:  ARRAY [0..2] OF BYTE;
               z3str:    ARRAY [0..4] OF CHAR;
               EnvType:  BYTE;
               expath:   POINTER TO path;    expaths:  BYTE;
               rcp:      ADDRESS;            rcps:     BYTE;
               iop:      ADDRESS;            iops:     BYTE;
               fcp:      ADDRESS;            fcps:     BYTE;
               z3ndir:   POINTER TO NDR;     z3ndirs:  BYTE;
               z3cl:     POINTER TO MCL;     z3cls:    BYTE;
               z3env:    Z3PTR;              z3envs:   BYTE;
               shstk:    POINTER TO ShStk;   shstks:   BYTE;
               shsize:   BYTE;
               z3msg:    POINTER TO MSG;
               extfcb:   ADDRESS;            extstk:   ADDRESS;
               quietflg: BYTE;
               z3whl:    POINTER TO Z3WHL;
               speed:    BYTE;
               maxdrv:   BYTE;               maxusr:   BYTE;
               DUOK:     BYTE;
               crt:      BYTE;               prt:      BYTE;
               crt0:     zcrt;
    CASE         :      Z3ver OF
      z33:     crt1:     zcrt     |
      z34:     DrvVec:   BITSET;
               spar1:    BYTE     |
    END;
               ptr0:     zptr;
    CASE         :      Z3ver OF
      z33:     ptr1:     zptr;               ptr2:     zptr;
               ptr3:     zptr;    |
      z34:     spar2:    ARRAY [2..5] OF BYTE;
               ccp:      ADDRESS;            ccps:     BYTE;
               dos:      ADDRESS;            doss:     BYTE;
               bios:     ADDRESS  |
    END;
               ShVar:    filename;
               file1:    filename;
               file2:    filename;
               file3:    filename;
               file4:    filename
             END ;

(*     procedure to return the environment's address to a    *)
(*     high level program.  Note, '0' = no Z3 environment    *)

PROCEDURE GetEnv(): Z3PTR;

END z34m2.
```

Listing 3

```
MODULE tz34m2;

(*      D. L. Clarke        for TCJ         (revised) 8 August 1990
*)

(*   Test ZCPR34-Modula 2 interface mechanism by listing some
contents  *)

FROM InOut  IMPORT Write, WriteString, WriteCard, WriteHex, WriteLn;
FROM z34m2  IMPORT Z3PTR, Z3ENV, GetEnv;
FROM SYSTEM IMPORT ADDRESS, ADR, BYTE;

VAR
        i:   CARDINAL;
        env: Z3PTR;
BEGIN
  env := GetEnv();
  IF CARDINAL(env) = 0 THEN
    WriteString("No Z-System environment");      WriteLn
  ELSE
    WriteString(env^.z3str);                     WriteLn;
    WriteString("Wheel byte is ");
    IF ORD(env^.z3whl^) = 0 THEN
      WriteString("OFF")
    ELSE
      WriteString("ON")
    END;                                         WriteLn;
    WriteString("DU usage is ");
    IF ORD(env^.DUOK) = 0 THEN WriteString("NOT ")  END;
    WriteString("okay");                         WriteLn;
    WriteString("Max drive is ");
    Write(CHR(ORD('@') + ORD(env^.maxdrv)));     WriteLn;
    WriteString("Max user area is ");
    WriteCard(ORD(env^.maxusr), 1);              WriteLn;
    WriteString("The command line is:");         WriteLn;
    WriteString("    ");
    WriteString(env^.z3cl^.str);                 WriteLn;
    WriteString("The named directories ");
    IF ORD(env^.z3whl^) # 0 THEN
      WriteString("(and their passwords) ")
    END;
    WriteString("are:");                         WriteLn;
    i := 1;
    LOOP
      IF ORD(env^.z3ndir^.rec[i].du[1]) = 0 THEN EXIT END;
      Write(CHR(ORD('@') + ORD(env^.z3ndir^.rec[i].du[1])));
      Write(CHR(ORD('0') + ORD(env^.z3ndir^.rec[i].du[2])));
      WriteString("  ");   WriteString(env^.z3ndir^.rec[i].Name);
      IF ORD(env^.z3whl^) # 0 THEN
        WriteString("  "); WriteString(env^.z3ndir^.rec[i].Pass);
      END;                                       WriteLn;
      INC(i);  IF i > ORD(env^.z3ndirs) THEN EXIT END
    END
  END
END tz34m2.
```

if the DU form is allowed for directory names and show the max user and drive values, the command line, and the named directories. Note that the wheel byte is consulted before displaying the directory passwords. Running the tz34m2 program will produce a listing something like the following:

```
Z3ENV
Wheel byte is ON
DU usage is okay
Max drive is P
Max user area is 15
The command line is:
TZ34M2
The named directories (and their passwords) are:
A0  COMMAND
B0  WORK
M0  ROOT
```

Patching the Program

Actually if we ran the program immediately after linking, the output of the program would be the message "No Z-System environment". After the program has been linked, it must be patched before it is ready. This may be done with your favorite patching program (I used ZPATCH). The process is as follows:

1. Make a note of the address in the two bytes at hex locations 101 and 102. Write them down.

2. Search for the string "Z33ENV" which was placed in Z34M2.ASM for this purpose.

3. Replace the bytes 10 and 11 locations after the beginning of the string with the two bytes recorded in the first step above. Be sure to keep them in the original order.

4. Go back and replace the values in hex locations 101 and 102 with the address of the string plus six. Remember that the low byte goes first.

For example, when I patched TZ34M2.COM, I found hex values of 41 and 34 at locations 101 and 012. The string was found at hex location 0497. Ten locations higher is 04A1, therefore I placed 41 in 04A1 and 34 in 04A2. Six locations above the string is location 049D. Therefore 9D goes in location 101 and 04 goes in location 102.

An Easier Way to Make the Patch

Although the patching exercise is not too difficult, after doing it a few times, it becomes quite tedious. The solution is to write a Modula 2 program that does the patching automatically. Listing 4 is my program that does this. M2instal may either be passed a file name from the command line or it will prompt for one. The file is read (and written) in blocks of binary data. I used a variation of the Boyer-Moore algorithm for the string search. This algorithm was described by Bridger Mitchell in TCJ issue # 45. Instead of comparing each file byte, one at a time, to the characters in the string to be matched, this procedure is capable of skipping over several bytes at a time—sometimes approaching twice the length of the string being matched. The number of bytes to be skipped depends on two functions, 'delta1' and 'delta2'. The first determines the places to skip if the byte being compared matches one of the pattern characters. The second determines the places to skip depending on the current location in the pattern that is being matched. The number of places skipped is the larger of the two delta values. This really speeds up the search. Once the string is found in the file, it is an easy job to make the substitutions. It is also helpful that the Modula 2 'ReadBlock' procedure is able to access the file blocks in a non-sequential manner. Once we have made the patch where the string is found, we have to go back to the first block of the file and patch the initial jump location.

Conclusion

In this article I have shown how Modula 2 can gain access to the Z - System environment. A simple program was used to print out some information from that environment. The process of patching the resulting program file has been simplified by yet another Modula 2 program. Now that the environment is accessible, there are many things that can be done. Next time I will show how to diagram the way in which the components of the environment are arranged in memory. We will also be making use of the TCAP definition. ●

Listing 4 is on the following page.

Listing 4

```
MODULE M2instal;
(*    D. L. Clarke        for TCJ         (revised) 6 August 1990  *)
(*    Patches Modula 2 programs that include the z34m2 module to   *)
(*    allow access to the Z-System environment.                    *)
(*    The filename of the program to be patched may be included    *)
(*    on the command line as follows:                              *)
(*            A> m2instal tz34m2.com                               *)
(*    or the program will prompt for the filename.  Only one       *)
(*    program may be installed at a time.  Also, the '.COM'        *)
(*    extension must be included in the filename.                  *)

(*    No, Hortense, you do not want to 'm2instal' the M2INSTAL.COM *)
(*    program itself.                                              *)

    FROM Files   IMPORT FileName, FILE, Lookup, ReadBlock, WriteBlock,
Close;
    FROM InOut   IMPORT ReadString, WriteString, WriteInt, WriteLn;
    FROM Command IMPORT Parameter, GetParams;
    FROM SYSTEM  IMPORT ADR, ADDRESS, TSIZE;

    CONST
        plength = 6;
        blk_length = 128;
        blk_count = 4;
        buf_size = blk_count * blk_length;

    TYPE
        block = ARRAY [0 .. blk_length-1] OF CHAR;
        buff  = ARRAY [0 .. blk_count-1] OF block;
        bufr  = ARRAY [0 .. TSIZE(buff)-1] OF CHAR;
        addr  = ARRAY [0 .. TSIZE(ADDRESS)-1] OF CHAR;

    VAR
        name:        FileName;
        file:        FILE;
        buffer:      bufr;
        addr_value:  addr;
        pat:         ARRAY [0 .. plength] OF CHAR;
        Param:       ARRAY [0 .. 10] OF Parameter;
        reply:       INTEGER;
        initial_blk: INTEGER;
        blk_offset:  INTEGER;
        fnd_locat:   INTEGER;
        old_strt:    INTEGER;
        new_strt:    INTEGER;
        buf_length:  INTEGER;

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

    PROCEDURE delta1 (ch: CHAR): INTEGER;
    BEGIN
        CASE ch OF
            'Z':  RETURN plength - 1
        |   '3':  RETURN plength - 3
        |   'E':  RETURN plength - 4
        |   'N':  RETURN plength - 5
        |   'V':  RETURN plength - 6
            ELSE  RETURN plength - 0
        END
    END delta1;

    PROCEDURE delta2 (n: INTEGER): INTEGER;
    BEGIN
        IF n = plength THEN RETURN 1 ELSE RETURN 2 * plength - n END
    END delta2;

    (* Boyer-Moore string searching algorithm                      *)
    (*    returns buflen + 1 if string is not found, otherwise     *)
    (*    returns offset (into 'bufr') to beginning of string      *)
    PROCEDURE bmsearch(i, j: INTEGER; buflen: INTEGER): INTEGER;
    BEGIN
        LOOP
            IF (i > buflen) OR (j = 0) THEN RETURN i END;
            IF buffer[i-1] = pat[j-1] THEN
                DEC(i);     DEC(j)
            ELSE
                i := i + max(delta1(buffer[i]), delta2(j));
                j := plength
            END
        END
    END bmsearch;

    (* search for string a block at a time                         *)
    (*    returns -1 if string is not found, otherwise             *)
    (*    returns offset (into file) to beginning of string        *)
    PROCEDURE blksearch(): INTEGER;
        VAR
            i, j:       INTEGER;
    BEGIN
        initial_blk := - blk_count;
        buf_length := 0;
        i := plength;   j := plength;
        LOOP
            IF i < buf_length THEN
                (* complete match -- return file offset *)
                blk_offset := i;
                RETURN initial_blk * blk_length + i
            ELSIF i >= buf_length + plength THEN
                (* no match -- read 'blk_count' more 'block's *)
                i := i - buf_length;
                initial_blk := initial_blk + blk_count;
                ReadBlock(file, ADR(buffer), initial_blk, buf_size,
                          buf_length)
                IF initial_blk = 0 THEN
                    addr_value[0] := buffer[1];    addr_value[1] :=
                          buffer[2];
                    old_strt := INTEGER(addr_value)
                END
            ELSE
                (* partial match at end of block  --  move last
                          'block' to  *)
                (* beginning of 'bufr' and read 'blk_count'-1 more
                          'block's *)
                i := i - buf_length + blk_length;
                initial_blk := initial_blk + blk_count - 1;
                ReadBlock(file, ADR(buffer), initial_blk, buf_size,
                          buf_length)
            END;
            IF buf_length > 0 THEN
                i := bmsearch(i, j, buf_length)
            ELSE
                (* End-Of-File -- string not found *)
                RETURN -1
            END
        END
    END blksearch;


BEGIN
    (* get name of file to patch and open the file *)
    GetParams(Param, reply);
    IF reply < 1 THEN
        WriteString("Enter name of file > ");  ReadString(name);
        buf_length := 0;
        LOOP
            IF buf_length > HIGH(name) THEN EXIT END;
            IF name[buf_length] <= ' ' THEN
                name[buf_length] := 0c;
                EXIT
            END;
            INC(buf_length)
        END
    ELSE
        buf_length := 0;
        WHILE (buf_length <= HIGH(name)) AND
              (buf_length <= HIGH(Param[0]^.Chars)) AND
              (Param[0]^.Chars[buf_length] # 0c) DO
            name[buf_length] := Param[0]^.Chars[buf_length];
            INC(buf_length)
        END;
        IF buf_length <= HIGH(name) THEN name[buf_length] := 0c END;
        IF reply > 1 THEN
            WriteString("All files except ");    WriteString(name);
            WriteString(" will be ignored!");    WriteLn
        END
    END;
    Lookup(file, name, reply);

    (* search for the string "Z33ENV" which identifies patch area *)
    IF reply >= 0 THEN
        pat := "Z33ENV";
        fnd_locat := blksearch() + 400B;  (* 400B = 100 hex *)
        IF fnd_locat > 400B THEN
            new_strt := fnd_locat + 6;
            IF blk_offset + 10 >= blk_length THEN
                INC(initial_blk);
                blk_offset := blk_offset - blk_length;
                ReadBlock(file, ADR(buffer), initial_blk, buf_size,
                          buf_length)
            END;

            (* patch and close the file *)
            addr_value := addr(old_strt);
            buffer[blk_offset+10] := addr_value[0];
            buffer[blk_offset+11] := addr_value[1];
            WriteBlock(file, ADR(buffer), initial_blk, buf_size,
                       buf_length);
            ReadBlock(file, ADR(buffer), 0, buf_size, buf_length);
            addr_value := addr(new_strt);
            buffer[1] := addr_value[0];      buffer[2] :=
                       addr_value[1];
            WriteBlock(file, ADR(buffer), 0, buf_size, buf_length);
        ELSE
            WriteString("Z34M2 not imported");    WriteLn
        END;
        Close(file)
    ELSE
        WriteString("Not able to open file ");    WriteString(name);
        WriteString(", error code ");             WriteInt(reply, 1);
        WriteLn
    END
END M2instal.
```

Tips on Using LCDs

Interfacing to the 68HC705

by Karl Lunt

Liquid crystal displays (LCDs) offer a cheap, simple way to add alphanumeric output to your microcontroller projects. The following discussion shows how easily such a display can be added to the Motorola 68HC705 MCU.

The LCD

The surplus markets carry a large variety of LCDs which have appeared at very reasonable prices lately. I purchased a couple of 40 x 2 displays from Timeline for about $25 a few months ago. Shop around and check the ads; you will probably find similar deals. Available formats include 16 characters by one line, 16 x 2, 20 x 1—up to a full 80 x 24. You can use this interface technique with nearly any LCD of up to 80 characters.

The ICs used on the LCD board itself determine if it will work in this design. You need to find an LCD that uses a Hitachi HD44780 display control chip. This will be a surface-mount device on the backside of the LCD board. You will probably also find one or more Hitachi HD44100 support chips as well. This display control chip turns up in LCDs from many different manufacturers; firms such as Hitachi and Optrex offer suitable displays.

LCDs using the Hitachi HD44780 chip-set connect to a computer via a simple parallel data bus. The controlling machine must provide the following signals:

An enable line (E) that, when low, enables the LCD. This line, like all other LCD signal lines, uses TTL-level signals.

A register select line (RS) that selects either data or command mode of operation. If the line is low when the LCD is enabled, the data bus will carry an LCD command; if the line is high, the data bus holds a character for display.

A read/write line (RW*) that can simply be tied low. This indicates that the controlling computer will only write to the display. Incidentally, the Hitachi literature describes how to read the contents of each LCD character position. You could use this feature to capture the display's contents, send a high-priority message, then restore the display.

An eight-bit bidirectional data bus that will normally send data and commands to the LCD. You can shrink this bus to only four bits, saving four I/O lines and permitting the controlling computer to service an LCD with only a single eight-bit output port. That is the basis of the technique used here.

Vcc (+5 volts), GND (ground), and Vo (display brightness) provide power and control the display brightness. Generally, Vo can be connected to ground with acceptable results.

Since the HD44780 chip-set consists of CMOS parts, the display draws very little current and interfaces directly to CMOS MCUs such as the Motorola 68HC705.

Communication Basics and the Power-up Ritual

Software in the host computer must perform two major functions in interfacing with the LCD; power-on/reset initialization and display updating. Both functions consist of writing the proper commands and data to the LCD.

When writing commands or data to the LCD, you must pay close attention to the sequencing of the RS*, E, and data bus lines (I am assuming that R/W is wired low). The following lines must change state in this order (initially, RS*, E, and R/W are low):

First, set the state of RS* as needed; low indicates that you will write a command to the LCD, while high means you will write a character to the display.

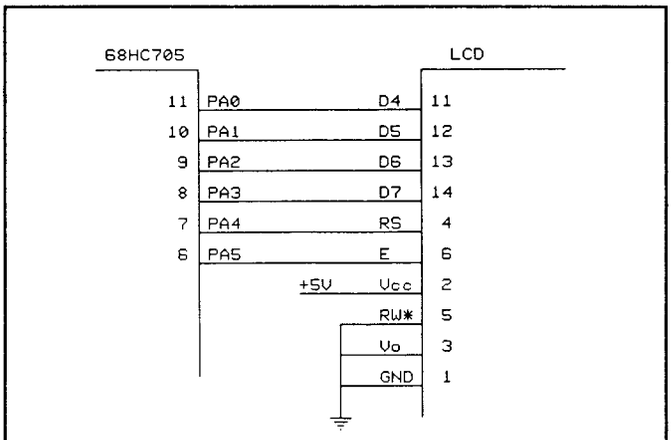
Second, place the upper four bits of the 8-bit command (or character) on the upper four data lines of the LCD. (Remember, we are using a four-bit interface here.)

Third, bring line E high, wait about 3 usec, then bring line E low again. I say "about" because the Hitachi documentation doesn't make the minimum time requirement very clear; I have used a 3 usec delay with good results.

Fourth, place the lower four bits of the 8-bit command (or character) on the upper four data lines of the LCD.

Fifth, bring line E high, wait about 3 usec, then bring line E low again. This completes the transmission of an 8-bit byte to the LCD.

The initialization ritual consists of a set sequence of commands that place the LCD in a known state, ready to process further



Typical connections betwee a 68HC705 and a Hitachi-style LCD

```
*
*  LCD library
*
*  These routines control a Hitachi-type LCD, connected with a 4-bit
*  interface to a single 8-bit port on the 68HC705.  The main routine
*  must contain equates defining this connection:
*
*  LCDPORT must contain the address of the '705 port.
*  LCD.E must contain the bit connected to the display's E line.
*  LCD.RS must contain the bit connected to the display's RS line.
*
*  These routines assume that the low nybble of the LCDPORT is connected
*  to the highest four data lines of the LCD.
*
*  The main routine must also allocate RAM storage for these modules:
*
*  LCDTMP    RMB     1
*  LCDTMP1   RMB     1
*  LCDOUTA   RMB     1
*
*  The main routine must also use the RAM variable WAIT$ to indicate
*  when it is safe to write to the display.  These routines will not
*  write to the display unless WAIT$ holds a 0; they will loop endlessly
*  until WAIT$ is cleared.
*
*  The logic in LCDCMD and LCDCHAR, that writes directly to the display,
*  automatically selects the appropiate delay value for use with WAIT$.
*  These values presume that WAIT$ decrements once each 200 usec.  If
*  you use a different time constant for decrementing WAIT$, adjust
*  these routines accordingly.
*



*
*  LCDINIT -- Initialize the LCD
*
*  This routine configures the LCD for 4-bit interface, cleared display,
*  two lines, 5x10 character font.  Upon exit, the LCD RS line is low
*  (command register selected).
*
*  The initialization sequence given here was taken from the Hitachi
*  manual.  That manual recommends specific time delays between the
*  first three commands that differ from the delays normally associated
*  with those commands.  Since LCDOUT is used to send these commands,
*  this routine explicitly sets WAIT$ to the recommended values following
*  each call to LCDOUT.  The values used assume a 200 usec delay for
*  each decrement of WAIT$.
*
LCDINIT:
        LDA     #0              DATA = XXXX, RS = 0, E = 0
        STA     LCDPORT         SEND TO LCD

        LDA     #75             GET A 15 MS DELAY
        STA     WAIT$

        LDA     #$03            8-BIT INTERFACE
        JSR     LCDOUT          SEND TO DISPLAY
        LDA     #22             GET A 4.2 MS DELAY
        STA     WAIT$

        LDA     #$03            SEND 8-BIT COMMAND AGAIN
        JSR     LCDOUT
        LDA     #1              GET A 120 US DELAY
        STA     WAIT$

        LDA     #$03            SEND 8-BIT COMMAND ONE LAST TIME
        JSR     LCDOUT
        LDA     #25             GET A 5 MS DELAY
        STA     WAIT$

        LDA     #$02            SET UP 4-BIT INTERFACE
        JSR     LCDOUT
        LDA     #25             GET A 5 MS DELAY
        STA     WAIT$


        LDA     #%00101100      2 LINES, 5X10 FONT, 4-BIT INTERFACE
        JSR     LCDCMD
```

commands. Since the LCD powers up in an unpredictable state, you must follow Hitachi's recommended startup sequence carefully if you want the display to behave properly.

NOTE: The four steps of the power-up ritual involve sending ONLY the upper four bits of an 8-bit command byte. For the power-up ritual, DO NOT send the low four bits of a command byte to the LCD.

Step 1: Wait at least 15 msec after power-up, then write the command byte 0011xxxx to the LCD.

Step 2: Wait at least 4.2 msec, then write the same command byte (0011xxxx) to the LCD.

Step 3: Wait at least 120 usec, then write the same command byte (0011xxxx) to the LCD.

Step 4: Wait at least 5 msec, then write the command byte 0010xxxx to the LCD.

You have now initialized the LCD and configured it for a four-bit interface. All subsequent exchanges between the host computer and the LCD must consist of two four-bit transfers, as described previously.

Setting the Display Parameters

Once you have completed the start-up ritual, you can set the LCD into a number of different configurations. I usually send the following sequence of commands:

00101100 to set the LCD for two lines (assuming, of course, your display can handle two lines) and 5x10 pixel characters.

00001000 to shut the display and the cursor off and to disable cursor blink.

00000001 to clear the display and bring the cursor to address 0 (upper left corner).

00000110 to cause the LCD to automatically increment the cursor position after each new character and to prevent the LCD from rotating the display underneath the cursor as each new character arrives.

00001111 to activate the display, the cursor, and cursor blinking.

Now you can begin sending characters to the LCD and see them appear on the display.

The above set of configuration commands are just a typical sequence and do not do justice to the versatility of the Hitachi chip-set. For a more complete description of the command set, along with an excellent discussion on the inner workings of the LCD, check out Ed Nisely's article, *The True Secrets of Working with LCDs,* in the April/May 1989 issue of Circuit Cellar INK.

You can also try reading the Hitachi technical literature, but I find their disjointed English text too distracting to be helpful.

```
        LDA     #%00001000      DISPLAY OFF, NO CURSOR, NO BLINK
        JSR     LCDCMD

        LDA     #%00000001      CLEAR DISPLAY, CURSOR TO ADDR 0
        JSR     LCDCMD

        LDA     #%00000110      INCREMENT CURSOR, NO DISPLAY SHIFT
        JSR     LCDCMD

        LDA     #%00001111      DISPLAY ACTIVE, CURSOR ACTIVE & BLINKS
        JSR     LCDCMD

        RTS



*
*  LCDOUT -- Send a nybble to the LCD
*
*  This routine sends the data in AR to the display.  The LCD's RS
*  line must already be at the proper value.  Upon exit, the AR is
*  clobbered and the LCD's E line is low.
*

LCDOUT:
        TST     WAIT$           WAIT UNTIL OK TO PROCEED
        BNE     LCDOUT

        BCLR    LCD.E,LCDPORT       DROP THE E LINE
        AND     #$0F            LEAVE ONLY LOW NYBBLE
        STA     LCDOUTA         SAVE IT
        LDA     LCDPORT         GET CURRENT PORT VALUE
        AND     #$F0            LEAVE TOP NYBBLE AS IS
        ORA     LCDOUTA         OR IN THE LOW NYBBLE
        STA     LCDPORT         WRITE DATA TO PORT
        BSET    LCD.E,LCDPORT       RAISE THE E LINE
        NOP                     WAIT JUST A BIT
        NOP
        BCLR    LCD.E,LCDPORT       DROP THE E LINE AGAIN
        RTS


*
*  LCDCHAR -- send a character to the display
*
*  Send the character, in AR upon entry, to the display.  This
*  routine sets the LCD's RS line high and leaves it that way
*  upon exit.  The AR is preserved.
*

LCDCHAR:
        TST     WAIT$           WAIT UNTIL OK TO PROCEED
        BNE     LCDCHAR

        BSET    LCD.RS,LCDPORT      SET RS = 1
        STA     LCDTMP          SAVE THE CHAR
        LDA     #1              GET A 120 US DELAY
LCDCHAR1:
        STA     LCDTMP1         SAVE THE WAIT VALUE
        LDA     LCDTMP          RELOAD THE CHARACTER
        RORA                    SHIFT TO HIGH NYBBLE
        RORA
        RORA
        RORA
        JSR     LCDOUT          SEND TO DISPLAY
        LDA     LCDTMP          RELOAD THE CHARACTER
        JSR     LCDOUT          SEND LOW NYBBLE TO DISPLAY
        LDA     LCDTMP1         NOW GET THE DELAY COUNT
        STA     WAIT$           AND SET UP WAIT$
        LDA     LCDTMP          RESTORE THE AR
        RTS


*
*  LCDCMD --  send a display command to the LCD
*
*  This routine sends a display command, in AR upon entry, to the
*  LCD.  Upon exit, the RS line is low and the AR is preserved.
*
```

(Listing continued on following page)

On to the Software

The LCD library routines shown here assume that all lines for controlling the LCD are connected to a single '705 parallel port, called LCDPORT. For example, you might dedicate PORTC to driving your LCD; in this case, you would simply equate LCDPORT to PORTC in your assembly language source.

Of the lines associated with LCDPORT, the lower four (bits 0 through 3) must be tied to the upper four data bits on the LCD (bits 4 through 7). While this might seem wrong based on the instructions given above, a quick look at the code (routine LCDOUT) shows that the software adjusts for the "twist" in the wiring.

One of the remaining four bits of LCDPORT must be selected as the E line. Whichever line you choose, wire it to the E pin on the LCD and equate the label LCD.E to the appropriate bit number. For example, if you select bit 7 to be your E line, you would equate LCD.E to 7.

You must also choose an RS line. Wire the selected line to the LCD's RS* pin and equate the label LCD.RS to that bit number.

The main body of your code must also provide some space for variables used by the LCD library. You do this by reserving one memory byte (RMB) each for the labels LCDTMP, LCDTMP1, and LCDOUTA.

Finally, you must provide some means of timing the delays needed between accesses to the LCD. This can best be done by setting up an OCMP (output compare) server that generates, then services, an interrupt every 200 usec.

The supplied code relies on the OCMP server properly processing a global variable called WAIT$. Each time the OCMP server wakes up, it must check the value in WAIT$. If that value is zero, the server simply reloads the OCMP timer and returns from the interrupt.

If, however, WAIT$ is not zero, the OCMP server must decrement WAIT$ before rearming the OCMP timer. This allows a routine to simply load a delay value into WAIT$, then wait around until WAIT$ goes to zero, indicating that the required amount of time has elapsed.

Of course, if you don't want to bother writing the OCMP server, you can simply build a small time-wasting loop that generates a delay, based on an argument passed to it upon entry. Call it WAIT (for example) and do a JSR to it, with the delay value properly set, each time you need to wait a little while.

Routine LCDSTRING shows how easy it is to add a string output function to the LCD library. This module uses a subroutine built up in RAM (sometime during program initialization) to provide a miss-

```
*  This routine jumps into the LCDCHAR routine to use common code.
*  Note that when the jump is made, the AR holds the proper delay value
*  to be stored in WAIT$ and the original value in the AR has been
*  saved in LCDTMP.
*

LCDCMD:
        TST     WAIT$           WAIT UNTIL OK TO PROCEED
        BNE     LCDCMD

        BCLR    LCD.RS,LCDPORT      SET RS = 0
        STA     LCDTMP          SAVE THE COMMAND FOR NOW
        CMP     #$03            SEE WHAT KIND OF DELAY WE NEED
        BHI     LCDCMD1         BRANCH IF NEED SHORT DELAY
        LDA     #25             GET A 5 MS DELAY
        BRA     LCDCMD2
LCDCMD1:
        LDA     #1              GET A 120 US DELAY
LCDCMD2:
        BRA     LCDCHAR1        GO USE COMMON CODE


*
*  LCDSTRING -- write a string to the display
*
*  This routine writes a null-terminated string to the display.  Upon
*  entry, the main routine must have set up the LDAIND subroutine in
*  low RAM:
*
*  LDAIND:
*       FCB     $D6             LDA (IX2)
*       RMB     1               MSB OF STRING ADDR
*       RMB     1               LSB OF STRING ADDR
*       FCB     $81             RTS
*
*  The calling routine must have positioned the cursor as desired.
*
*  Upon exit, the AR and XR are destroyed.  Note that no check is made
*  as to string length, CR, LF or tabs.
*

LCDSTRING:
        CLRX                    START AT BEGINNING
LCDSTR1:
        JSR     LDAIND          GET A CHAR
        BEQ     LCDSTRX         BRANCH IF DONE
        JSR     LCDCHAR         SEND TO DISPLAY
        INCX                    POINT TO NEXT CHAR
        BRA     LCDSTR1         LOOP UNTIL DONE
LCDSTRX:
        RTS
```

ing but vital addressing mode. The RAM-based routine LDAIND consists of the byte $D6 at location LDAIND and the byte $81 at location LDAIND+3.

To output a string to the LCD, a calling routine must first store the address of the null-terminated string into locations LDAIND+1 (msb) and LDAIND+2 (lsb). Next, the calling routine executes a JSR to LCDSTRING to output the string.

LCDSTRING first clears the XR, then repeatedly calls LDAIND, bumping the XR each time. Control finally exits LCDSTRING when LDAIND returns the terminating null byte in the string. Note that as written, LCDSTRING only works with null-terminated strings of 255 characters or less.

In Summary

Though these routines are quite short, they provide considerable control over the LCD and should serve as a good foundation for any project using an LCD with the 68HC705 MCU. At the very least, they will save you from having to reinvent the wheel just to get your display working. •

Real Computing

Debugging, NS32 Multitasking Trick, and Distributed Operating Systems

by Richard Rodman

Debugging Real-Time Systems

Programmers tend to be vain. Most of them like to write programs that have visible beauty, like Mandelbrot fractals. But then there's the real-time nuts, like me, that insist on writing programs that you can't see working at all. You know you're a real-time programmer when, after hours or days of mind-racking effort, you finally get "hello, world" to appear on the screen, or maybe an LED to turn on, as proof of the functionality of some highly complex but invisible system. Your spouse and/or colleagues shrug and say "that's nice." These people don't understand! Oh, they'd be amazed, wouldn't they, if you did some cheap, flashy graphics trick, or played back a recorded voice or music! What short-sighted dabblers! What superficial dilettantes!

(Ahem) Anyway, debugging techniques on embedded processors and other real-time systems range from the marginally effective to the inscrutable. What's more, the price has little to do with the usefulness of the tool. In the right hands, the cheapest tool can be surprisingly effective.

The most expensive tool is the in-circuit emulator. These tend to cost several thousand dollars and up. Most emulators today use a PC as a user interface. Hewlett-Packard has an emulator for the NS32532, and National has an emulator for the NS32CG16. I haven't had opportunity to use either of these, but I have used a variety of other emulators on a variety of processors. An emulator allows you to stop the program at any time, even upon an external trigger, and examine the state of everything. For particularly tricky timing relationships, throw in a logic analyzer, too. Emulators influence the operation of the target system very little—in fact, if you don't stop the program, they don't influence it at all.

In my experience, emulators are not very useful on larger processors. Because they're so machine-oriented, unless you're using assembly language, it's difficult to make sense of things. The ICE-51 is a terrific product, extremely useful for 8051 development. Of course, only a very silly person would program an 8051 in a higher-level language. Rule of thumb: If you need external RAM, forget the 8051 and use a Z-80.

The next level below this is what's called a "hardware-assisted debugger." An example of this is National's SPLICE board. This is a device which plugs into a CPU socket, has a CPU on it, and gives you some of the capabilities of an emulator. Generally, you can set breakpoints, single-step or trace code, examine registers, query memory locations and ports, but you can't stop the processor on an external condition or at random. These devices may have some influence on the target system's operation; system timing could be significantly altered. They can be very effective once you gain skill in their operation, but they can be frustrating.

The next level below that is a software debugger, such as a monitor with breakpoint capability like TDS. Some software debuggers allow "source-level" debugging, where single-stepping of C statements, examining variables, and so on, are possible. In the Intel world, I'm fond of Soft-Scope—CodeView is too flashy-looking. The VAX has a nice debugger, too. I hope to find or write something like Soft-Scope for the NS32 processor. Anyhow, the drawbacks of software debuggers are that they're bulky to incorporate in a target system, and then they influence the behavior of the machine very heavily.

Finally, the bottom level is what I call the "application-integrated debugger". In simpler terms, a quick print statement, or a character "poked" on the display, or an LED turning on. With care, these can be integrated in such a way as to cause virtually no interference to the target system. An advantage is that this extra output can be obtained without stopping the program--unlike even the very expensive emulator approach. Experienced practitioners can debug almost anything with this method.

But there's one tool that is essential for debugging that can't be bought, and that's *intuition*. Intuition is a tricky thing; sometimes it has blind spots, mental blocks. Debugging is an art, not a science. As such, it can't really be taught. Some good programmers are poor debuggers, and vice versa. So if it turns out you're a better programmer than a debugger, just don't put the bugs in in the first place!

One last thought: Because debugging often gets called into play in "post-mortem" situations, I like to call it "forensic programming."

NS32 Trace Trap Trick

The multitasking executive presented several issues ago in *TCJ* was a non-preemptive multitasker. This means that tasks must voluntarily give up the CPU to other tasks. I have been working on a preemptive version which will ultimately be incorporated into Metal.

Generally, preemptive multitaskers set up a timer interrupt which causes a task switch. The tricky problem here is how to acknowledge the timer interrupt without returning control to the user program that was executing. The method usually used by kernels is to go through a very messy routine which creates a dummy stack image to be processed by the RETI instruction, then falls into the kernel task switch logic.

After studying the NS32 databook, however, a much cleaner (hence simpler and faster) mechanism occurred to me. It's possible to set the trace (T) bit in the PSR image which will be restored by the RETI instruction. Then, after RETI completes, the CPU will perform a trace trap. This unhooks the tick interrupt

logic completely from the kernel task switch logic.

Now, if the trace trap is used for a kernel task switch, this will also provide a foundation for a future single-stepping debugger.

Floppy Disks, Continued

Radio Shack is selling a Personal Word Processor that uses a proprietary 2.8 inch "flippy"—it has two sides, but uses only one side at a time.

The 2.8 megabyte "ED" (presumably, Extended Density) drives are becoming available and are about to be included by IBM on a new PS/2. IBM has lead consumers on a succession of steps, from 360, to 1.2 megabyte, to 1.44, and now to 2.8—another incremental upgrade in capacity, instead of a revolutionary one like the 20 megabyte Floptical drive. Each of these small steps has meant enormous costs in new drives and diskettes, not to mention manual labor in transferring data. Will consumers follow like sheep on yet another incremental, yet expensive upgrade? Of course they will.

PC-532 Update

The PC-532 was designed by the team of Dave Rand *and George Scolaro,* who I neglected to mention last time. There are several of these out there at this time, and Minix has been released for the machine. However, since the machine's only I/O is serial and SCSI, how do you get the operating system into the machine?

There is a board made by OMTI, model 5200, which is a SCSI floppy controller. This board will control up to four floppy drives, which may be any combination of 360, 1.2 or 1.44 types. The board is available from ??WHO?? <--<< for around $150. The PC-532 ROM monitor includes logic to control this board.

The Rumor Mill Grindeth Away

Apparently, Berkeley 4.4 BSD is about to be released. However, it has *not* been expunged of all AT&T code. As usual in academia, the wheels turn exceedingly slow. Carnegie-Mellon continues to work on Mach with the same goal in mind.

Meanwhile, Andy Tanenbaum has been working on a distributed operating system called Amoeba. At the same time, Concurrent Computer has announced a distributed OS called Alpha, and AT&T Bell Labs has recently shown a few folks a distributed OS called Plan 9 (named for the memorable sci-fi classic).

What's a Distributed Operating System?

The basic idea of a distributed OS is that you log in somewhere, and your programs run somewhere, but it doesn't really matter where. The computer system acts as a network of nodes, each of which does a little of the total job. For example, there might be file servers, compute servers, gateways, and so on. The idea of a user being "logged into a particular node" is done away with. Instead, he has a "display server" (like an X terminal), and whatever resources he wishes to use, he does.

The distributed OS is actually an outgrowth of simple network concepts and the client/server model. Most distributed OSs tend to be protocol-based; hence, the underlying nodes may be dissimilar and run a wide variety of different operating systems internally. However, it is advantageous for the underlying nodes to run very simple, fast, low-overhead software.

Next Time

Next time I plan to have an updated report on Minix as well as on Bare Metal. I'd also like to discuss some of the interesting developments in the area of home automation, such as the new X-10 devices that are coming out. Meantime, why not drop us a line care of *TCJ* with that neat tidbit? I don't have any custom coffee mugs or T-shirts, but who needs them, anyway. •

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.

Issue Number 46:

- Build a Long Distance Printer Driver.
- Using the 8031's built-in UART for serial communications.
- Foundational Modules in Modula 2.
- The Z-System Corner: Patching The Word Plus spell checker, and the ZMATE macro text editor.
- Animation with Turbo C: Text in the graphics mode.
- Z80 Communications Gateway: Prototyping, Counter/Timers, and using the Z80 CTC.

Long Distance Printer Driver

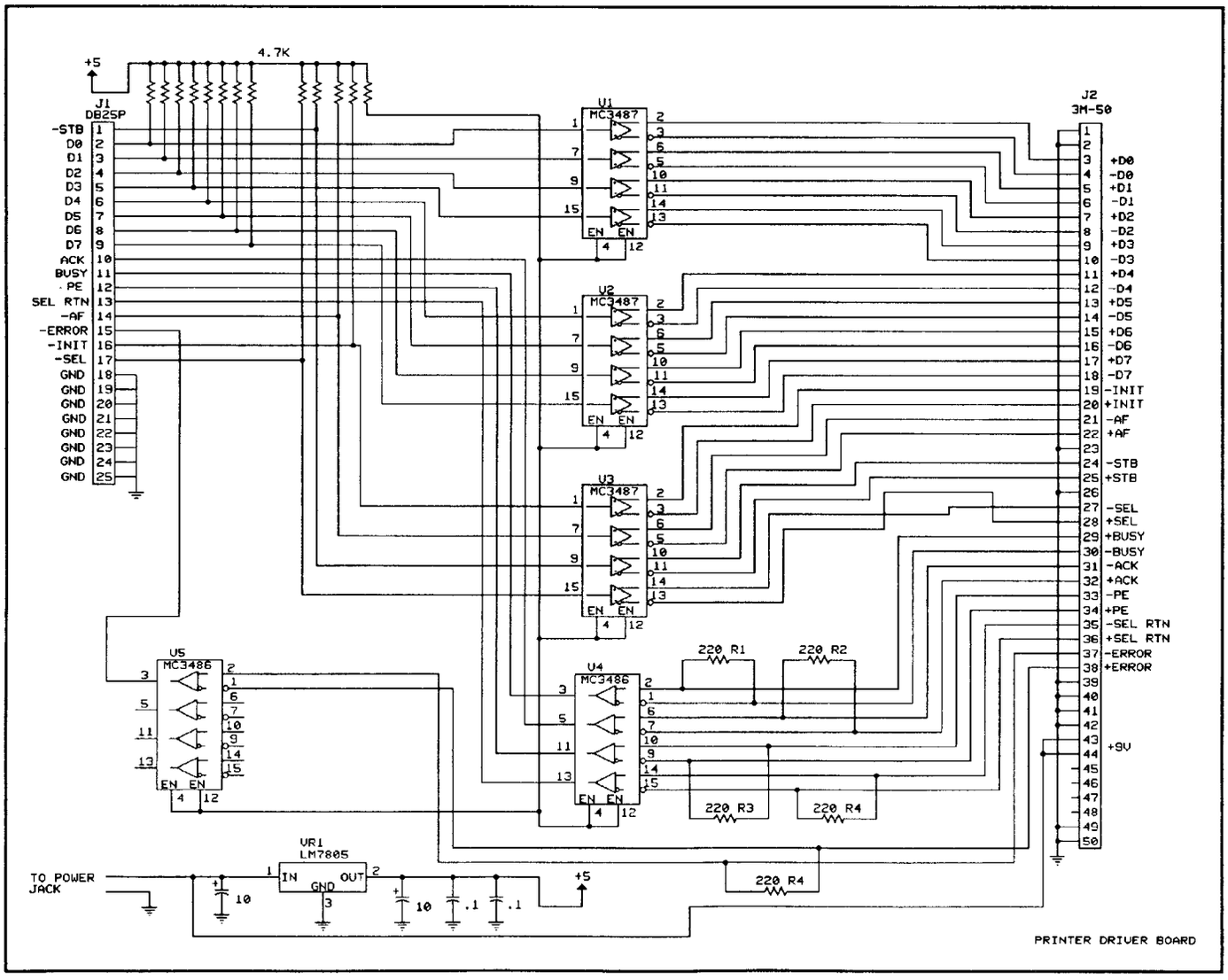
Corrections

The article *Build a Long Distance Printer Driver* by Stuart Ball (TCJ #46) is something that many of us have needed for applications where serial communications is too slow, but the standard parallel interface won't handle the distance. I had been advised to use SCSI for a similar task, but that involved more hardware and software hassle than it was worth to me. The best solution is the simplest one which does the job, and his implementation demonstrates what smart thinking can accomplish.

Unfortunately, I lost part of the schematics in the transfer process. Rather than trying to publish the missing parts with a lot of explanation, I'm including the entire schematics here.

Stuart had included his Schema III files, and since I had Schema II+, I decided to import the drawings into PageMaker 3.0. It took a lot of experimenting plus a call to Omation (the Schema vendor) in order to produce a TIF file which PageMaker would import. Everything looked good, but my libraries did not include the LM7805 voltage regulator. As a result, there was a blank spot in each schematic which I did not notice.

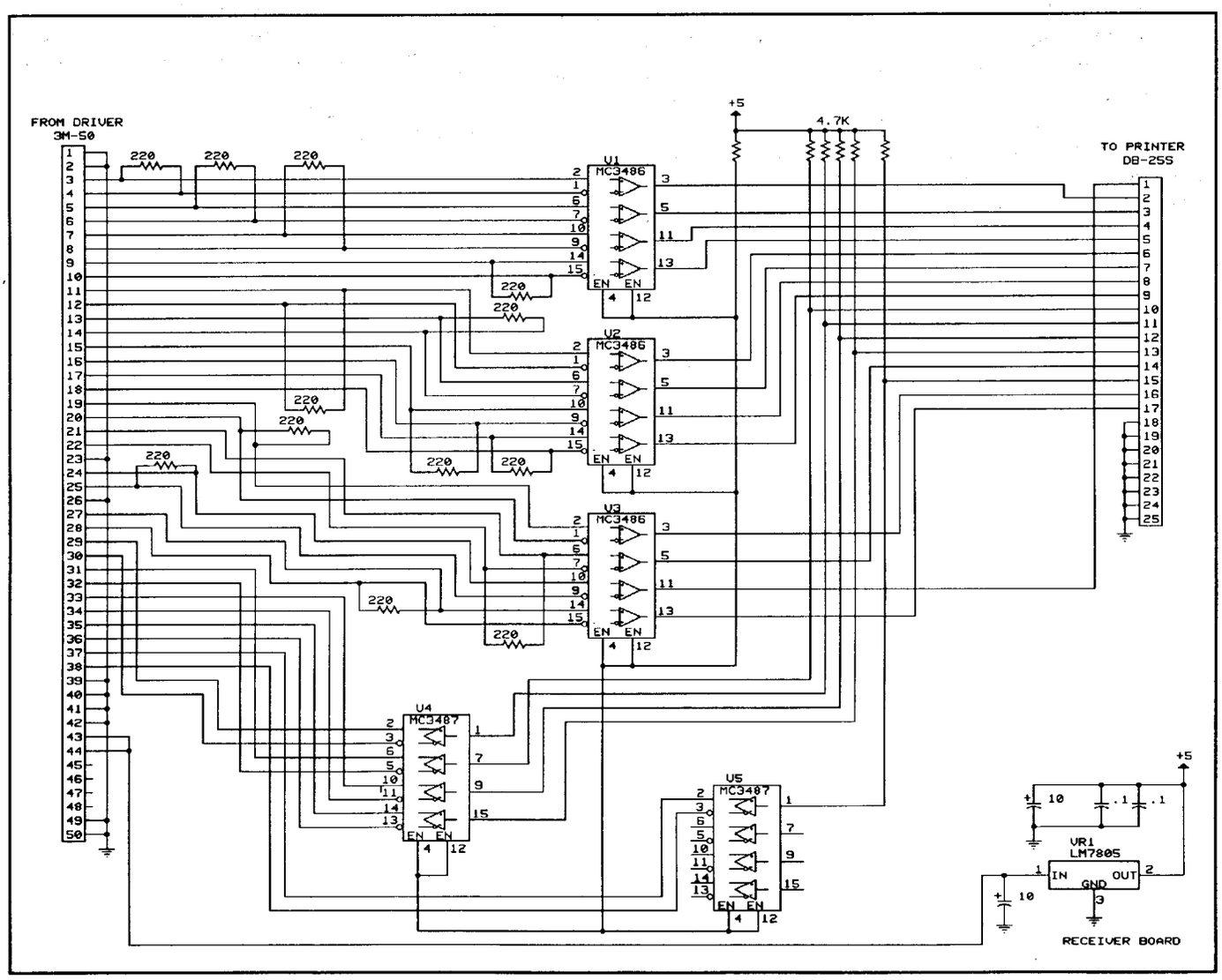
It is usually difficult (or impossible) to transfer files between



programs, but it is often necessary--especially in the publication business where I receive files from many different authors. I am currently using Schema II+ for schematics, and AutoCAD with AutoPCB for printed circuit board routing, but there are also ORCAD and EEDesigner files around.

AutoCAD can output DXF (Date Exchange Format), CDF (Comma Delimited Format), or SDF (System Data Format) files. It can also input DXF files from other sources. I've output DXF files from AutoCAD, converted them to Gerber photoplotter files for the board vendor, converted the Gerber files back to DXF format, and then read the DXF file back into AutoCAD so that I could pen plot something comparable to what the photoplotter will produce. Occasionally we received boards with a missing trace, which was there in the AutoCAD file, but missing in the Gerber files because of a bug in the conversion utility. Checking the plots after converting the Gerber files back to AutoCAD enabled us to pinpoint the problem and to obtain a corrected Gerber conversion utility from the vendor.

I'll be taking a good look at schematic capture and PC board routing programs, including their data input/output capabilities. I'd appreciate feedback on your experiences, especially with regards to file conversions and software techniques for DXF and Gerber files. ●



ROBO-SOG 90

by Michael Thyng

Okay. So what DID happen at ROBO-SOG 90? And what is it anyway? The ROBO stands for the robotics event theme. SOG stands for Semi Official Gathering. The 90 is coincident with this year. SOGs have been annual events for the last 8 years. Mostly, they have been held in Bend, Oregon. But, for the last 2 years they have been regionalized and have varied in location from Denton, Texas and Gunnison, Colorado to up as far as Port Alberni, British Columbia. Others have been held east of the Mississippi, but I can't remember where.

ROBO-SOG 90 was an informal, educational event with a robotics theme. We met at the Applied Technology Training Center (ATTC) in Everett, Washington. This is a college-like facility used by manufacturing firms to train and upgrade employee skills in intensive, generally short term, training sessions.

We began our first day with an informal Junque de Jour tour in Seattle. We visited three local personal robotics suppliers (here read 'surplus stores') and natives showed newcomers places for the best bargains. At the end of the day, we had a barbecue "out back" and listened to John play the guitar. I don't know if the music was better before or after we started singing with him, but I have a theory.

Friday morning (the second day) we began with the only formal portion of the event. Jim Wright, the guy who helped us land the ATTC for ROBO-SOG 90 gave the opening speech and welcomed the attendees. Then began a succession of one hour seminars, both morning and afternoon.

Our first speaker demonstrated an industrial robot. Yes, right there in the ATTC! Then an Intel representative talked about embedded controllers. Suzanne Weghorst of the University of Washington told us what's what about Virtual Reality. Thomas Anderson, who wrote the TASM cross assembler, described how it came about and the probable future of his work. Ward Silver described the trials of being a successful consultant. That fit beautifully with Bo Ray's descriptions of how he helped build up his company, Rapid Systems, into the success it is today. It just worked. Dr. Sandy Spelman explained in an hour about his work with cochlear implants. This is an on-going study to help the profoundly hard of hearing to hear with surgically implanted electronic devices. We slimmed ourselves by limiting him to only one hour. The day concluded with Gary Godecke's "show and tell" of computer aided manufacturing and the small business.

What did other members of the family do when one of the members was so heavily involved in this electronic talk? We had three totally non-electronic events, quilting, origami and simulated robot building.

After sessions, we scattered to a variety of places for dinner, then reconvened for Jolt Sigs. A Jolt Sig is a chance to visit with the speakers, and other attendees into the wee hours of the night. Sig stands for Special Interest Group. Jolt is the name chosen to honor the heavily caffinated soft drink consumed to help keep us awake after the coffee pot is turned off.

Saturday morning, tech sessions began at 9:00. Attendance was sparse until part way through the 10:00 session. Uh-huh. We stay up late talking, then don't manage to get up quite when we need to. We should probably apologize to out early morning speakers for the light attendance, but one of them was among the missing at start time.

Randy Young talked about his vision systems applications in a sawmill. By inspecting the logs by camera and analyzing the images with a computer, they can choose the best way to run the log through the saw. Art Horne described some new electronic boards he has produced which take advantage of Forth. Fred Martin, from the MIT Media Labs, talked about how he has combined the LOGO language with LEGO building blocks and made little robots. Al Ross showed his specially designed wheelchair with a computer built into it, which allows quadriplegics to communicate even with minimum appendage control.

After Saturday lunch, the tech sessions closed with a robotics panel discussion. There were about 10 home made robots, and their inventors were proud to have a chance to tell all about them. The speaker shark (someone who lets the speakers know it's time to stop talking) limited their dissertations to about 10 minutes each or we might still be there. That was as lively a session as any held during the rest of ROBO-SOG 90, and a fitting way to make the transition from lecture sessions to practical application. We had set up the Seattle Robotics Society Maze, you know, "out back" and organized a get-it-through-the-maze contest. Murphy, the truly brainless robot, finished third. Karl's camera equipped robot finished second, and Fred's LEGGO chassis robot finished first. My robot didn't embarrass me with a truly poor performance, but it came close.

One thing I learned was how much easier it is to build a robot when you use LEGGOs. They are light, so the massive battery needs go away, and you can carry a completed robot around in a fairly small box. If I told you how many weeks I spent trying to make my aluminum robot work, and trying to provide power to those big motors, including the painful drilling incident, you would only laugh.

We closed with a banquet dinner of salmon, chicken, rice, some orange and green stuff, and a mint ice cream with chocolate dessert. We handed out awards, patted each other on the back and went our separate ways. And, you know what? Only one speaker didn't show. For an

event with this many speakers, that was pretty darn good.

But there are two other things I'd like to mention.

1) *Micro Cornucopia* magazine went out of business early this year. I'm sorry. Among other things, I may have lost the only contact with SOGs held around the country. [But if you're reading this, that's obviously not true.] If you went to a SOG this year, would you make contact with either me or *The Computer Journal* and tell about it? I'm sure others would like to find out what happened too.

2) There will be another robotic event in July, 1992. To say it will be big is a gross understatement. If you want information, I am:

Michael Thyng
11036 40th N.E.
Seattle, WA 98125
(206) 362-5373 (voice)
(206) 362-5267 (modem 24hr)

Editor

(Continued from page 2)

before file extensions and placing a zero before decimal fractions. Some file filter/conversion programs may also exhibit this problem.

My printer driver will be similar to the UNIX-like MKS PR command where all commands will be in the command line or a config file. I need to be able to select header details (print date, file creation date, etc.), lines per page, single or double spacing, compressed or regular type, page width, etc. without altering a programming source file. While I prefer assembly language, this looks like a good application for C, because the speed will be limited by the printer I/O and it only need a small disk data buffer. Does anyone else have strong feelings about printer drivers?

Borland Bargains??

Borland started by selling Turbo Pascal direct at a very low price. It seems to me that it was something like $49.95. In recent years Borland's prices have escalated and they announced to dealers that there was enough profit margin for the dealers to handle Borland products and that Borland would refrain from direct sales so that they would not compete with their dealers.

But I've been getting many special offers from Borland (as many as five in one day), such as Quattro Pro 2.0 for $99.95 instead of $495 and the ProShow PowerPack for $39.95 instead of $300. Then there's Turbo C++ Professional for $149.95 instead of $299.95, and a similar bargain for Paradox. This makes me wonder about the success of their dealer sales program.

It would be very interesting to see a direct vs. dealer sales ratio study for the past four years. Borland has also dropped BASIC, PROLOG, MODULA-2 (yes, there was a CP/M version), and I believe SPRINT. All this makes me wonder how Borland is doing and where they are headed—any Borland watchers out there who can fill us in on this?

Jameco

I remember Jameco as an inexpensive source for surplus and odd-lot component bargains. Last year their catalog increased the emphasis on computers and their minimum order was $25, but I still used them for most of my purchases. Their current catalog has full page four color ads for computers, with a few pages of components which have been shoved way to the back. Their minimum order is now $50 which is too steep if you need a few chips for a project.

I'm looking for a new source. Digi-Key (1-800-344-4539) looks good for most components. They have a great selection of connectors, resistors, capacitors, transistors, SCRs, etc., but they don't carry Motorola or Intel microcontroller and peripheral chips. They have a toll-free order line, no minimum order with a $3 service charge for orders under $25, and your order is entered on-line so that you know if the items are in stock. I'll be using them for most of my orders (except, unfortunately Motorola and Intel)—get their catalog (Digi-Key, PO Box 677, Thief River Falls, MN 56701-0677), you'll be surprised how helpful and friendly people in small towns are.

Floppy Disks

Richard Rodman explained floppies in his article *Mysteries of PC Floppy Disks Revealed* in issue #44, and somewhere I believe that he mentioned the interchangeability problems of 360K and 1.2M disks. It appears that there are problems when you format and write to a disk in a 360K drive, then reformat the disk (as 360K) in a 1.2M drive. Many 360 drives will not reliably read the disk. Apparently the 360K drives write a wider path than the 1.2M drives and when reformatted as 360K in a 1.2M drive some of the old data remains at the edges of the track. A 360K drive may read both the 360K and 1.2M data, which results in garbage.

Lee Hart, in *The Staunch 8/89'er* says, "Don't reformat a disk with a different format or number of tracks unless you bulk erase it first. Use a bulk tape eraser, or rub (the disk, still in its sleeve) with a permanent magnet if nothing else is available." •

sheet. One of my complaints has been the need to go back and forth between several different manuals to find facts. GA port 0 is address 18 hex and ONLY stated as such in the specification sheet, nowhere else! The entire 2 inches of programmer manual only gives you an example of how to interface more memory to the board. There is no help anywhere in the book or specification sheet on interfacing to peripherals. If you intend to use this to drive anything, you must build an interface without their help. I was told that several reprints on how to talk to peripherals is in the works and may be mailed by the time you read this. As a contestant, if you didn't have the schematic of their development board like I did, my guess is that you would have either given up or had some problems. I say that only because the schematic indicates that some timing problems on the read cycle required delay devices.

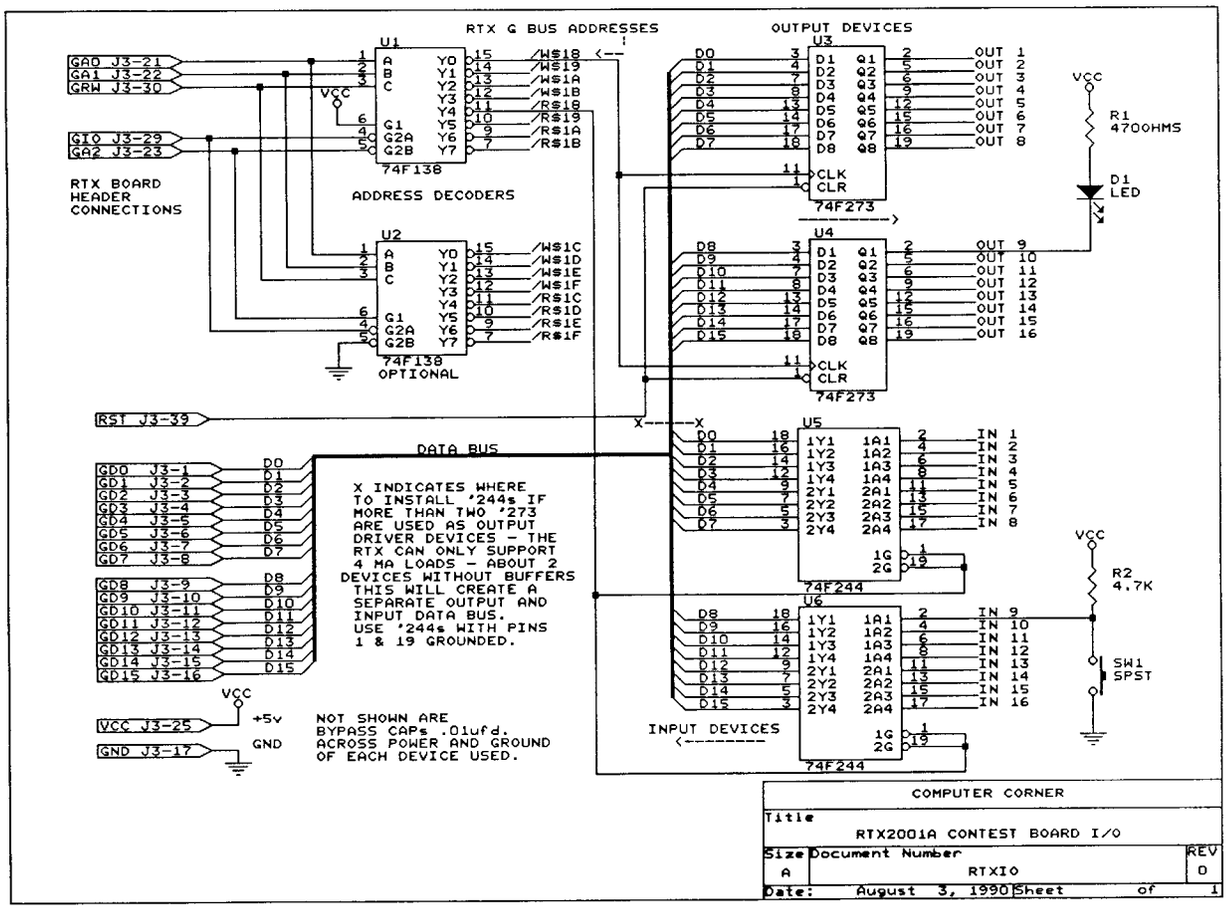
Doing I/O

What my intentions are, is showing how doing a simple task in RTX FORTH compares with doing the same task using a 6805 device. For the sake of this column I have kept it real simple. Figure 1 shows the interface I built for the RTX contest board and lists the header pins to get the signals from. It is not necessary to build all the board, as I only wanted 8 bits to do testing on and only installed devices U1, 3 and 5. This also explains why I made the '138 address decoder split out the read and write operations. This is pretty much a minimum arrangement and has not been tested fully to see if some timing problems might appear. What testing I have done says it should work reliably for most things. Also note that using more than one or two '273s will require splitting the data bus into two buses with '244s used as bus driver (see the X comment and indication in the figure).

What it shows us doing is reading a switch and turning an LED on or off depending on the switches status. For real world type of work this is probably the most common operation performed. If your system can't do this, I would not consider it as being a controller, especially an industrial controller. To do the same thing with an 6805 would not require the devices shown in Figure 1. The 6805 port A could acts as the input port, while the B port has the ability to drive LEDs directly (I would watch the size of the resistor used so as to limit current to less than 10 MA).

Our listing shows the difference between the two codes needed to read the switch and turn the LED on. What is not listed is any initialization code. The 6805 would need the port set up for being either input or output operation. Also not shown would be the master loop used to call these routines. They could be tied to interrupts or sampled on a timer tick, but that sort of decision remains in the imagination of the programmer.

I checked the RTX generated code and a 16 bit word is created for each word used, plus the header name. That means the LED_ON word used about 14 bytes (7 words) of the 3200 bytes of available dictionary space. Our same operation in the 6805 used only three bytes. To further the comparison we need look at how the code would be used. In the 6805 you have to assemble the code and try it. Should it fail, you will need to reassemble your guesses and burn a new chip to try again. I do know of a FORTH based 6805 development product, but for now you must assemble, test, and reassemble. The RTX however allows you to test the operation in interpretive mode first, and then only when the desired results is achieved do you compile it into the dictionary. I did



```
RTX EBFORTH

HEX  (NEEDED TO USE HEX VALUES)
: READ_PORT  ( - N )
  18 G@ 100 AND ;
( MAKE HI IF SWITCH OPEN )


: LED_ON     ( - )
  FEFF 18 G! ;
  ( NEED LOW TO TURN ON LED )

: LED_OFF    ( - )
  FFFF 18 G! ;
  ( TURN ALL OFF/HI)

: SW=LED  ( - )
  READ_PORT
  IF   ( >0 = OFF )
    LED_OFF
  ELSE
       ( 0 = ON )
    LED_ON
  THEN  ;
```

```
6805

READ_PORT  EQU  *
    LDA  PORTA  DATA TO A
    COMA        COMPLEMENT
    AND  #$0    BIT 0
    RTS
LED_ON     EQU  *
    BCLR  0,PORTB  0=LED
    RTS

 LED_OFF   EQU  *
    BSET  0,PORTB  HI=OFF
    RTS

SW=LED     EQU  *
    BSR    READ_PORT
    TSTA
    BEQ    SW=LED1
    BSR    LED_ON
    RTS
SW=LED1    EQU  *
    BSR    LED_OFF
    RTS
 (a faster SW routine is
    BRCLR  0,PORTA,SW1
    BSR    LED_OFF
    RTS
 SW1  EQU   *
    BSR    LED_ON
    RTS
  read_port not needed!)
```

just that for the code shown, discovering that EBFORTH's NOT is actually a 1's compliment and does not change 0's to one's as is the case in some Forths. I can't stress the importance of being able to pretest your modules before putting them in your program. The pretesting for me is worth any other inconvenience it might (but doesn't) produce.

Review

In summing up this column I would say I have found the RTX manual to be excessive and not that helpful. The appendixes are probably what you will use the most. The RTX contest board is a nice toy and with an added interface board could do considerable work at a reasonable price. Should enough interest be shown, I can make I/O boards available for a small cost (a project yet to be started). In comparing the 6805 to an RTX, I would say the 6805 has the RTX beat in generating compact code and easy interfacing to the outside world. The RTX is probably a better development platform for more complex projects, and if HARRIS produces an RTX with built in I/O like the 6805, I might chose it over the 6805.

I guess the bottom line then goes this way, 6805 for small projects with simple I/O and limited parts count. The RTX for more complex or difficult projects. Projects requiring field changes or options would definitely go to the RTX.

Next Time

I have run out time again, so next time more on the MINIX operating system. Till then, happy programming and hacking. •

For RTX information contact:
HARRIS at 1(800)-4-HARRIS, ext 1301

For information on MINIX contact:
Prentice Hall publishing
1(800)-624-0023

The Computer Corner

by Bill Kibler

It is a busy time again with plenty to talk about. Lots of little things have been happening and I will complete the second part of our RTX review. Let's start by reviewing a UNIX clone.

MINIX

For those interested in learning UNIX operations, there are plenty of DOS packages that will teach you how to use the utilities that come with the standard UNIX. However if you are like me and want to learn how to write drivers and programs to work under UNIX something else is needed. You see I have applied for several jobs over the past few years and my limited work with UNIX has been a problem. So finally I had some extra money and decided to break down and buy MINIX.

Why MINIX? MINIX is an UNIX substitution program developed by Tanenbaum (a teacher of operating system design) that can be run on PC type computers and the Atari ST. There are a few other advantages that come with the package, mainly the entire source code. Not only do you get a book explaining most of the why and wherefore of the operating system, but you get all the source to everything (and I mean all 100 utilities as well). The features really don't stop there, unlike UNIX it will run without hard disk, doesn't need megabytes of memory, and could be used by normal programmers.

You might have noticed I said "NORMAL PROGRAMMERS" as my only problems so far has been lack of information for the real novice user. I have had considerable troubles with the AT version and will let you know later about solving those problems. The Atari ST version however came with a small book (no how it works book with it—you buy it separately) that is set up for a more experienced novice user. You will also find that some of the cheaper (as outdated) books on version 7 UNIX will explain how the utilities work. I find the entire supply of documentation set up for the operating system student who has already been using a real UNIX system. For those who are a little rusty with UNIX commands and operation (like me) or a novice user I think some other package would be more useful.

The PC versions have a DOC file that contains most of the information contained in the Atari book, but I had considerable trouble getting it printed from MINIX. I had to finally use DOSWRITE to move it to a PC disk and print from regular DOS. MINIX has a UNIX style disk format so you can not move files without using the special programs. At first that really turned me off, but as I have learned more about how things work, I can understand and accept the reasoning involved.

Since I have only gotten started on MINX I will leave a more complete review for later. My objective is to learn the inner operations and driver interfacing needed for UNIX like operations. I am also looking at using the system on several different platforms (PC and 68K) in hopes of being able to have ONE operating system for all my different types of hardware. Failing to like MINIX for that purpose it will leave me with FORTH as the only way to have one system for many hardware platforms.

RTX

The FORTH on the RTX is not in the running for use on all my hardware platforms. I did find EBFORTH adequate for the evaluation board. In last month's corner, I covered the board you received in the contest unit and how it wsa basically setup. The manual is rather large and at the time I thought more than adequate. My experience in trying to build a hardware interface to the board proved that the manual has many short comings.

The manual is primarily setup to give you programming information, too much programming information in fact. As I started using the manual for real, I discovered major problems with finding the facts I needed. My current feeling is that the entire book could be cut down to about half the size with several new sections needed. Those new sections have to do with hardware. The main problem with the board is NO interfaces to the outside world are provided except the serial link. If you want to learn about the RTX by running it as a slave serial port on your PC you will be just fine. If however you are like me and want to have your unit talk to something as simple as LEDs you got problems.

Our local FORTH interest group had a visitor many months ago, who supplied us with copies of the RTX development information package. In that package were schematics of the unit. By looking at those I was able to design and build a parallel port interface for testing (see Figure 1). At first I felt Harris had not even provided current output abilities of the RTX. After calling their technical support people I found out the printed specs of 4 MA maximum per output was correct. That means you can only put a few (as in one or two) logic devices off of the ASIC bus. To be able to talk to anything, you must first build an address decoder and bus drivers for the ASIC bus. The other problem which slipped by me at first was the ASIC bus itself.

In the NOVIX (which I have and use) there are the B and X ports. These ports can handle 30 MA (easily drive LEDs) and can be latched up much like any parallel port. I had correctly understood that the B and X became the ASIC bus on the RTX. The point I missed was that all the new and some old registers are now directly tied to this bus at all times. The B and X ports are now a FULL TIME BUS that talks to everything inside and outside the chip. You have a simple FORTH command to use the bus (G! and G@ to get and put data to it) but you must be careful. There are 32 addresses on the bus and the lower hex 17 are all internal and can crash the system when improperly used. I know as I wrote to them by mistake several times.

The information on which addresses do what is contained in the manual and the RTX2001A manufacturing specification

(Continued on page 34)