CONTROL THE WORLD WITH HP-IL

By
GARY FRIEDMAN

A New World Of Uses For Hewlett-Packard Handheld Computers

CONTROL THE WORLD WITH HP-IL

A New World Of Uses for Hewlett-Packard Handheld Computers

By
GARY FRIEDMAN

Pictorial Artwork by
Steve Luchsinger

Published by **Synthetix**
P.O. Box 1080
Berkeley, CA 94701-1080
USA

Library of Congress Catalog Card Number: 87-60210

ISBN 0-9612174-9-9

To my parents,
who during the course of this book
have forgotton what I look like.

Special thanks to:

Ken Emery, David Erbas-White, Michael D. Varnen,
Sergio Morales, Jason and Jenni Levine,
Ruth Brodsley, and Brian Ramage.

TABLE OF CONTENTS

FOREWORD

There is probably no group of people in existence that can appreciate all of the 41's capabilities more than college students who must constantly crunch numbers. As opposed to professionals who use it for a handful of things in their day-to-day activities, students rely on it most of the day to help solve an incredibly diverse set of problems. And the more one is introduced to new mathematical applications, the more one can appreciate the power and versatility of this simple-looking machine!

But most people never see the power of both the HP-41 and HP-71 pocket computers beyond their obvious number-crunching capability. With the addition of a deceivingly simple-looking interface, these same calculators are transformed into powerful controllers that can interact with over 900 devices in the outside world simultaneously! Coupled with their characteristic small size, continuous memory and low power consumption, they are uniquely suited to field applications that everyone else's IBM or Apple would be too bulky, noisy, and cumbersome to implement.

Control the World with HP-IL illustrates the interface capabilities of the HP-41 and 71. It shows useful and (to be certain) unique functions that exploit these machines' unique properties and offers enough background so you can design solutions to suit your own applications.

Many chapters in this book provide complete information for you to duplicate what's presented; others are designed to give you enough background information so you can implement your own modifications. Or, as is the case in Chapter 8, enough information for you to convert whatever is available into something quite

useful.

This book assumes the reader is reasonably familiar with either the HP41 or 71 handheld computers, and has a very basic knowledge of digital electronics (i.e., knows what a gate is), and is confident about their ability to solder and disassemble things. Although I do try to describe the circuits at a detailed level for those who are unfamiliar with electronics, experience and experimentation are the best tools to insure project completion, and most of all learning.

Chapter One

THE BASICS

"There are only 2 kinds of people:
those who own Hewlett Packard calculators,
and those who say, 'Where's the equals?'."

--G. Friedman

This book is centered around 2 pocket-sized units you can use to control the outside world: the Hewlett Packard 41 and 71 computers. Do not let their size fool you; both machines were light years ahead of their time when they were introduced.

I prefer these small controlling machines to their more famous larger counterparts for some fairly fundamental reasons: 1) for control applications, they are just as powerful as the larger machines, (even more so when the 71's math capabilities are compared), 2) HP-IL (Hewlett Packard Interface Loop) gives you an order of magnitude more Input/Output (I/O) opportunities than ANY personal computer, and 3) you can use them in the field, as they do not need to be tied to an AC outlet. This last point is especially important if the devices you are controlling are battery operated, such as the time exposure camera controller of Chapter 3.

There are many who might say these machines greatly lack the speed necessary for any application involving instant response to an event, and therefore should not be bothered with. To this I say HA!!!, and I offer as evidence the chapters covering ultrasonic distance measurement, the intelligent autodialer that can recognize ringing and busy signals, and the slide projector dissolve unit, which varies the intensities of 2 high-wattage lamps

using pulse-width modulation in real time! (Don't worry; these terms will all be explained clearly later.)

The Machines

In order to demonstrate versatility, two handhelds will be utilized in this book: the HP41C/CV/CX calculator, and the HP71B BASIC computer. Because it is unlikely that any person would own both, brief descriptions of each are presented here, and some of their strengths and weaknesses can be explained, and then later on exploited.

The HP-41

Possibly the most powerful handheld calculator ever made, the 41 was designed to be efficient, versatile, and easy-to-use. A numeric-entry system called RPN (which stands for Reverse Polish Notation, giving credit to the origins of this method) allows users to get answers using fewer keystrokes, bypassing the traditional nested parentheses and "equals" keys. Most people are put off by this feature because it does take a while to learn, but ask anyone who has invested the few hours it takes to master it and they will tell you about the quantum level of superiority RPN has over any other method.

This characteristic is further enhanced by programmability, a feature that is so straightforward on the 41 that learning RPN and learning how to program are practically the same thing.

4 expansion slots, or "ports", exist on the 41 to add to its existing built-in library of over 200 functions, and to give it extra capabilities such as timer and alarm functions or the ability to communicate with peripherals.

The 41 comes in 3 flavors: The just-described 41C, a unit so basic that it's no longer being made; the 41CV, which is identical to the 41C except that it has the maximum amount of memory already built in, thereby freeing a port space; and finally the 41CX, which builds on the CV by adding a Time module, Extended Functions/Extended memory module, and even throws in some extra functions never before seen on a 41. The unit you choose

depends on what level you plan to exploit your calculator and how many port spaces you plan to occupy while exploiting it. Configured properly, any flavor 41 provides a tool, forever at your disposal, that can solve your unique problems more easily than any other machine available today.

The Extra ROMS

The 41, however, initially has several shortcomings when it comes to acting as a controller. These stem mostly from the fact that it was the first machine to implement HP-IL, (it's murder being a pioneer!) and back then the designers had only envisioned an interface that was so user-friendly that people wouldn't NEED to have low-level control of the loop. The 41's HP-IL was initially introduced with 2 products designed to work together: the 82162A thermal printer and 82161A digital cassette drive, and if these were your only 2 peripherals, you'd swear that the system was well-implemented.

Well, suddenly other HP divisions started producing HP-IL products that actually adhered to specifications (like responding to "Send Device ID" requests, something the original printer and tape drive didn't do), and it was clear that new functions had to be provided. HP then came out with 2 plug-in ROMs as an afterthought: The IL Development ROM, which gave absolute control to those who knew what they were doing, and the Extended I/O ROM, for "the rest of us". AT LEAST ONE of these ROMS is mandatory for just about every project described henceforth. In most cases it doesn't matter, except for Chapter 4, which requires the Development ROM. (An explanation of HP-IL and how this ROM is used will also be presented later.)

The other shortcoming of the 41, the inability to quickly load the ALPHA register (which doubles as the Input/Output register when communicating with the loop) with characters other than A-z or 0-9, only slows the machine down and greatly inflates the program size. For example, let's say you want the thermal printer to go into Double wide mode (normally done by setting flag 12) by sending an "escape sequence". A quick check on page 13 of the thermal printer's manual reveals that this is what the printer will

expect to see:

```
ASCII Character sequence: {escape}  &  k   1  S
                             |      |  |   |  |
Decimal Equivalents:          27    38 107 49 83
```

And using XTOAR (append X to Alpha on the Right) with an Extended I/O ROM, (or XTOA with the Extended Functions ROM), this is what it takes to send this sequence:

```
01 CLA          09 XTOAR
02 27           10 83
03 XTOAR        11 XTOAR
04 38           12 SF 17
05 XTOAR        13 OUTA
06 107          14 "ABCDE"
07 XTOAR        15 CF 17
08 49           16 OUTA
```

Lines 1-11 put the necessary escape sequence into the ALPHA register. (Line 12, SF 17, is needed to suspend the normal Carriage Return/Line Feed normally sent via OUTA.) Try running the program and stopping at line 11. The ALPHA register will contain what the escape sequence looks like: GARBAGE! This is one of the reasons these characters cannot be keyed in the normal way.

Synthetics to the rescue!

Synthetic text lines contain instructions that cannot be keyed into program memory by conventional means, but will execute flawlessly once they are there. Their history is a little spectacular, as they were discovered and nurtured by user groups and their existence denied (for the first few years, anyway) by HP. Synthetic

instructions in general allow any user to access internal registers, pointers, noises (127 tones, rather than just 10), and display characters. They allow the 41 to do things that aren't normally within the machine's realm.

In this book, synthetics will be used for one thing: loading the ALPHA register quickly. (Okay, two things: In Chapter 4 we need to suspend all Time Module alarms and synthetic techniques are the ONLY way to do this.) There are currently two best ways to load synthetic instructions into the 41. The best is via the ZENROM, a plug-in accessory produced in Great Britain that makes entering synthetics as easy as normal instructions. The second best, and possibly most common, method is by the LB (Load Bytes) program which is found in the PPC ROM. Both programs take as input numbers between 0 and 255, and insert them into program memory. A valid 41 instruction then appears in that space if a proper number sequence has been entered.

If you are not familiar with synthetics, there is no need to learn them. Appendix A contains barcode of every HP-41 program listed in this book, so the entire program, synthetics and all, can be loaded with the Wand. Additionally, all the necessary program bytes for creating synthetic instructions are included in each program listing, for use with the utilities provided by either ZENROM or the PPC ROM. If you would like to learn more about synthetics, there are a couple of highly-recommended books on the subject:

HP-41 Synthetic Programming Made Easy by Keith Jarett
Published by SYNTHETIX, P.O. Box 1080
Berkeley, CA 94701-1080 USA

Synthetic Programming on the HP-41 by Bill C. Wickes
Larken Publications
4517 NW Queens Ave.
Corvallis, OR 97330 USA

HP-41 Instruction Summary

I'm sure many who own both the 41 and the IL module are

wondering exactly how these two items are used to talk to items other than printers and tape drives. Well, just to quench your thirst for knowledge, here's a table which summarizes all the instructions available in the 41's IL Module that can be used to send data to an 8-bit port. (This list does not include the supplemental instructions provided by the Extended I/O or IL Development ROM.) The 41 must be in MANIO (Manual I/O) mode when these are used, otherwise most commands will try to route the data to a printer.

OUTPUT

ACA
: Sends alpha register contents to the 8-bit port as a string of ASCII characters.

PRA
: Same as ACA but terminates with an End-of-Line indicator. Some special non-alpha characters, such as decimal 13 and 126, are not transmitted properly.

OUTA
: Same as ACA but sends an End-of-Line indicator only when flag 17 is clear.

ACX
: Sends X register contents to the 8-bit port as a string of ASCII characters using the current display format.

PRX
: Same as ACX but terminates with an End-of-Line indicator.

ADV, PRBUF
: Sends End-of-Line indicators to the port.

ACCHR
: Will send any one character specified by its decimal equivalent in the X register. For example, to send the Greek character (mu) or to send the binary word 00001100, put "12"

into the X register and ACCHR. Any character code up to 127 can be sent except 10, 13, and 126. (See text.)

TRIGGER
On the GPIO or the IL Converter, pulses the GETO (Group Execute Trigger Out) line low for a brief period.

INPUT

INA
Fills the alpha register with up to 24 ASCII characters from the 8-bit port. If flag 17 is set, the calculator does not wait for a CR/LF (Carriage Return/Line Feed) to terminate the incoming string of characters.

IND
Interprets the next incoming ASCII encoded word as a number and places it into the X register.

INSTAT
Places the first status word into the X register and sets flags 0-7 accordingly. This is the only way to test the 8-bit port's MSRQ (Manual Service Request) line.

The HP-71

The HP-71 is a rather deceiving machine. It was originally designed to be a more powerful replacement for the 41 (it was originally called the HP-44), and I guess that during the development process they just got carried away. Contained within its tiny dimensions are 3 programming languages (BASIC, 20-bit FORTH, and Assembly), an advanced CALCulations mode, the world's best HP-IL, and a 512K address space. It's number-crunching capability is unparalleled for anything of its size; in fact it adhered to the IEEE floating point math standard even before it was a standard!

But I am not trying to be a salesman. Blatantly missing from the 71 is the RPN environment raved about a few pages ago; it has

been replaced by a more sophisticated CALCulations mode. (HP does sell a plug-in ROM that turns the 71 into an incredibly fast 41, thereby closing the gap somewhat.) The most impressive aspect as far as this book is concerned is its implementation of HP-IL, which is the world's best. Offering both high- and low-level control (something the 41 had a hard time doing), it has the capability to act as a device as well as controller, pass control to another device on the loop, wake the 71 up to process important frames, or pass unimportant ones through while the machine stays off.

If you are a newcomer to HP equipment and especially to HP-IL, the 71 will make for an easier transition than the 41 because 1) it's programmable in a very robust implementation of BASIC, a language that marketing people seem to think is so obvious to use that everyone is born with the intuitive knowledge of it, and 2) because of its superb HP-IL interface, it is much easier to manipulate devices on the loop and have absolute control over what's going on.

Just as the 41 can't communicate without its IL Module, the 71 must have 2 peripherals to allow it to interface with the projects described in this book. The first is the IL Module, which is absolutely mandatory for obvious reasons. Almost as necessary is the FORTH/Assembler/Debugger ROM, which practically doubles the power of the 71. Programming in FORTH gives you absolute control over the machine at 10 times the speed of BASIC, and is an easy language to learn if you're already familiar with RPN on the 41. But if FORTH isn't for you, then there's an assembler that allows you to program in assembly language, giving you the freedom to write your own commands and extend the BASIC or FORTH languages to fit your own needs. (Assembly language can also be used to bypass HP-IL and control the outside world at lightning-fast speeds, as demonstrated in Chapters 9 and 10.) The Debug utilities put the icing on the cake, allowing you to look inside the machine's registers and see why that simple little assembly language program didn't work right. Learning how to get the most out of this ROM takes time. But then again, the same goes for any good tool. In short, the 71 without the FORTH/Assembler/Debugger ROM is like a Jeep without 4-wheel drive.

Equally handy are the ROM's line editor, called EDTEXT, and its KEYBOARD IS lexfile, an assembly language routine that allows you to hook up an external keyboard to the 71 so it can be used as comfortably as the big machines. (See Chapter 8 or 11 for examples of external keyboards using KEYBOARD IS.)

HP-IL Introduction

HP-IL is one of those unique solutions that, like the handheld units that drive them, fit such a specialized need so perfectly it's hard to imagine anything that compares to it in terms of speed, versatility, and especially power consumption. A typical loop setup is shown in Fig. 1-1.

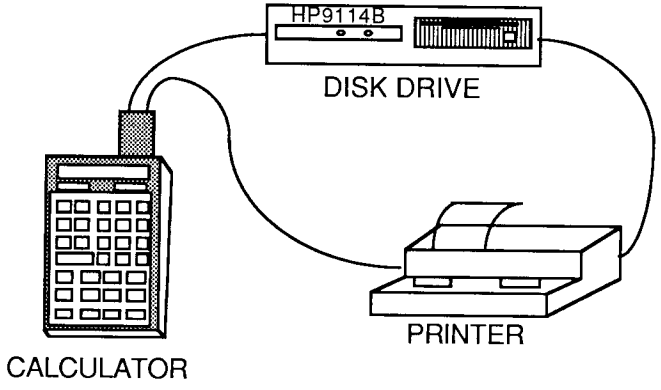


Figure 1-1
Typical Loop
Configuration

Very basically, all the wires between devices form a loop. Messages sent out are passed along from one intelligent device to the other. In this case 'intelligent device' means they each have

built-in microprocessors, so they can react to messages in a meaningful way. Eventually the message reaches the sender, which then compares the returned message against the one sent to check for errors. Each device then tries to act on the message it had just copied, but most discard it because it wasn't meant for them.

HP-IL's simple physical layout is also responsible for these attributes:

1. Low power, allowing operation on batteries for extended periods of time.
2. Versatile, can set up multiple listeners for mass printing or tape duplication. Imagine 930 LaserJets printing at the same time! (Imagine the cost!)
3. All devices can be selectively controlled. With normal addressing, up to 31 devices can be connected to a controller. With extended addressing, over 900 devices can be accessed!
4. Average data rate is comparable or faster than RS-232.
5. Devices can be separated by as much as 10 meters with special twisted-pair IL cables.
6. Devices can be powered down and powered up by remote commands, meaning the entire battery-operated system can conserve its energy until needed. (Not all devices possess this capability.)

Getting into the full details of HP-IL is too complicated to cover completely in this one section. Rather, this chapter offers a beginner's introduction to HP-IL, encompassing everything you need to know to understand what goes on in the rest of the book. If more information is desired, I highly recommend another book:

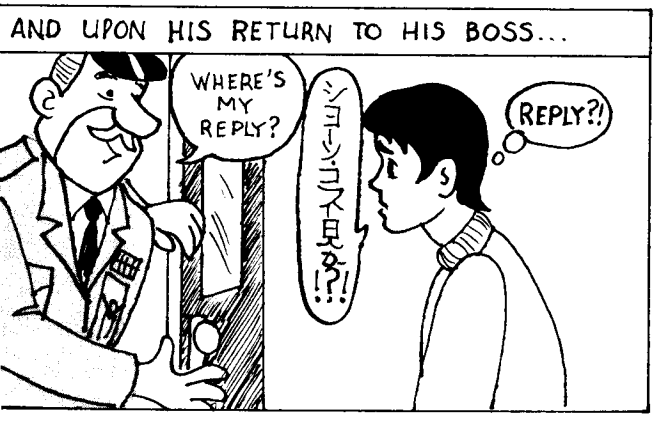
> THE HP-IL SYSTEM: An introductory Guide to the Hewlett-Packard Interface Loop by Gerry Kane, Steve Harper, David Ushijima; Osborne/McGraw-Hill, 1982

Let's start off with some basics. The following is a list of some HP-IL messages that all devices respond to. These are commands that are invisible to most IL owners; the IL Module's dedicated microprocessor invisibly generates and decodes them in response


IL YAMOTO'S ™
FIRST DAY


YAMOTO, TAKE THIS MESSAGE TO GENERAL # 27.
YAMOTO WAS VERY NERVOUS AS HIS FIRST DAY AS MESSAGE-BOY FOR THE PENTAGON BEGAN...
TAD 27


EVEN THOUGH THIS RECENT FOREIGN EXCHANGE STUDENT HAD DIFFICULTY READING ENGLISH, HE COULD ALWAYS RELY ON HIS JAPANESE SAVVY TO INSURE THAT THE MESSAGE GOT TO THE RIGHT PLACE.
WITH THE BILLIONS OF COPIES I'VE MADE, AT LEAST ONE WILL GET TO THE GENERAL!
TAD 27
TAD


AND UPON HIS RETURN TO HIS BOSS...
WHERE'S MY REPLY?
シコーンコイ見る?!
REPLY?!


(GULP!) I DON'T EVEN KNOW WHO THE ORIGINAL MESSAGE WAS MEANT FOR?

to some high-level user command. Don't worry if they look like Greek to you; some usage examples are coming up shortly.

DCL--Device Clear. All devices on the loop receiving this message will do something different. A printer might clear its buffer and eject the current page. A tape drive might just rewind its tape.

SDC--Selected Device Clear. Same as above, except instead of it being directed to every device on the loop, this one is executed only to devices that are currently "listening".

LAD xx--Listener Address. This tells the xxth device on the loop to become a "listener", which means the peripheral or controller is to grab all subsequent data bytes as they pass around the loop and treat them as input. "Listener" status stays in effect until a clearing instruction, such as the "unlisten" (UNL) command, is sent.

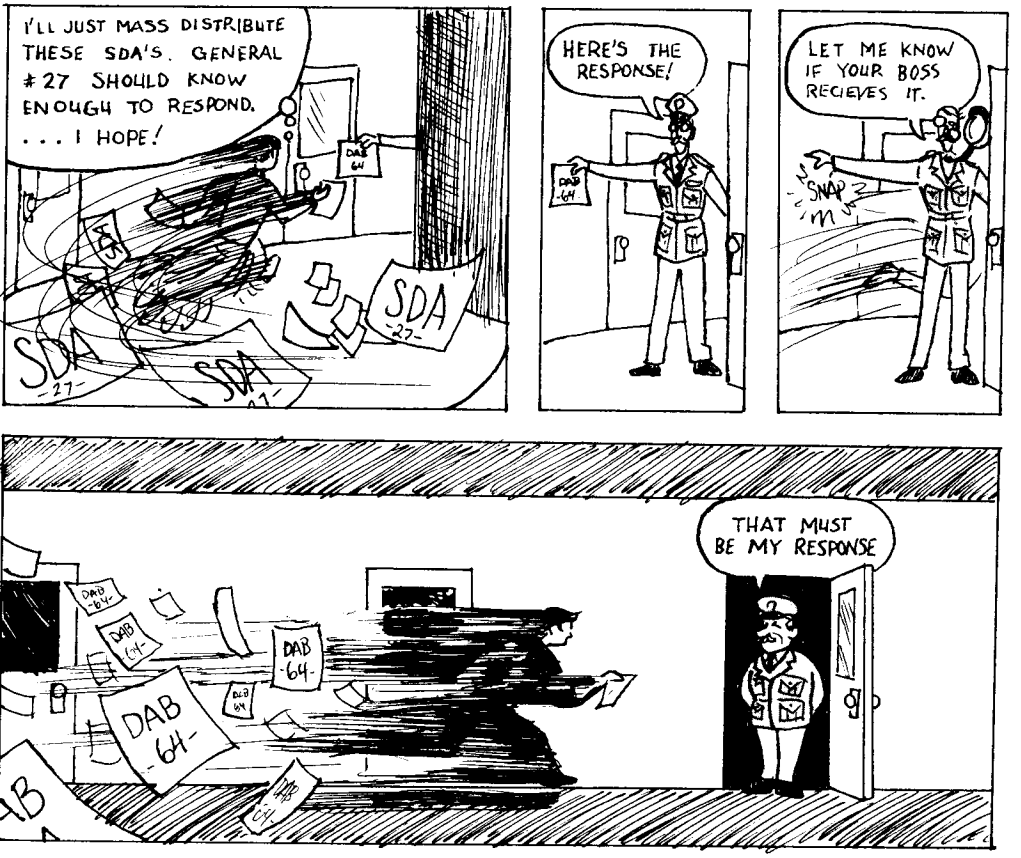
TAD xx-- Talker address. The xxth device gets ready to transmit data when receiving this command. It is up to the controller to set up listeners on the loop to receive the transmission. Although you can have as many listeners on the loop as you want, only one device can be assigned the role of "talker". UNT (Untalk) or assigning a different talker cancels this mode.

SDI--Send Device ID. This instructs the current talker to send its name. For example, the HP-IL/GPIO interface would respond by sending the character sequence "HP82165A", the non-descriptive name given to it by HP.

SAI--Send Accessory ID. The talker sends a number describing the type of device it is. Accessory ID's are discussed in detail shortly.

DDL--Device Dependent Listen. Tells the current listener(s) to interpret the following data as commands rather than input. Every device responds to DDL commands differently, depending on how it was programmed at the factory. You'll be hearing plenty about this later, believe you me!

DDT--Device Dependent Talker. Same concept as DDL, above, except this command addresses the current talker. Different devices will transmit different information depending on their device class and programming.

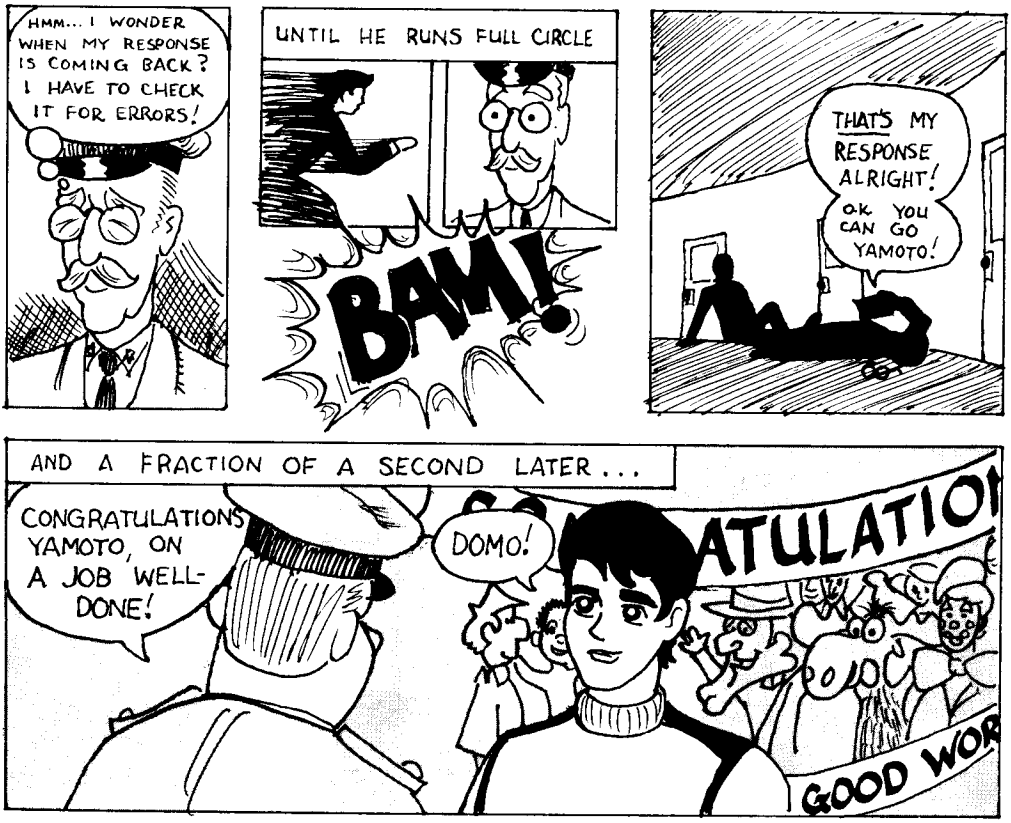

I'LL JUST MASS DISTRIBUTE THESE SDA'S. GENERAL #27 SHOULD KNOW ENOUGH TO RESPOND. . . . I HOPE!
SDA -27-
SDA -27-
SDA -27-
HERE'S THE RESPONSE!
LET ME KNOW IF YOUR BOSS RECIEVES IT.
SNAP
THAT MUST BE MY RESPONSE
DAB -64-
DAB -64-
DAB -64-
DAB -64-
DAB -64-

SST--Send Status. Tells the current talker to send at least 1 byte of status. Different devices, of course, need to report on different things, which is why every status word is interpreted differently, depending on the device that sent it.

AAD--Auto Address. This lets all devices on the loop know what position they're in. This command (also known as "configuring the loop") is performed every time the controlling calculator is turned on.

Here is a sample of how these commands are used in a typical loop session. Soon after the controlling device (such as a calculator) is turned on, an "auto-address" takes place, which works like this: The controller sends out the Auto address command. The first machine on the loop captures the message and says to itself "Hmmm... Nobody's modified this AAD message yet... I must be the first device on the loop. Until further notice, I'll call myself #1." The first device then modifies the AAD message to show that one device has responded to it so far and sends it onto the next device in the loop. The next device does precisely the same thing, calling itself #2 and passing the message along the loop. When the message finally makes its way back to the sender, the modified frame will contain the number of devices on the loop. This information is crucial since loop addressing is done by position number rather than by name. (NOTE: This is not always the case, but for this book, it is gospel.)

The user then asks the computer to print a file, to which the computer responds, "Golly, you told me to print something, and I'm not even sure there's a printer in the loop!" So then it goes about the job of asking each device, in order, what its function is. First, it makes the first device a talker, and then it issues the SAI (Send Accessory ID) command, to which the mystery device responds with a number, classifying it as a printer, mass medium, plotter, 8-bit port, etc. This process is repeated for every item on the loop until the first printer is found. When this happens, the computer stops and says, "Well, why didn't you say you wanted to send it to device #N in the first place?". It then makes that printer a listener, makes itself a talker, and then starts


HMM... I WONDER WHEN MY RESPONSE IS COMING BACK? I HAVE TO CHECK IT FOR ERRORS!
UNTIL HE RUNS FULL CIRCLE
BAM!
THAT'S MY RESPONSE ALRIGHT!
O.K. YOU CAN GO YAMOTO!
AND A FRACTION OF A SECOND LATER...
CONGRATULATIONS YAMOTO, ON A JOB WELL-DONE!
DOMO!
ATULATIO
GOOD WOR

sending data. After the last byte is transferred, the controlling computer de-assigns the talker (UNT) and listener (UNL) status' and returns to the tedious task of waiting for the user to do something.

That was a brief going-over. Now, on to some badly-needed explanations.

After each device has been assigned a number, it is sometimes useful to know the function of each device, so information destined for a printer can be routed there (as in the previous example). Most HP-IL devices have 2 ways of conveying this information: Device ID's (such as "HP82166", which is the name of the IL Converter), and Accessory ID's (such as 64, which classifies it as an interface type device). Having two methods at your disposal can be advantageous, as the following example will show:

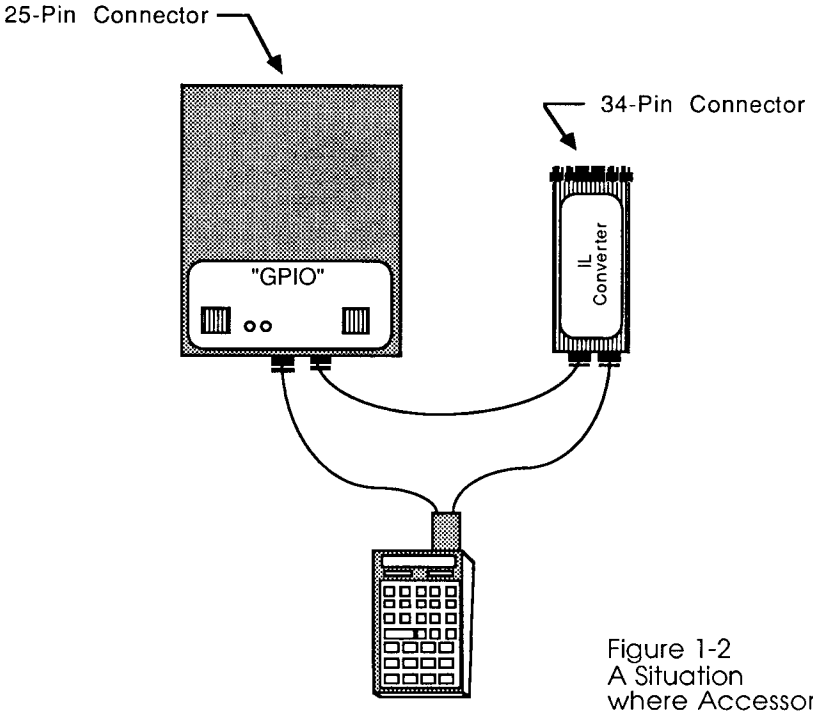


Figure 1-2
A Situation where Accessory IDs will not uniquely identify the device.

As will be fully explained in a couple of sections, the two devices on the loop in Fig. 1-2 are almost identical. Because they are slightly different, they have two different names and transmit two different ID's. Because they perform the same function, they will send the same accessory ID. We can take advantage of the sameness and the differences by tailoring their use to the situation.

With the 41 as a controller, as shown in Fig. 1-2, we wish to send information to the GPIO but not the IL Converter. Two steps are needed in order to talk to that device: first, find its loop position, and second, SELECT it. (SELECTing it on the 41 means that all future communications will go there by default.) Device ID's are used to insure talking to the proper device. Here is the code needed for the 41 to accomplish this:

01 MANIO	The printer is no longer the default device. (Not necessary in this case, but it's a sound programming technique that helps avoid confusion later.)
02 "HP82165"	Device ID for the 82165A GPIO 8-bit port, loaded into the ALPHA register.
03 FINDID	Puts the device's loop position into the X register.
04 SELECT	Makes it the primary device. (All subsequent data gets routed to there.)
05 "ABCDEFG"	A test string.
06 SF 17	Disables automatic CR/LF (Carriage Return/Line Feed) on output.
07 OUTA	Send the ALPHA register to the selected device.

or for the 71:

```
OUTPUT :HP82165; "ABCDEFG"
```

Sometimes situations come up (like in this book, for example) when you don't care whether an IL Converter or a GPIO will be connected to the loop, but you do want to the program to work without modification in either case. For this, we can use Accessory ID's, which were created specifically for this purpose. Accessory ID's just identify the type of device without worrying about specifics, such as whether or not the mass storage device is the cassette type or the floppy disk variety. The Accessory ID's fall into the following categories as defined by Hewlett Packard:

DEVICE CLASS	ACCESSORY ID RANGE
Controllers	0-15
Mass Storage Devices	16-31
Printers	32-47
Displays	48-63
Interfaces	64-79
Instruments	80-95
Graphics Devices	96-111
Undefined	112-223
EPROM Programmers	224

So, if your program wanted to find an interface device, it would (ideally) search for an Accessory ID anywhere between 64 and 79. It will be seen shortly, though, that in our case only the number 64 need be sought after.

Using Accessory ID's, the following program will work with either of the configurations in Fig. 1-3 (next page):

```
01 MANIO              Again, good programming
                      practice.
02 64                 Accessory ID for "interface
                      class" device.
03 FINDAID            Find Accessory ID, from the
                      Extended I/O ROM.
04 SELECT             Make it the primary device.
05 "ABCDEFG"
06 SF 17
07 OUTA
```

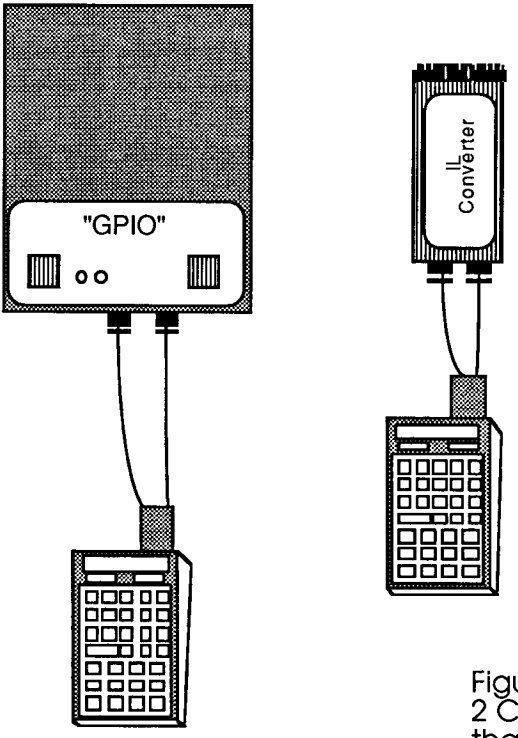


Figure 1-3
2 Configurations
that respond to
"64 FINDAID"

or for the 71:

```
PRINTER IS :INTRFCE          Treats the first
                             interface-class device
                             as a printer.
```

or

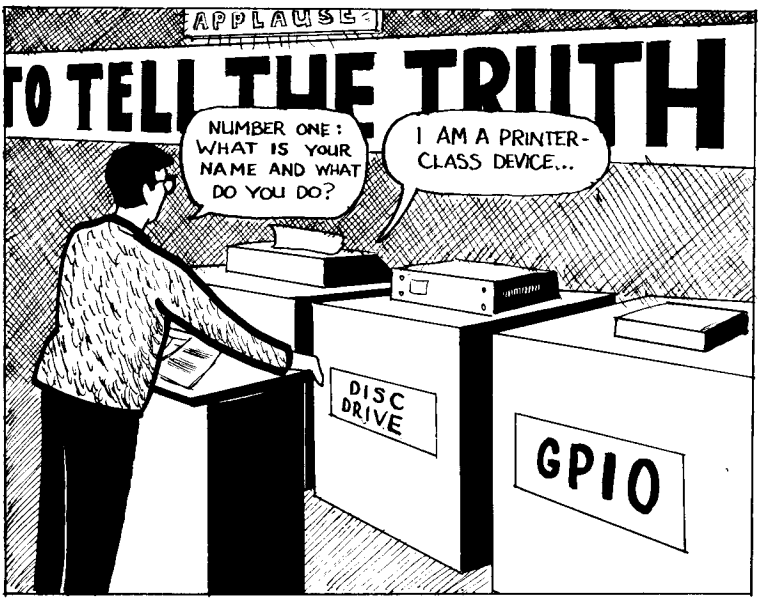
```
OUTPUT :INTRFCE; "BLAH, BLAH, BLA-BLAH!"
                              Sends string to first
                             interface-class device.
```

Another example: When the 71's "PRINTER IS :PRINTER" command is executed, it just looks for the first device on the loop whose Accessory ID (falling between 32 and 47) classifies it as a printer, and then routes all future PRINT functions to that device. It shouldn't care whether there's a thermal or a ThinkJet (tm) printer attached.

When information is to be transferred from one device to another, the controller must first assign the appropriate devices to be either talkers or listeners and then start the data going. Most of the time this activity is invisible to the user, but if we take the time to understand what's going on we can do some pretty impressive things. For example, let's look at the common case of the HP71 sending data to a printer, and then we'll perform some magic and have it drive 30 printers at once!

Here are the steps involved in "outputting" to a printer:

1) Identify the first printer on the loop using Accessory IDs. (Look for an Accessory ID of 32.)
2) Make the printer a Listener.
3) Make the 71 a Talker.



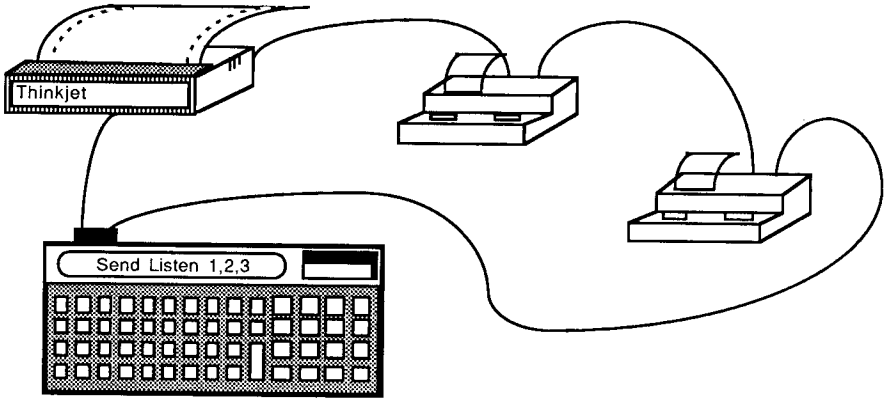


Figure 1-4
Multiple Listeners

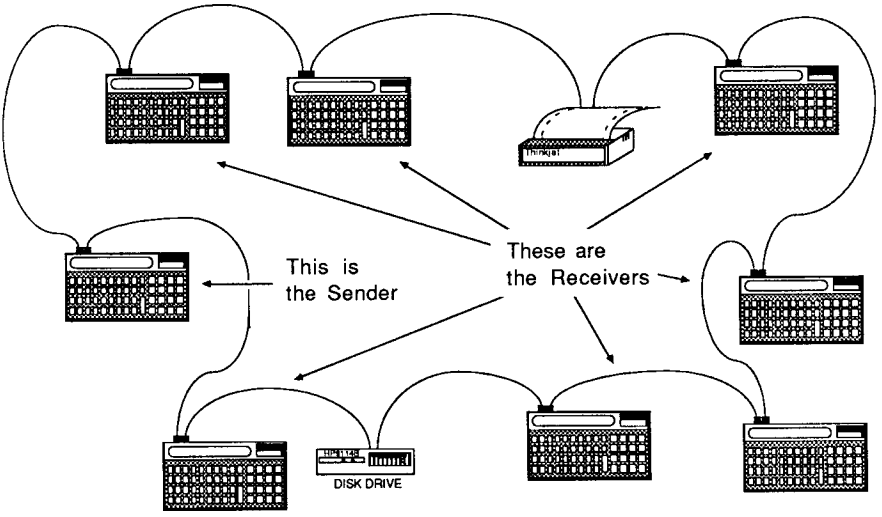
4) Send the file as a bunch of data bytes.
5) Remove the 71 from Talker status (the UNTalk command).
6) Remove the printer from listener status (the UNListen command).

All the above are automatically performed for you every time anything is printed using the standard commands. But if you wrote your own program, you could set up more than one listener on the loop and transmit the same information.

The following 71 program does just that.

```
05 RESET HPIL              ! Resets the machine to a known
                           ! condition.
10 PRINTER IS :LOOP        ! All future PRINT functions go
                           ! only to user-assigned
                           ! listeners.
20 SEND UNT UNL LISTEN 1,2,3 MTA ! Assigns the first 3
                           ! devices on the loop as
                           ! listeners and the 71 as a
                           ! talker (MTA)
30 PLIST ANYFILE           ! Print any old file.
40 SEND UNT UNL            ! Undoes Talker and Listener
                           ! status.
```

Here's another wonderful possibility: Take 20 71's and connect them all together in a loop. Throw in some other peripherals in random places on the loop if you desire. By turning only one of the 71s on and running the program below, you can have it turn all the other devices on, assign only the 71s as listeners, transfer a program to them, and shut everything off! Here's the program that does this:




```
10 INPUT "PROGRAM NAME? "; P$    ! P$ is the name of
                                 ! the program you want to send.
20 CONTROL ON @ OPTION BASE 1
30 DIM A(32)                     ! A is the array which will
                                 ! contain the loop position of
                                 ! all the 71s.
40 FOR N=1 TO 31
50 A(N)=DEVADDR("HP71("&STR$(N)&")")   ! Find the Nth
                                 ! 71 on the loop and store its
                                 ! loop position in A(N).
60 IF A(N)=-1 THEN 80            ! Branch here if there are no
                                 ! more 71s.
70 NEXT N
```

```
80 REMOTE :LOOP          ! Remote Enable all devices.
90 FOR X=1 TO N-1        ! For each 71 on the loop
100 SEND LISTEN A(X)     ! Make it a listener
110 NEXT X
120 SEND MTA             ! Make the 71 a talker.
130 OUTPUT :LOOP;"CONTROL OFF"      ! The first remote
                         ! command to be sent.
140 OUTPUT :LOOP; "BEEP @ COPY ";P$;":LOOP @ BYE"
                         ! Line 140 sends the remote
                         ! command to copy the specified
                         ! program from the loop.
150 COPY P$ TO :LOOP     ! Copies the program to all the
                         ! current listeners.
160 BYE                  ! Shut the sending machine off.
```

[Note for experts: Both loops in the above program could have been eliminated by sending the Auto Address Unconfigure command and doing a LAD 3 (the 71's default address). Keep in mind, though, this is supposed to be for educational purposes, and special cases like this only serve to confuse beginners.] [Note for beginners: See what I mean?]

41 vs. 71 Implementations

Well, that's a brief introduction. Throughout the examples you many have noticed that, even though the messages generated around the loop were the same, the steps required to generate them on each machine were quite different. The 41's instructions are much closer to what's actually sent around the loop; whereas the 71's instructions are of a very high level in order to shield the average user from unnecessary complexity.

Unlike the 41, however, the 71 has the capability to go from high-level, user-friendly mode to low-level, user-has-total-control mode. The 41 needs outside help to do this, and HP has provided it in the form of one of two plug-in ROMs: the IL Development ROM and the Extended I/O ROM.

The IL Development ROM (sometimes affectionately known as the DevIL ROM) is the more powerful of the two ROMs, and therefore a little more difficult to use. In addition to allowing you

to generate ANY type of HP-IL message, it also offers a SCOPE mode, which lets you put your 41 in the loop and will display all the low-level IL messages being passed around, giving the user a "window" to what's being sent from machine to machine. You also have the ability to store these frames away for future analysis, or capture them and output something completely different. For development purposes, this is invaluable. In addition, this ROM also provides very handy routines for base conversion, covering Hex (base 16), Octal (base 8), and Binary (base 2) words up to 32 bits wide.

Although not quite as powerful, the Extended I/O ROM is nevertheless the one that will be recommended for all but one of the 41-based projects described in future chapters. This is because in the never-ending tradeoff struggle between absolute control and user friendliness, this one wins out on friendliness (and in some cases speed). (Chapter 4 is the only example that requires a feature in the Development ROM.)

The Extended I/O ROM generally requires fewer commands than the DevIL ROM when doing loop configuration or sending out a finite number of characters, and provides a means of accepting a byte value of "0" as the first character in a string, something the 41 doesn't usually allow. It also uses the ALPHA register as a 24-character input/output buffer, whereas the DevIL ROM uses a separate memory buffer that must be read and written to with greater difficulty.

It is unfortunate that one must make a choice between the two ROMs. Even those of us who are foolish enough to buy both have difficulty using them simultaneously since some of their duplicate function names perform their tasks slightly differently.

Well, that should be enough information to get started. The next section gives the basics of the hardware aspects of interfacing. Then, Chapter 2 will offer some simple examples so we can put all this information to use immediately.

Theory of I/O: Triacs, Opto-Isolators, Gates

This section will help to answer the age-old question, "How do you actually get a computer to interact with the outside world?". It

certainly is not an obvious thing even to those who possess degrees in computer science, yet it is no less important a topic than Fast-Fourier Transforms or Runge-Kutta root finders. (For those of you who just said "Huh?": Don't worry; these are subjects best saved for another book.)

Surely everyone can associate with the analogy of a light switch: When someone or something throws the switch, two pieces of metal come into contact and complete a circuit. Relays were the offshoot of this; they added an electromagnet to perform the mechanical action of "throwing the switch". Relays, however, have several shortcomings: 1) They consume a lot of power (and therefore are dangerous to drive directly from a tiny computer circuit), 2) their contacts should be cleaned on a regular basis, and 3) they are slow. This last shortcoming will prove to be prohibitive when we discuss light dimmers in a later chapter.

The Opto Isolator

Consider a better alternative to a relay: the Optical Isolator. (Also called an Opto Coupler.) Comprised of an LED (Light Emitting Diode) and a photosensitive transistor, the Opto Isolator comes in an innocent 6-pin DIP package and is perfectly suited for turning small things in the outside world on and off by computer control.

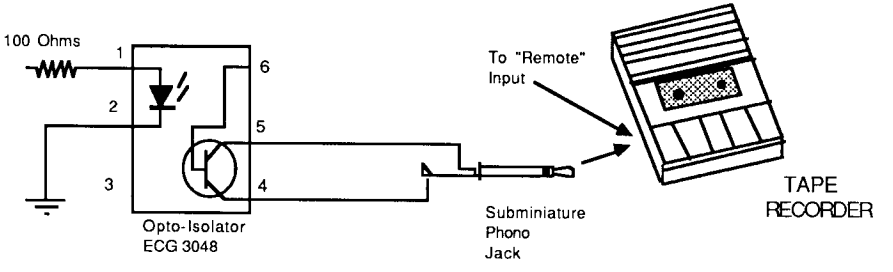


Figure 1-5
Turning on
a tape recorder

As an example, consider Fig. 1-5. Here, we have an Opto-Isolator connected to the REMote input of a tape recorder, an input originally designed for the on/off switch on many common microphones to turn the recorder off or on depending on how many mental blocks the dictator has. Here the Opto Isolator is behaving like the microphone's switch: it connects/disconnects the REMote input's contacts.

The LED, firmly encased inside the Opto Isolator, works like a light bulb: apply 2.5 to 4.5 volts to it and it will light up. In this case, when the LED (which you cannot see) lights up it makes the light-sensitive transistor conduct electricity, in effect acting like a switch and starting the tape recorder. (At this point, don't ask how to get the computer to make the proper voltage appear on the LED's input lead-- we'll cover that shortly.)

The Triac

Opto isolators are not the ultimate in interface tools. Despite their unparalleled ease of use, they can only handle small loads like tape recorders, cameras, buzzers, LEDs, and the like. This makes it a good time to introduce the TRIAC.

Triacs were specifically designed to allow computers to control alternating current devices consuming up to about 650 watts. Figure 1-6 (next page) shows how a triac can be hooked up to solve the infamous "how do you turn on a light bulb" problem. (I might mention that triacs are made for AC loads only; this circuit will not work if the bulb were hooked up to a 110V DC battery. (Actually it would; but when you remove the input signal from the Opto Isolator the bulb will not turn off.))

Triacs should only be used on resistive loads (such as light bulbs, clocks, and toasters) and not on reactive or inductive loads (such as blenders, vacuum cleaners, or generally anything with a motor).

As with all projects involving AC voltages, certain precautions must be taken to insure safety. Never touch the triac during operation; the large tab used for mounting the heat sink is electrically connected to pin 2, which means touching it is the

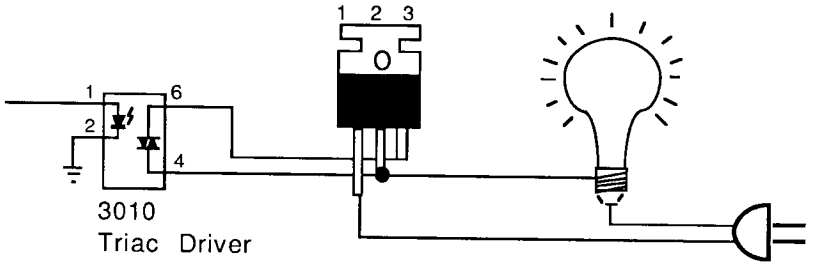


Figure 1-6
How to Drive
an AC Light
Bulb.

equivalent of sticking your finger into the wall outlet.

One must also be careful of excess heat dissipation from triacs when they are driving heavy loads. Unchecked heat buildup in these situations has been known to cause one or more parts to explode. Heat sinks (devices that help radiate the heat to the air) come in two varieties: the "clip-on" type for relatively light loads such as low-wattage light bulbs, and "monsters", which should be employed as shown in Fig. 1-7 (next page). Note the use of both a Mylar thermal insulator and silicon grease between the triac and the large heat sink! If not for the insulator, the heat sink will be attached to 110V and it's easier to accidentally electrocute oneself.

Real Heavy

For incredibly large applications, (like turning Las Vegas on and off by computer control), the only way to go is to have the Opto Isolator drive a relay (Fig. 1-8, next page). Relays are kind of like vacuum tubes, in that for the most demanding applications nothing else will suffice. For control applications, they should be driven with an opto-isolator, just like the triacs. Never ever connect a heavy duty relay without an opto isolator or transistor driving it!

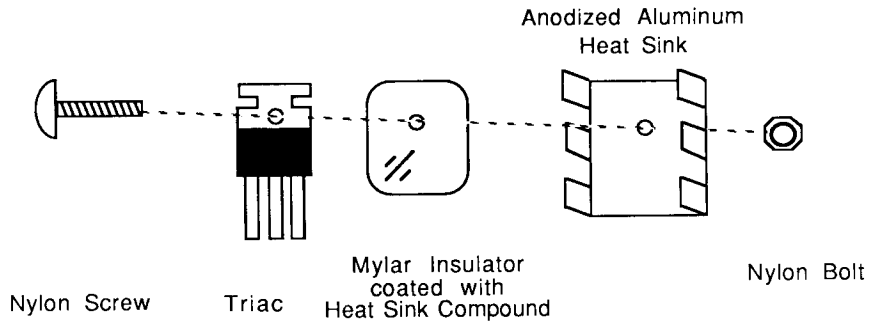


Figure 1-7
The proper way
to mount a Triac.

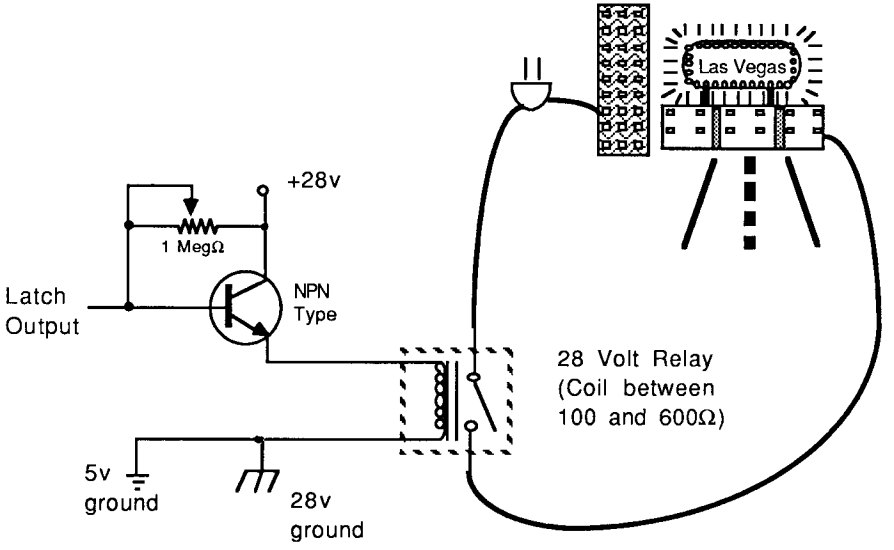


Figure 1-8
Controlling
Large Loads

To be sure, there are many other methods of computer I/O. The ones mentioned here are the ones that will be used from now on, so these simple interfaces will be recognizable in future chapters.

What's an 8-bit port?

You've seen larger computers with all those connectors in the back that allow it to communicate with the outside world. The most common type is a serial (RS-232) port, a 3-wire (most of the time) scheme which connects your machine to common peripherals such as modems, printers, etc. An HP-IL computer can be hooked up to these peripherals, too. Using a box called (appropriately enough) the HP-IL to RS-232 converter, you can attach more than 900 serial ports onto one machine! (Who says the small machines can't blow the big ones away?) RS-232 is covered more thoroughly in Chapter 11.

The other type of computer port common to most machines is called a parallel port, and gets its name from the way words are sent out. Many popular computer architectures represent data using 8-digit binary numbers, and therefore sending them out to the outside world is accomplished with 8 wires. (Refer to Fig. 1-9.)

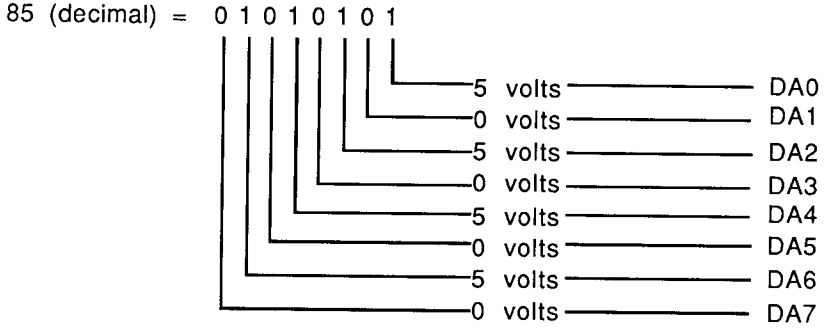


Figure 1-9
How One Byte Translates to Eight Wires.

There are three devices which attach to the loop and provide the function of an 8-bit port. All of them feature software selectable options, a 32 word data buffer, and handshaking lines to allow it to talk to almost any other machine. But before talking about their features and differences, let's go over some of the basics so the above sentence will seem less cryptic.

One of the most crucial aspects of transmitting data is making sure the destination machine is ready to accept it. For example, printers are notorious for printing information about a billion times slower than they receive it. How does a computer handle it? Well, in the olden days, it would just wait. Today, the printer might suck up the information as fast as the computer sends it and store it in its own memory, then print it at its own sweet pace. Either way, the machines transfer the information via an accepted social pattern.

Here is what a typical conversation between a computer and printer looks like while transferring 1 byte of data:

```
COMPUTER: Are you ready to receive some
          information?
PRINTER:  Yes, I am.
COMPUTER: Okay, here it is: "Blah, blah,
          bla-blah!" Did you get that?
PRINTER:  Yes, I did.
```

(This offers some insight as to why computers are so boring.)

There are three extra wires (added to the 8 we already have) whose sole purpose in life is to allow the above conversation to take place:

RDYI (Ready Data In): This line "goes high" (measures 5v) when the printer says "Yes, I am", and tells the 8-bit port that it is OK to send the next piece of data.

DAVO (DAta Valid Out): This line goes high each time the 8-bit port says "Okay, here it is:". It lets the printer know that whatever is appearing on the 8 data lines is valid, so the printer had better grab it now before it goes away.

DACI (Data ACcepted In): This wire goes high when the printer

tells the 8-bit port "Yes, I did". (Transmitting information to a computer can be kind of like writing a book: You never really know if your information was received properly. This extra wire fixes that situation; it reassures the computer that its information is being accepted and to please continue, for it really is interesting!)

These three wires are used only for data transfers from the 8-bit port to the printer. But some peripherals, such as telephone modems, must transfer information in both directions, making it necessary for the 8-bit port to possess three more lines of opposite function:

RDYO (ReaDY Out): Tells the modem that the 8-bit port is now ready to receive information.

DAVI (DAta Valid In): The modem sets this line high to tell the 8-bit port that the data it sees on the 8 wires is valid; better grab it now before it goes away.

DACO (Data ACcepted Out): Tells the modem that the last datum was received; thank you very much.

(Yes, there are reasons why the 8-bit port might not be ready to receive information all the time. These will be covered in a few pages.)

All three of the 8-bit ports share some impressive common attributes:

1) You can, by software commands, tell the port how many of its 16 wires to use for data transfer: You can have 8 wires that carry data in both directions; you can have 8 wires carrying data in each direction (16 total); or you can have all 16 wires carry a 16-bit word in both directions.

2) You can, by software commands, specify positive or negative logic. Positive logic means a wire measures 5 volts when the computer wants to say "Yes"; negative logic means the same wire measures 0 volts when the computer wants to say "Yes". The 8/16

data lines and the six handshake lines can be specified separately so you can have positive data logic and negative handshake logic if you wanted to. (This is, by the way, the default configuration.)

3) You can have it automatically affix a Carriage Return/Line feed (CR/LF) (or ANY OTHER character sequence) at the end of every line. You can also have CR/LFs automatically extracted from the input and replaced with the termination characters of your choice in case you're talking to some weird equipment that expects different line-termination etiquette.

4) You can specify how much handshaking is to occur. This ranges from full handshaking as just described, to valid/accept handshaking where you assume the other side is always ready to receive, to none at all (kind of like mailing a letter and not knowing if the person received it or even if he/she were still alive).

5) You can specify how long the data is to appear on the data lines, from 5 to 250 milliseconds.

6) The unit possesses a 32 register transfer buffer that temporarily holds only data in case one computer receives more slowly than the other transmits. It's not a lot, but for slow machines like the 41 it can make a big difference!
(WARNING! This is not a full duplex buffer, meaning it cannot accommodate both machines trying to send at the same time! The buffer will only hold onto information going in one direction, and information travelling the other way will be lost!)

7) MSRQ (Manual Service ReQuest) and GETO (Group Execute Trigger Out) wires that can provide two additional bits of I/0 (Input/Output) for unusual interface situations. Future chapters will explain how each computer interacts with these wires.

Most of these incredibly versatile features are accessible only via DDL and DDT (Device Dependent Listen / Device Dependent Talk) commands. Their use is not obvious and certainly not well documented in average HP literature, and an off-the-shelf 41 can't

even do it without one of the two afterthought ROMs described a couple of sections ago. So, an example of usage is in order.

Setting up the GPIO's internal control registers is easiest on the 71. Let's say we want to establish the following configuration: (See Fig. 1-10 for GPIO's control register map):

First word (R00): Respond to service requests when one of three conditions is met: Manual Service Request line is low, the buffer is full, or there's data ready for HP-IL.

To specify these conditions, the total of bits for the first word should equal: 64+8+2 = 74.

Second word (R01): Send an End-of-Data frame if the 32 register buffer is empty.

Total of bits for the second word: 16.

Third word (R02): Full handshake, positive data logic, negative handshake logic, 100 microsecond DAVO duration, 8-bit bidirectional, DAVO timeout disabled.

Total of bits for third word: 128+64+0+16+0+0+0+0 = 208.

R03-R18: Default values will suffice.

Here's a program for the 71 that will perform the above setup:

```
5  A = DEVADDR("%64")
10 SEND UNT UNL MTA LISTEN A DDL 0 DATA 74,16,208 UNT
   UNL
```

UNT and UNL just clear all devices from whatever status they might have been in prior to line 10. The next two words assign talker and listener status: MTA means My Talk Address, meaning the 71 is both a controller and a talker. LISTEN A assigns listener status to the device with an accessory ID = 64, as defined in line 5. (We also could have used its full name, which is "HP82166A", but for reasons described previously the Accessory ID is more useful.) DDL 0 is the first command sent after the talker

R00—Service Request Conditions (Default 01000000, Value = 64)

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
Status Service Requests 0 = Disable 1 = Enable	Manual Service Request 0 = Disable 1 = Enable	All Status Service Requests 0 = Disable 1 = Enable	Buffer Busy 0 = Disable 1 = Enable	Buffer Full 0 = Disable 1 = Enable	No GPIO Handshake 0 = Disable 1 = Enable	Data Ready For HP-IL 0 = Disable 1 = Enable	Ready For HP-IL Data 0 = Disable 1 = Enable
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

R01—Control and Status of Handshake (Default 00000000, Value = 0)

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
DACO and RDYO Control 0 = Disable 1 = Enable	Not Used	Set DACO 0 = False 1 = True	Buffer Empty End-of-Data 0 = Disable 1 = Enable	Set RDYO 0 = False 1 = True	DACI Status 0 = False 1 = True	Not Used	RDYI Status 0 = False 1 = True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

R02—Handshake and Data Formats (Default 11011000, Value = 216)

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
Handshake Options 00 = Strobed 01 = Ready/Valid 10 = Valid/Accepted 11 = Full		Data Logic 0 = Positive 1 = Negative	Handshake Logic 0 = Positive 1 = Negative	DAVO Time Unit 0 = 100 μs 1 = 5 μs	Data Format 0 = 8-bit 1 = 16-bit	Data Bus Setup 0=Bidirectional 1=Unidirectional	DAVO Timeout 0 = Disable 1 = Enable
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

R03—DAVO Pulse Width Number (Default 00000101, Value = 5)

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
Total value specifies number of DAVO time units added to basic 25-μs DAVO pulse width, except that a value of zero specifies 256 units. (DAVO pulse width is limited to 25 μs plus specified number of time units—40 μs minimum.)							
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Figure 1-10
GPIO Register
Control Map

and listener roles have been assigned. An 8-bit port interprets a DDL 0 as meaning "Don't send the following bytes to the outside world. Rather, interpret them as input to the control registers." Finally, UNT and UNL clean things up for the next data transmissions.

For the 41 to perform the identical task, either the HP-IL Development ROM or the Extended I/O ROM must be plugged into any port. Using the latter ROM as an example,

```
01 LBL "TEST"          12 64
02 CLA                 13 FINDAID
03 68                  14 SELECT
04 XTOAR               15 LAD
05 74                  16 0
06 XTOAR               17 DDL
07 16                  18 3
08 XTOAR               19 OUTAN
09 208                 20 UNL
10 XTOAR               21 ADRON
11 ADROFF              22 END
```

Lines 2 through 10 (about half the program) load the bytes destined for the control registers into the ALPHA register. Notice that there is an extra character (decimal 68), which is needed because the OUTAN command at line 19 insists on a leading dummy character in ALPHA. Line 11, ADROFF (Address Off), is a strange command needed to bypass some automatic feature in the Extended I/O ROM. 64, FINDAID, and SELECT set up as the primary device anything that has an accessory ID = 64 (which classifies is as an interface device). LAD makes it a listener; and 0, DDL prepares the 8-bit port for the forthcoming data. 3 OUTAN puts out 3 bytes of ALPHA to the listeners, and then UNL removes the listener status. Finally, ADRON (Address On) nullifies ADROFF (Address Off).

We could shorten the 41's program length and execution time by replacing lines 2-10 with only one synthetic text line: 244, 68, 74, 16, 208. All this helps to prove a simple point: Low-level

loop control is easier and faster on the 71.

Once one of the above programs have been run, the 8-bit port retains the parameters until the power is removed or some other reset occurs. Subsequent data transfers are then accomplished by SENDing individual strings, or pretending the device is actually a printer and sending it characters. For example, to output characters on the 41:

```
MANIO
[position of 8-bit port in loop] SELECT
[ALPHA] ABCDE [ALPHA]
ACA
```

and, of course, for the 71:

```
OUTPUT :GPIO; "ABCDE"
```

Now, on to the individual 8-bit-port descriptions.

The Individual 8-bit Port Descriptions

The first and the best is the HP82166A IL Converter, manufactured by HP. It featured taps from the HP-IL transformers so you could design your own power up/power down circuitry to turn on when a message passed through the loop or turn off when a special command was given. It possessed a chip select input (haven't quite figured out why I'd need it since handshake lines already exist), and ran forever on 4 AA batteries (and a 5v regulator, of course!). Notice the past tense in this paragraph; as this compact and power-mising unit is no longer available from Hewlett Packard.

All is not lost, however. All of the individual components that comprised the IL Converter EXCEPT THE CIRCUIT BOARD are available from speciality shops, so for the couple of projects that require the Converter's unique properties you can still wire one

together yourself.

(You see, the reason HP's IL Converter was so small is because they used a custom 4-layer printed circuit board, and when the initial production run was exhausted, HP decided to come out with the more expensive (but less versatile, see below) 82165A HP-IL/GPIO interface, and stopped making the 4-layer boards. Tooling costs for such boards are prohibitive for any volume less than 10,000, which is why no third party hasn't jumped in and designed a suitable replacement board.)

The third option, the HP82165A HP-IL/GPIO interface ("GPIO" is a suitable nickname), is still available from HP as of this writing and performs the same function as the IL Converter. HP has added a power supply (but still expects the other circuitry to have one of its own) and has taken away four interface lines. Other drawbacks are A) your battery-powered computer and your battery-powered circuitry must now be tied to an AC outlet if they are to talk to each other, B) power-conserving features (the ability of the circuitry to respond to the powerup/powerdown commands) cannot be implemented, and C) it costs roughly twice as much as it's predecessor.

I have found the GPIO extremely valuable for one task: prototyping. Its built-in power supply (you have to open the unit in order to "tap" into it) gives you one less thing to worry about when

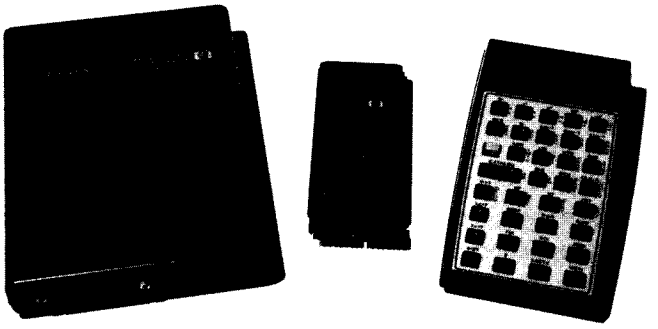


Photo #1 A visual comparison of the two 8-bit ports.

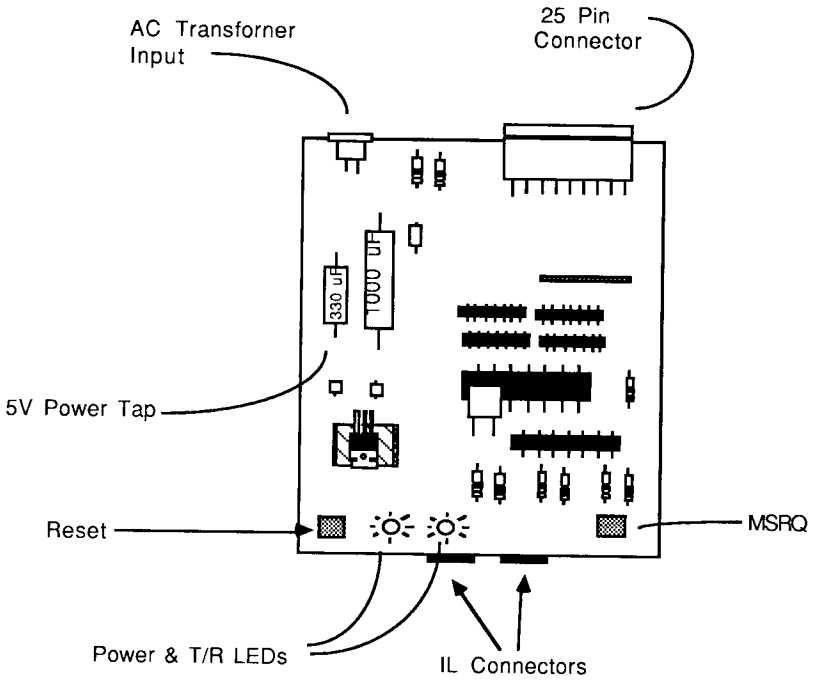


Figure 1-11
Power Tap inside the GPIO

throwing circuits together and trying them out. Its shape invites the installation of a proto-board on top of it. Its POWER and T/R (Transmit/Receive) LEDs offer just a little visual confirmation of the transference of data.

To tap into the GPIO's power supply, it is necessary to void the warranty and open the case. (Unlike HP's calculators, this is an easy thing to do.) This is done by removing the two Phillips-head screws on the case's bottom. Then just solder a wire onto the 330 microfarad capacitor as shown in Fig. 1-11 This becomes the +5v wire to power your outside circuitry; the GND (0v) wire is already available at pin 21 of the 25-pin connector.

I should emphasize that, despite the stated differences and drawbacks, any of the 8-bit ports described above can be used to construct all the forthcoming projects (Well, almost all; the

telephone answering machine in Chapter 7 requires the extra PWRDN interface line).

Now that the differences have been all spelled out, I will hence forth refer to any of the generic 8-bit ports mentioned thus far simply as "GPIO". This should make life easier for both of us.

This page intentionally left blank.

(Yet another in a series of self-referential jokes.)

Chapter Two

MORE SIMPLE EXAMPLES

"Nothing is so simple that somebody doesn't know it".
--George Friedman

Let's finally get to some immediate gratification. The circuit shown on the next page is the hardware needed to have the computer visually count in binary. It is a simple experiment, yet reveals a great deal about how the GPIO (meaning ANY of the 3 8-bit ports described in the last chapter) works.

Figure 2-1 shows the circuitry. In addition to the 8 LEDs (Light Emitting Diodes) needed for the counting, only 2 other components are necessary: a latch, and 1/6 of a 4069 hex inverter.

The latch is a device that holds onto the data it sees until the next byte comes along. Compare this with the output of a GPIO, whose data appears on the data lines for a mere 65 ms. We know when the data appears because the DAVO line pulses low at that instant.

The hex inverter is shown symbolically as a triangle with a circle on its output. When you consider that computers represent signals as either being 0 or 1, and that a "0" is less than 2v and a "1" is about 5v, an inverter simply outputs the opposite of what it was fed. One of the most common uses for inverters is as "glue" to make different components work together. In this example, it is making a negative signal coming from the GPIO suitable for the latch, which expects a positive signal.

The particular latch mentioned, the 74C373, has the 8 LEDs of Fig. 2-1 connected directly to its output. *This Method of driving*

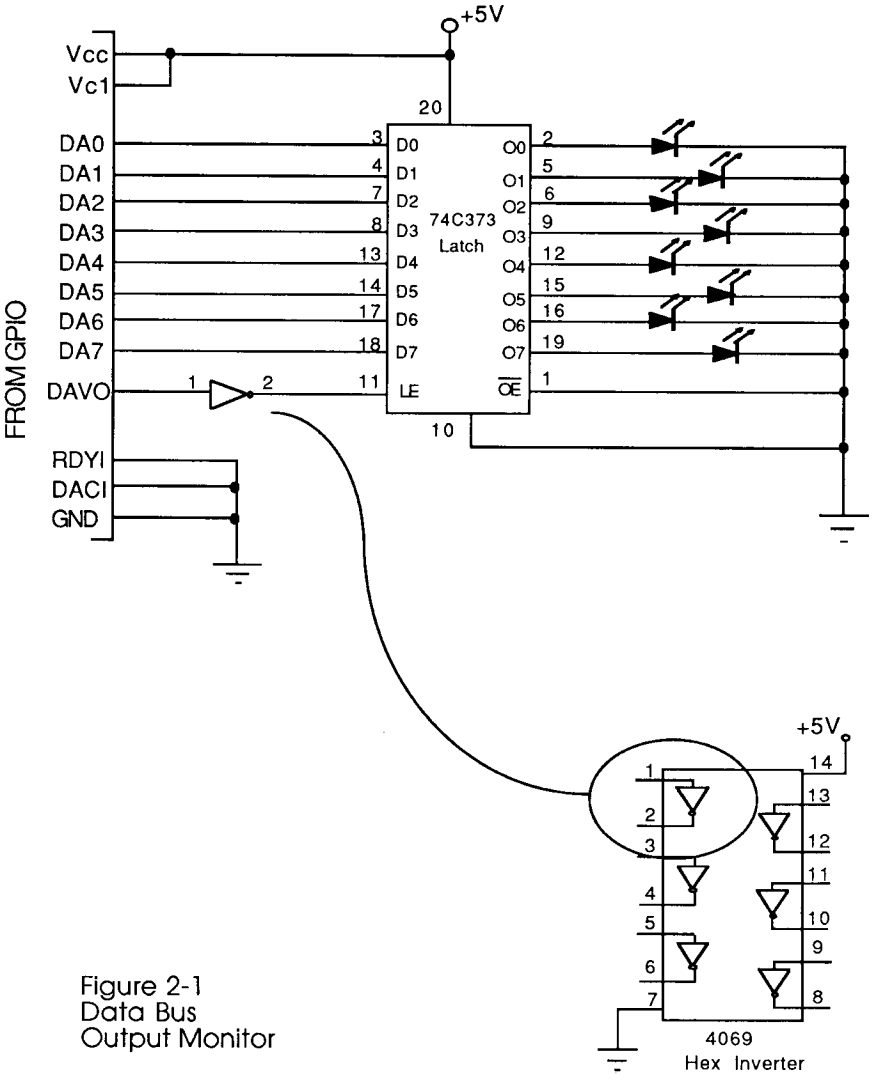


Figure 2-1
Data Bus
Output Monitor

external components (including Opto-Isolators) should not be used with anything but CMOS integrated circuits!! In addition to their many other attributes, CMOS circuits have the ability to provide up to 4 mA of current to outside devices that need them, like LEDs.

Other popular logic families such as TTL or LS can't supply that much current; and placing such a demand on them could damage the IC! CMOS-type circuits can be identified by a "C" in their part number, or a part number of the form 4XXX.

Building the Circuit

If this is your first encounter with either the GPIO or electronic circuits in general, I offer a few tips. Often the most difficult aspect of assembling a circuit is trying to isolate what went wrong when it doesn't work. (And it never works on the first try!) Generally, all the hardware can be broken down into 2 commonly troublesome categories: the GPIO's wiring, and the hand-assembled circuitry.

The GPIO, and specifically the IL Converter, make many connections to the outside world via the devices' 25-pin (82165A GPIO) or 34-pin (82166A IL Converter) connectors. One miswired handshake line or one forgotten ground will make the whole system inoperable, and generate all sorts of errors on the controlling computer.

The most common error occurs while trying to hook up the IL Converter, as its dense 34-pin connector is both hard to count and its first pin is not clearly labeled. Even worse, most people use the standard 34-pin ribbon cables (used for common computer applications) to interface the converter. This often requires cutting the cable in half, separating and stripping each conductor, and carefully counting from the end every wire that is to be connected. This situation, although unavoidable, invites all sorts of wiring errors.

Therefore, when constructing a project, an important intermediate step is to power-up only the 8-bit port and see if it responds to commands like STATUS or INSTAT. If you get a LOOP BROKEN error message, remove all power immediately and check for miswiring. This simple step is guaranteed to save you much frustration during construction and troubleshooting of projects.

Also recommended for troubleshooting is something called a Digital Logic Probe, which is used to visually show whether an IC

pin is at a "1" or a "0" state. If you're on a budget, an LED connected between the pin in question and Ground will show the same thing; but the logic probe is designed to interfere with the circuit as little as possible so as to not influence its behavior.

After the circuit in Figure 2-1 is hooked up and working, run the driver program on the 41 (In this example, make sure the GPIO is the only device on the loop):

```
01 LBL "COUNT"
02 MANIO
03 SF 17
04 SF 21
05 .127
06 STO 01
07 LBL 01
08 RCL 01
09 ACCHR
10 ISG 01
11 GTO 01
12 GTO "COUNT"
```

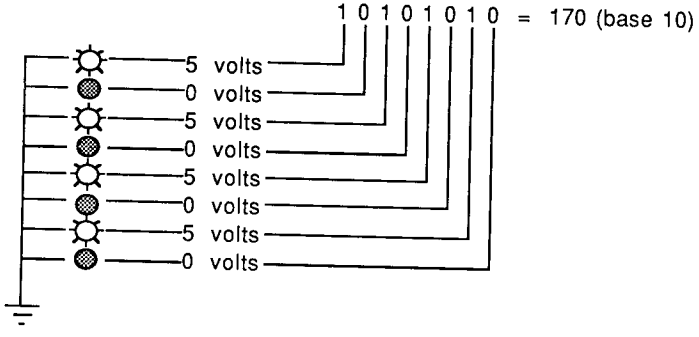
Well, the program certainly looked simple enough! All it is supposed to do is count from 0 to 127 in binary, and send each number to the loop via the ACCHR (ACcumulate CHaRacter) command. However, that didn't quite happen. Notice that the numbers 10, 13, and 126 appeared on the LEDs as 0, 253, and 28, respectively. Also notice that we couldn't have counted up to 255 even if we wanted to. (We do have 8 LEDs, after all!) Why?

This traces back to what the IL Module was originally designed to do: communicate in ASCII characters. In the ASCII definition, character codes 10 and 13 represent Carriage Return and Line Feed. ACCHR's function is to accumulate characters, so it decides to change codes 10 and 13 to something else. The reason 126 doesn't translate correctly is a little less obvious: 126 on the 41 defines the "Sigma" character, which is represented on the 82162 printer as 28. The 41 is simply translating one non-standard character so it will show up on the printer correctly.

The way to correctly count to 255 is to change line 5 to .255 and replace line 9 (ACCHR) with three lines: CLA, XTOAR, and OUTA. XTOAR (X to A Right) is the same as XTOA in the Extended Functions Module: it takes the decimal in X and stores it as a character in ALPHA. Throughout this book, these two

commands are interchangeable.

From watching the unit count, one can conclude that it is possible to turn on any desired LED or combination of LEDs just by sending the proper number via ACCHR. For example, to turn the first one on, just press 1 ACCHR. To turn on every other LED (positions 2, 4, 6, and 8), just press 170 ACCHR. (Where did the 170 came from? See below.)



Some interesting things to note about flags 17 and 21. OUTA sends out the entire contents of the alpha register and terminates it with a CR/LF if flag 17 is clear. If you don't believe me, try clearing flag 17 and run the program from line 4. Three characters are now sent in succession instead of one, and the last one (line feed, =1010 in binary) remains showing.

Two things can be concluded here: 1) For control applications, a CR/LF is most undesirable; therefore always set flag 17 or use ACA instead of OUTA. 2) If a control program contains an AVIEW (mine always do), clear flag 21 before each AVIEW, otherwise your message will be "outputted" onto the loop. (There I go, verbing nouns again!) Be sure to immediately SF 21 after an AVIEW, otherwise future GPIO commands will be ignored.

A NOTE TO 71 OWNERS: You should be glad to know that this short program does the same thing as the 41 version above, except it doesn't require 5 paragraphs of discussion:

```
 5 ENDLINE ""
10 FOR X=1 TO 255
```

```
20 OUTPUT ":GPIO"; CHR$(X)
30 NEXT X
40 GOTO 10
```

The ENDLINE "" of line 5 does the same thing as setting flag 17 on the 41: it suppressed the automatic CR/LF on output.

There are some other hardware aspects of this experiment that are worth mentioning. Notice that both RDYI (Ready Data In) and DACI (Data Accepted In) lines are always held low. This is an easy way of specifying no handshaking: always fool the handshake lines into thinking you're always ready to receive and that you're always acknowledging the data. (Incidentally, the way you're supposed to do this is by sending the following sequence to the GPIO:

```
10 A=DEVADDR("%64")
20 SEND UNT UNL MTA LISTEN A DDL 0 DATA 64,0,24 UNT
   UNL
```

This sends out a DDL 0 configuration command and tells the GPIO not to look for handshake. Here's one case where the hardware solution is infinitely easier than the software version). Line 10 of the above 71 program, incidentally, looks for a device of interface class (accessory ID=64). This is done because the IL Converter and the GPIO have different loop names ("HP82166A" vs "HP82165A"), and looking for the accessory ID rather than the standard ID will find either one. (You probably already knew that after reading the previous chapter, though.)

Turning an AC Device On and Off

In the previous chapter we discussed how to make an AC light turn on and off, but never went into the details of how to do it. Here, we remedy that situation.

At this point, after having the 8 LEDs turn on and off in a binary

fashion, adapting the circuit to control lamps instead of LEDs is a straightforward task. Fig. 2-2 shows the same circuit used for binary counting, except the LEDs are gone. In their place is a single 110V AC lamp, being controlled by circuitry introduced earlier in this chapter.

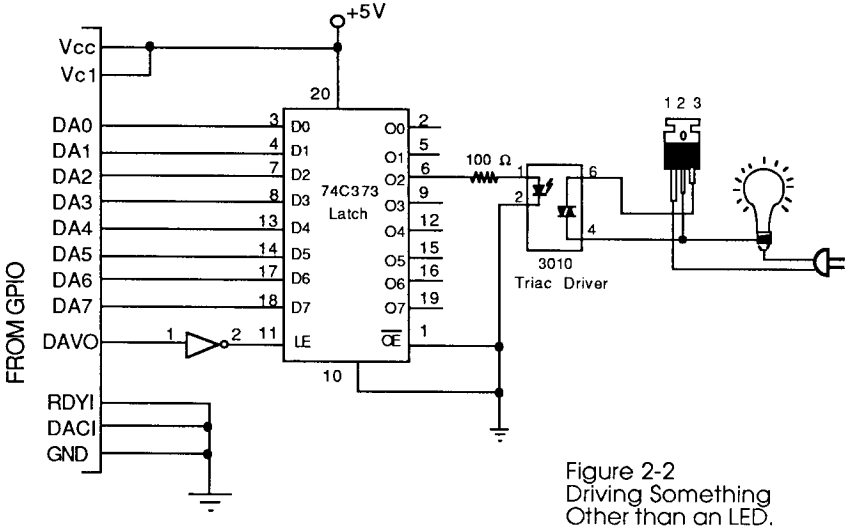


Figure 2-2
Driving Something
Other than an LED.

There just as easily could have been 8 lamps in figure 2-2; which should start to give you an idea of the IL Converter's power. Generally speaking, if you can turn an LED on and off, you can turn anything on and off!

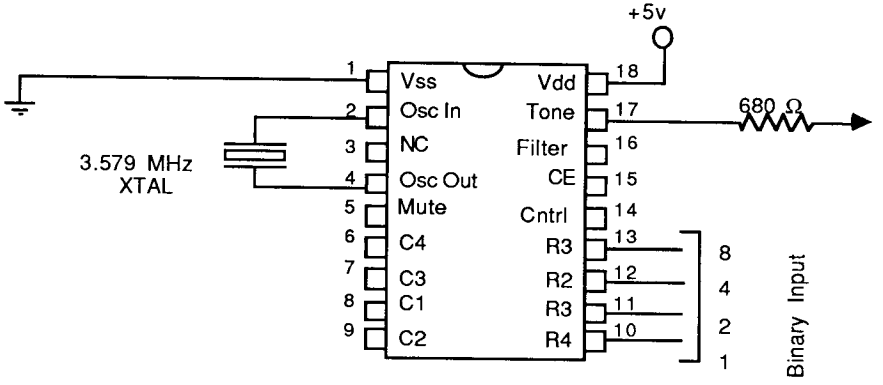
This is a good time to bring up an impressive attribute of controlling the world via HP-IL. With the above method, one GPIO type interface can control up to 16 devices (if it has been properly configured to be 16 bits wide with the DDL command). Using primary and secondary addressing, a 41 or a 71 can address up to 961 such interfaces, making a grand total of 15,376 light bulbs (or anything else) that can be controlled by one computer. Not impressed yet? Multiply that result by three, since the 71's operating system can support three separate IL loops simultaneously. We now have one hand-holdable computer that can have direct control over 46,128 devices!! Try that with an IBM PC!!

Touch Toning®

Just to show that interfacing can do more than just turn things on and off, here's a simple circuit that will allow you to turn the 41 into an automatic pushbutton telephone dialer. This project came in particularly handy back in the days when accessing an alternative long distance phone service was a painfully long 22-digit (plus waiting) process, and getting a subsequent busy signal would only help to raise one's blood pressure.

"Touch Tone" refers to a method of telephone dialing which uses two simultaneous, non-interfering sine waves rather than a pulse train to signal the central office of the desired digits. Their advantages are many: in addition to the increased speed in placing a call, it also provides an inexpensive and accurate method of data transmission over narrow-bandwidth telephone lines.

First, let me introduce the chip. I am using a National Semiconductor MM5395 Touch Tone (that's a registered trade mark of AT+T, you know!) generator which was designed to replace the many components in a standard pushbutton phone's keypad with only 3 components: the chip, a 3.579 MHz quartz crystal, and a resistor.



MM5395 Touch Tone® Generator

The chip's pinout is shown above. It was designed to accept input from either a standard "2-of-8" keypad (each keystroke closes

2 of a possible 8 contacts), or from a computer, depending on the state of the "XMIT" input. Normally, a controlling computer "dials a digit" by placing a binary representation of the digit onto pins R1-R4 and setting the chip enable (CE) line high (= 1). After waiting a short interval, the computer then brings the CE line low (= 0) to instill a brief period of silence the phone company expects to hear. It then sends the next digit to the chip and starts the process all over again.

How does one regulate the speed that the number is dialed? We theoretically could tell the computer to not transmit so quickly, but as previous experiments showed, the 41 has two transmitting speeds: Fast (in 24 character bursts with the 41), and slow, with no chance of intermediate speeds. The solution employed is this: use the fast mode, but have the circuitry behave like a printer and say "you can transmit your information when I'm good and ready!". How does a printer normally say this? Why, with handshake lines, of course!

The top half of Fig. 2-3 is pretty straightforward; we are simply replacing the LEDs with a National Semiconductor MM5395 Touch Tone generator chip. The bottom half is a little more interesting; it blindly says "Yes, I'm ready" only at predetermined time intervals, which holds up the output data. This essentially is clocked output.

Notice that, in the figure, the RDYI (ReaDY data In) and DACI (Data ACcepted In) lines are connected to the complementary outputs of a flip-flop, which in turn gets its pulses from a 555 pulse generator. The flip-flop's output will alternately hold one of the lines true, and will change states as soon as a pulse comes in from the 555. The GPIO will hold its output until the RDYI line goes true. When it does, it transmits a word which the latch holds onto and feeds to the Touch Tone chip, and then waits for the next pulse that makes the DACI line go true. With this method you can make the Touch Tones go at any speed you wish by rotating the 1 Megohm potentiometer.

The following program shows one way to drive this circuitry:

```
01*LBL "TTONE"     04 CF 21          07 SF 21
02*LBL 05          05 "   READY"     08*LBL 00
03 MANIO           06 AVIEW          09 GETKEY
```

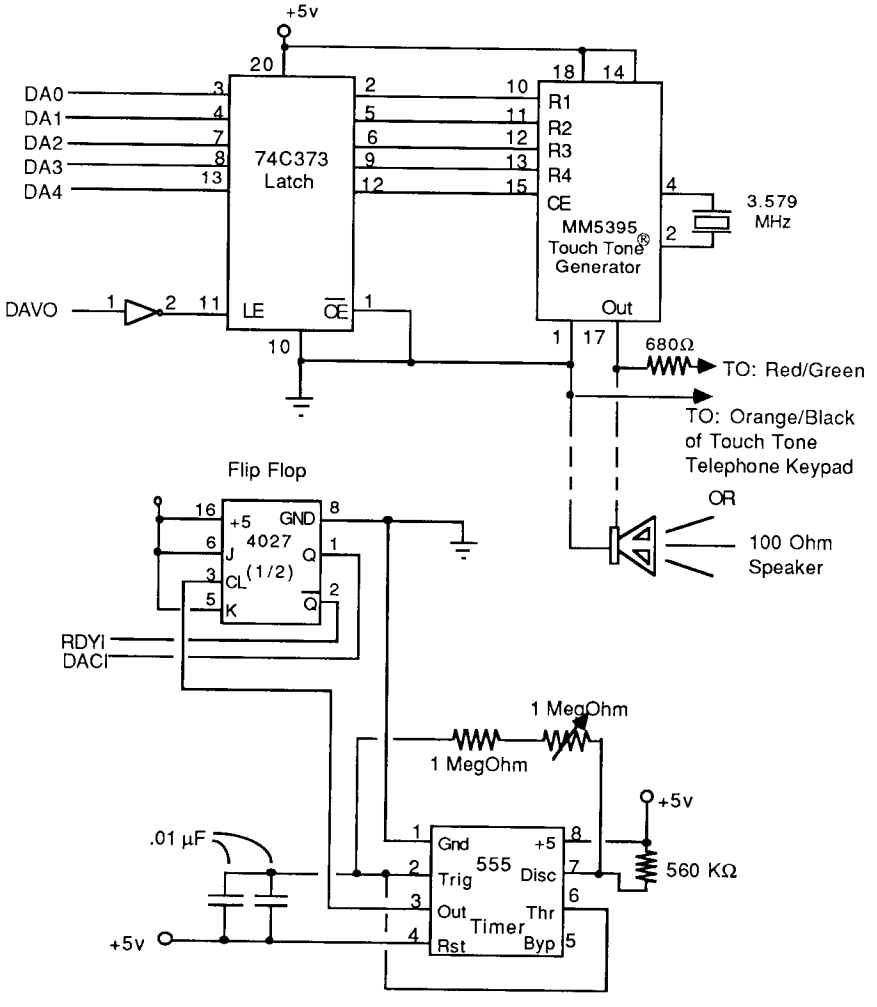


Figure 2-3
Touch Tone
Hardware

```
10 GTO IND X
11*LBL 11
12 " 8 5 3 1 2 1 2"
13 PRA
14 GTO 05
15*LBL 12
16 " 1 7 1 4 "
17 ACA
18 " 5 4 9 7 6 7 4 "
19 PRA
20 GTO 05
21*LBL 14
22 " 9 8 9 0 9 2 8 "
23 PRA
24 CF 21
25 "  WAITING"
26 AVIEW
27 SF 21
28 .08
29 STO 06
30*LBL 06
31 ISG 06
32 GTO 06
33 " 9 9 9 9 9 "
34 ACA
35 " 3 1 9 "
36 ACA
37 " 5 5 5 2 3 1 0 "
38 PRA
39 GTO 05
40*LBL 84
41 CLD
42 END
```

(Barcode for this program, as well as all other 41 programs, is provided in Appendix A pg. 295.) The program works like this: After XEQ'ing TTONE the 41 displays "READY" and the GETKEY function waits for a key to be pressed. The above program only has numbers to dial for the A, B, and D keys, but can easily be expanded (you can substitute your own numbers, too!). If the A key is pressed, we branch to LBL 11 (the keycode returned by GETKEY) and it promptly sends out the string to dial "Time". Notice the spaces between the digits; they are there to serve as inter-digit silence so the phone company will be happy. LBL 14 is a little elaborate; it was designed to access one of those alternate long-distance phone service back in the days when it was a pain. It first dials a local number, waits for the local computer to answer (that's the loop between 28 and 32), dials a 5-digit access code, and finally the desired area code and number.

ASCII-encoded numbers are handled using 6 binary digits: bits 0-3 for the BCD digit, and bits 4 and 5 which are always on. I used the first four bits and fed them directly to the touch tone chip, and used the remaining two bits as "chip enable" signals; one going to the touch tone chip, the other disabling phone line isolation. To minimize the number of external parts, the circuit connects to the

speaker may be attached instead of the resistor and the touch tone frequencies fed audibly through the phone's mouthpiece. Both should work equally well.

The stability, immunity to noise, and common availability of Touch Tone signals makes them easy to use for things other than dialing. More versatile uses for them are covered in Chapters 6, 7, and 10.

Well, these have been some examples. Despite their simplicity, these few techniques will be used and built upon in vastly different ways throughout the remaining chapters. Armed with the knowledge presented here, you will be able to take the ideas presented and modify them to fit your own needs, or come up with completely new applications!

Chapter Three

INEXPENSIVE I/O USING THE TIME MODULE

"Never trust a computer bigger than you can lift."
--Anonymous

This method of I/O (which stands for Input/Output) doesn't use HP-IL. In fact, it's tough to justify including it in this book because it only works with the 41, is very limited, and requires some modification (maybe). (As it turns out, we're in for more of the same later, so we may as well include this, too!)

It's primary advantage is it's the cheapest method for turning one device on and off. No HP-IL module or IL Converter (with its mandatory extra power source) is needed. Furthermore, both 41Cs and 41CVs can use it without modification. (CX owners: read on!)

The Time Module has to be the best-implemented afterthought that HP has produced for the 41. Yet upon reading some of their internal documentation, one discovers that the Phineas chip was designed for even more versatile use. (Phineas is its code name; every project has one. Its real name is 1LF6, but that's too difficult to pronounce.) Consider these extras:

1) The Phineas I.C. has inputs to start and stop the stopwatch by hardware control.

2) Similar inputs exist to start/stop the clock. (Those of you who enjoy power trips will appreciate this new ability to stop time.)

3) Two outputs are available to mark the occurrence of an alarm or a stopwatch zero-crossing. This way, when either of the 2 events occur, you can trigger a hardware function rather than or in addition to waking up the calculator and running a program.

All these extra inputs and outputs are accessible to users who

are "in the know"; all you have to do is crack open the time module and there they are! Figure 3-1 contains pictures of what the time module looks like when you crack it open. When you flip it over, (2nd photo), you will see 18 unidentified solder pads; 6 of which have been identified in Fig. 3-2. The pads function as follows:

#1 Stop Stopwatch
#2 Start Stopwatch

These two pads allow you to hook up the stopwatch to the outside world and measure real time events to 1/100th of a second. If the two pads are connected together, each positive pulse will then toggle the stopwatch on or off. The only curious behavior these pads exhibit is the last digit (100ths position) of the stopwatch display does not update, although internally it is stored correctly. Hitting any key will restore the last digit to the correct value.

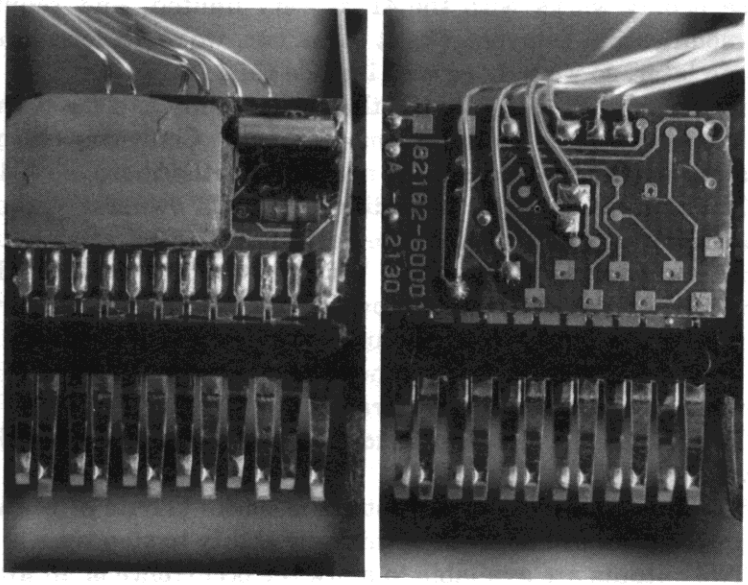


Figure 3-1: Photos of Phineas, the nickname for the time module, when cracked open. The left photo (3-1a) shows the front view; the rear view on the right (3-1b) reveals previously unknown solder pads which can be used for interacting with the outside world.

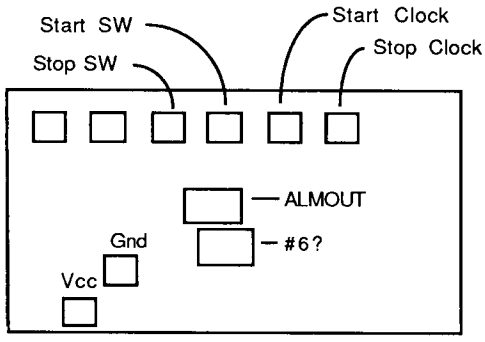


Figure 3-2
Phineas' Rear View

#3 Start Clock
#4 Stop Clock

Although less useful than the above, these two pads enable you to start or stop the clock advance. The clock functions still operate; i.e. the quartz crystal still vibrates, the stopwatch still runs, etc., it's just that the clock register never gets incremented. If these two pads are tied together, a single pulse will toggle the clock on and off. Toggle mode will also turn the 41 on, but since user flag 11 is ignored this is not a useful form of input. Another problem with stopping time while in clock mode is that sometimes the ALMOUT output pad will "go crazy" and will randomly turn on and off several times a second. Undesirable behavior.

Both the clock and stopwatch test pads are internally tied low, debounced (to a minimal degree), and require a minimum pulse width of 10ms to activate.

#5 ALMOUTA (Alarm "A" Out)
[#6 ALMOUTB (Alarm "B" Out)]

Pad #5 is the new means of control mentioned earlier. Every time an alarm becomes active, the time module will 1) pulse the ALMOUTA pad (once for a normal alarm or twice if it's a repeating alarm OR if there are any other alarms in the

ALMCAT), and then 2) will turn the 41C on and service the alarm. This test pad, therefore, becomes the only means for output.

The pad below it, ALMOUTB, is supposed to behave similarly whenever the stopwatch is counting backwards and crosses zero (a "TIMER ALARM"), but it seems this never got implemented, as pad #6 doesn't do anything.

Transforming the ALMOUTA pad's pulses into useful output is a task that is only slightly more involved than the techniques discussed in Chapter 1. This time, instead of using an 8-bit latch to hold onto the momentary signal, we will use something called a JK Flip-Flop, which will be wired to act like a divide-by-two device.

Such devices' functions are almost self-explanatory. The output is exactly half the input: two pulses in, one pulse out. One pulse in, half a pulse (the output stays either low or high) out. This is precisely what we need to drive the standard light bulb configuration in Fig. 3-3.

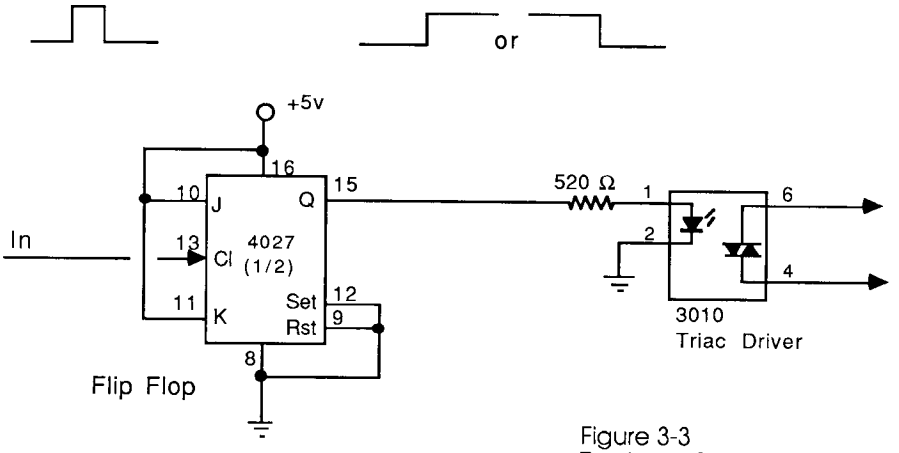


Figure 3-3
Turning a Single Pulse into an On/Off State.

In the simplest case, where you have a non-repeating alarm and an empty alarm catalog, a single pulse appears on the ALMOUTA pad when an alarm activates. A JK Flip-Flop wired as in Fig. 3-3 will take the pulse and change the state of the output


HEY, ROCKY! WATCH ME PULL AN INEXPENSIVE MEANS OF I/O OUT OF A HAT!
AGAIN?
NOTHIN' UP MY SLEEVES-
... PRESTO!
BUT BULLWINKLE, THAT'S A TIME MODULE! TIME MODULES CAN'T INTERACT WITH THE OUTSIDE WORLD!
SURE THEY CAN! BECAUSE INSIDE EVERY ONE ARE THESE LITTLE UNDOCUMENTED SOLDERING PADS!
"UNDOCUMENTED" SOLDERING
ATTACHING WIRES TO THEM WILL ALLOW OUTSIDE EVENTS TO CONTROL THE CLOCK OR THE STOPWATCH, SO WE CAN ACTUALLY MAKE USE OF IT'S 100TH OF A SECOND ACCURACY!
WHICH MEANS WE CAN ACCURATELY MEASURE THIS SPRINTER'S TIME AT THIS RACETRACK.
FINISH LINE
START STOPWATCH
ELECTRIC EYE
SOUND SENSOR
STOP STOPWATCH


OR HAVE AN ACTIVATING ALARM TRIGGER AN OUTSIDE EVENT.
BUT MOST IMPORTANTLY YOU CAN MAKE TIME MEASUREMENTS AND RUN PROGRAMS SIMULTANEOUSLY!
GOLLY, BULLWINKLE, THAT SOUNDS GREAT! BUT HOW CAN 41-CX OWNERS ACCESS THIS VERSATILE CAPABILITY?
THAT'S EASY ROCKY... PRESTO!
GRRR!
LOOKS LIKE I GOTTA GET ME A NEW HAT!
AND NOW HERE'S SOMETHING WE HOPE YOU'LL REALLY LIKE!

labelled "Q". Since a logic 1 is electrically equivalent to +5v, this output can be applied directly to the Opto-isolator to switch anything on and off every time ANY alarm activates.

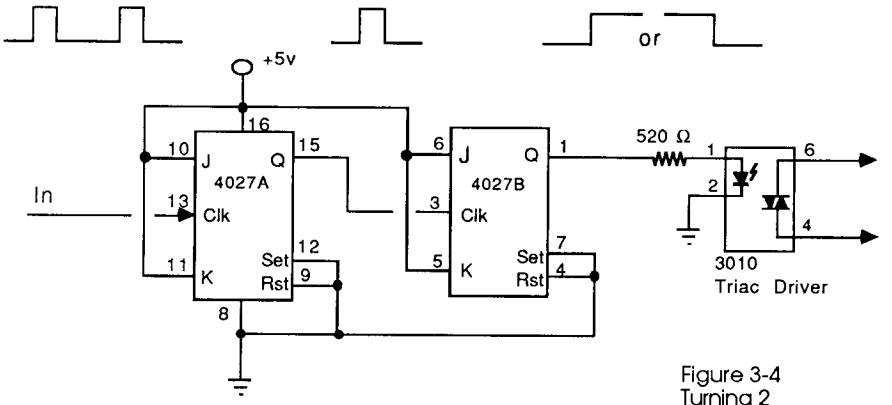


Figure 3-4
Turning 2
Pulses into
an On/Off
State.

If on the other hand your alarm catalog is not empty or a repeating alarm activates, two pulses come out and a second divide-by-two Flip-Flop is added to compensate, as in Fig. 3-4. For consistent behavior, I constructed Fig. 3-4, and always keep a dummy alarm, set to go off in 1999, in my ALMCAT. This guarantees two pulses at the ALMOUTA pad.

Now the big question is: how can we harness these new features and still retain the 41's neatness and portability? The answer depends on your configuration. If you have a C or CV model, use two Time Modules: one with wires coming out, one without. Then just swap the two when the occasion arises.

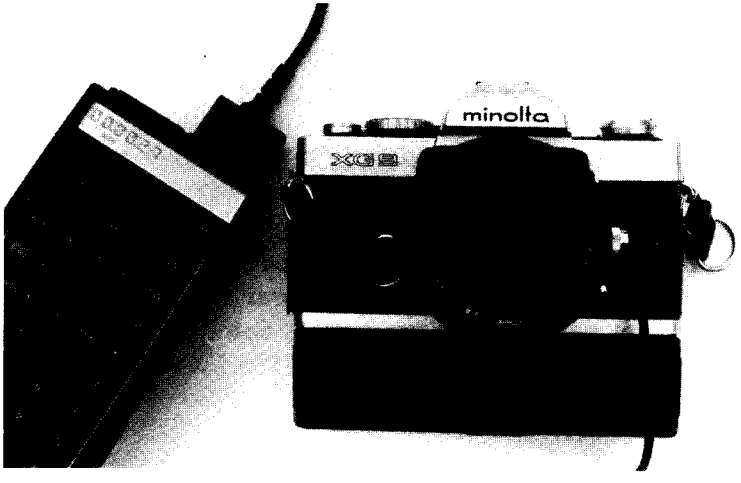
If you have a 41CX, then more work awaits you, but you get the fringe benefit of never losing your time, date, or accuracy factor when switching modules. Modification of the CX will be covered at the end of this chapter.

The Perfect Field Application

I purchased an extra time module, and got a hold of a plug and cable assembly originally designed for the 41's dedicated printer (82143A). (I scrounge at swap meets a lot.) I then proceeded to build into it the circuitry shown in Fig. 3-5. This handy little package allows maximum versatility, as the 7-conductor cable can carry 2 wires from the triac driver's output, as well as leads from pads 1 through 4, and Vcc. (No reference ground is needed.) This means that all the new I/O previously unattainable is now available to the outside world through this cable; and in addition the pulses coming from ALMOUTA are already conditioned!

The first thing I did with it was to hook the triac driver's output to my electronic 35MM camera and have it take time exposures at night, one of my all-time favorite applications.

Anyone who's ever taken pictures of the city at night knows that any good results are lucky ones. With the camera on a tripod and set to "B", the photographer can only guess as to how long to leave the shutter open for proper exposure. Good photographers will take several pictures with many different exposure times, hoping that one of the frames comes out right. This tedious procedure involves constant clock watching (ever try watching a clock in the dark?), boredom, and a stiff finger if you forgot your locking cable release.



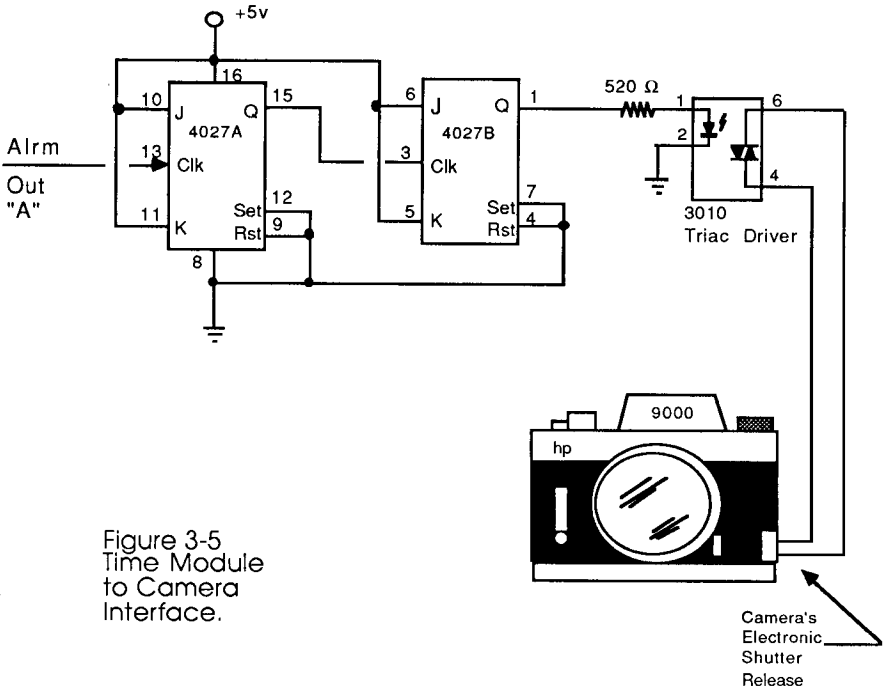


Figure 3-5
Time Module
to Camera
Interface.

If your camera has an electronic shutter--the kind that can be actuated by shorting a pair of contacts together instead of moving a mechanism--and if you have an automatic winder, then the time module adapter and the accompanying program called CAMERA can not only make time exposures fun again, but will also help guarantee a good result.

The program works as follows: After loading and executing, the program prompts for BASE TIME? Let's say we want to start with a 1-minute exposure. With the camera on "B", we hit 1 R/S, and the camera will open the shutter and the display will start counting up, a la stopwatch mode. After 1 minute has elapsed, the shutter closes, the camera winds itself, and the HP quickly calculates a new time to give the next frame 1/2 stop more exposure, and another picture is taken. The program will take 6 pictures, each increasing the exposure in 1/2 stop increments. With a base time of 1 minute, the 6 exposures will last 1:00, 1:24, 2:00, 2:49, 4:00, and 5:39. (Keep in mind that the shutter speeds are

a geometric progression, so halfway between 1:00 and 2:00 is not 1:30, but rather 1:24.5.) One of these times is highly likely to yield a good picture. When running the program with this sample time, one might notice a 1-second discrepancy; i.e. 4:00 being displayed as 3:59.99, but the correct duration is still employed.

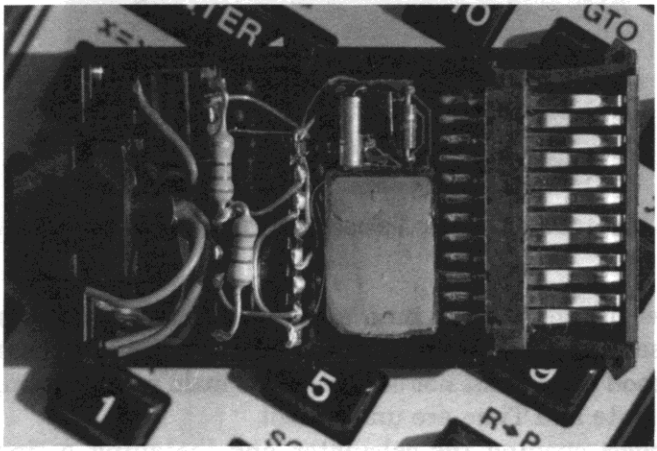
If you wish to halt all operations during an exposure, hit the R/S key. It will restore the clock time and tell you to CANCEL ALARM. Do it!! If this crucial step isn't done, your clock time and possibly the date will be changed.

Barcode for the CAMERA program, presented below, begins on page 271.

```
01*LBL "CAMERA"
02 CLST
03 10.201999
04 ENTER^
05 10.2
06 XYZALM
07 FIX 4
08 "BASE TIME?"
09 FC? 01
10 PROMPT
11 100
12 /
13 STO 01
14 1.006
15 STO 03
16*LBL 03
17 CLST
18 "^^OPEN"
19 TIME
20 .00015
21 HMS+
22 XYZALM
23 RTN
24*LBL "OPEN"
25 CLST
26 RCL 01
27 CLK24
28 "^^SHUT"
29 TIME
30 STO 02
31 .00005
32 HMS-
33 CHS
34 T+X
35 CLX
36 RDN
37 XYZALM
38 "  "
39 ATIME24
40 AVIEW
41 PSE
42 PSE
43 CLOCK
44 CLST
45 "^^CLOSE"
46 TIME
47 .00015
48 HMS+
49 XYZALM
50 RTN
51*LBL "CLOSE"
52 RCL 02
53 T+X
54 CLK12
55 "CANCEL ALARM"
56 AVIEW
57 RTN
58*LBL "SHUT"
59 RCL 02
60 T+X
61 CLK12
62 RCL 01
63 HR
64 X^2
65 2
66 *
67 SQRT
68 HMS
69 STO 01
70 ISG 03
71 GTO 03
72 "  THATS IT."
73 BEEP
74 AVIEW
75 RTN
76*LBL "BB"
77 CLST
78 "^^CC"
79 TIME
80 .00015
81 HMS+
82 XYZALM
83 RTN
84*LBL "CC"
85 END
```

Flag 1 is used when you set an alarm to have the sequence start while unattended. With flag 1 set, the number in the X register is taken as the base time without the program prompting.

This wonderful little setup allows you to walk away from your camera and have a dozen or so cups of coffee while your equipment does your bracketing for you. The program will allow for time exposures as long as 24 hours; however if you try that you'll find that the camera's batteries will die much earlier. This is because electronic shutters require battery power to electromagnetically hold back the second shutter curtain. When your batteries die, your exposure terminates.

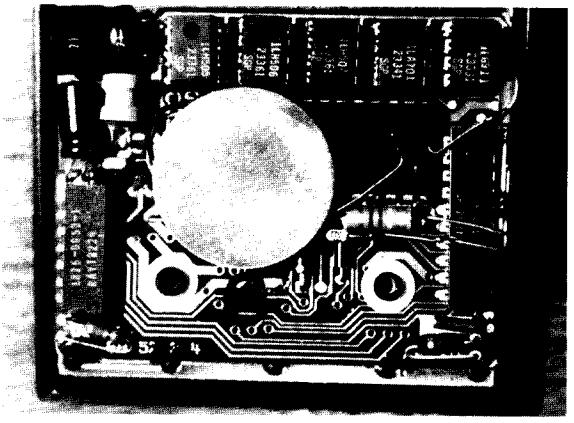


The Versatile Adapter, consisting of a time module, a CMOS Flip-Flop, and an opto-isolator, fits snugly inside a printer plug.

Information for CX Owners

So here's the thing: if you build the Time Module adapter as described above and tried to plug it into a CX, you would have two time modules competing with each other each time a time function is called. This renders things like CLOCK, DATE, and RCLSW useless, since their results would randomly come from one module or the other. (XEQ'ing CLOCK has some really interesting effects when two time modules are plugged in!)

The suggested solution may seem a little extreme, and some of you may not wish to attempt it. In the 41CX, the Phineas chip takes the form of a 20-pin DIP IC rather than in plug-in module form. It appears as the black vertical IC in the photo below. (The other parts on top of it are the components required for a speedup.)



Inside the 41CX. The time module chip is located at the lower right.

In order for an external time module to work without conflict, it is necessary to disable the internal 1LF6 chip. Fortunately, an easy method exists that not only requires little modification, but the internal Date and Time are unaffected!

It involves opening the calculator and installing a single, tiny DIP switch as shown in Figure 3-6. Two wires from this switch are then drawn over to the 20-pin DIP Phineas chip whose ISA (pin 7) has been severed between the chip and the PC board it's mounted on. (A tiny pair of wire cutters should be enough to clip this pin in half). Using the tiniest of low-wattage pencil soldering irons, one must solder each of the two DIP switch wires to each part of the split ISA pin.

The 41 uses the ISA line to either address its peripherals (ROM, RAM, IL Module, etc.) or to have the peripherals wake the 41 up. If this crucial line is severed the CPU doesn't "see" the time module, even though it is still there and still keeping good time. With the isolation complete, we can now plug the Time Module Adapter

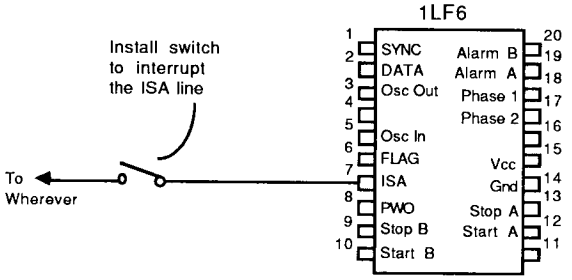
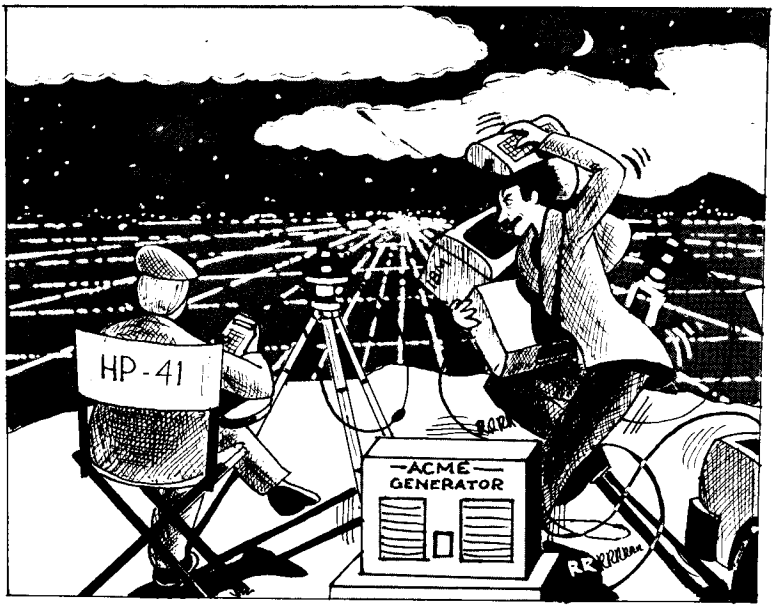


Figure 3-6.
How to disable the timer chip's ISA line on the 41-CX.

into any of the 41's four ports and use it normally. When the application is completed and the adapter removed, a flip of the DIP switch will reinstate the internal Time Module without loss of time or date, and any past due alarms will immediately start processing.

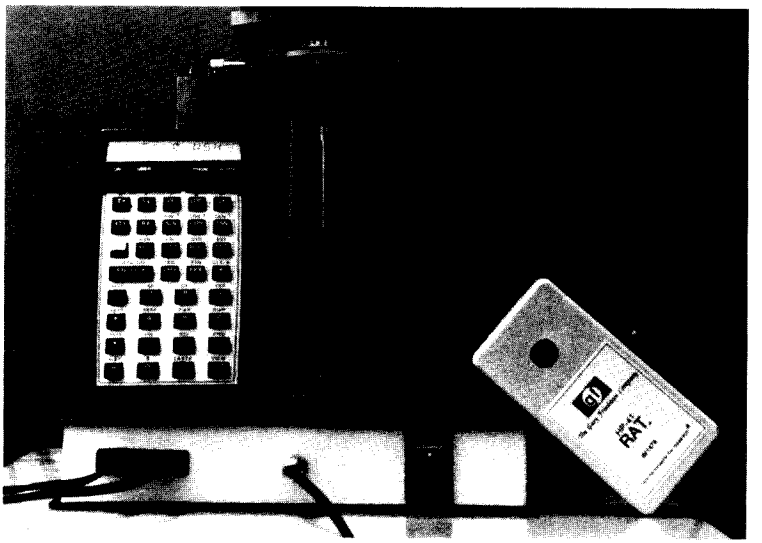
I realize that most people are not well-versed in micro-electronic surgery, so I will issue the standard warning: If you're not



confident with your skills in electronic assembly, get someone who is to perform the modification for you. (There are also services available for those of you who would like to have the work done by somebody else. See Appendix B for more information.)

For applications involving turning only one item on and off, this new time module technique is superior in terms of size, power source, and cost. When it comes to taking crucial time measurements, the time module provides the only means of doing so. The two capabilities provide an unexploited method of real world control that rivals many desktop systems.

A DIP switch on the 41's exterior can temporarily disable the internal time module while the versatile adapter is plugged in. The module can be made "invisible" without the loss of time, date or accuracy factor.


RAT.

Chapter Four

DARKROOM CONTROLLER

"Someday my prints will come..."
-- Snow White after her pictures didn't arrive.

Here is one of my all-time favorite applications. Not only has it shortened my sessions in the darkroom and increased my productivity, but it is clearly a case where computers can help take the tedium out of work and leave the user free to concentrate on creative processes.

You will find that I will try to introduce new techniques in every chapter while building on previous knowledge, and this chapter is no exception. Here, with the introduction of the analog to digital converter, I will describe a 41-based darkroom controller that performs the following useful functions:

-- It acts as an enlarger timer, precisely controlling the duration of the enlarger's light source. The latest exposure time is always remembered, and can be accurately reproduced or "tweaked" as the user sees fit.

-- It keeps track of all prints in all chemical baths, and issues distinct audible signals to announce when it's time to move a print.

-- It computes the change in exposure time necessary when printing with variable-contrast papers. Changing filters is no longer a trial-and-error task.

-- Using an analog-to-digital converter, it takes N readings from the negative and recommends both an exposure time and a filter to achieve a technically good print.

This chapter will be divided into two parts: The first part is quite straightforward; it explains how to build a simple enlarger and print timer using familiar hardware and software techniques. The second part introduces the A/D converter for further automation benefits, and a few software techniques to improve the response time and consume less battery power.

Part I: A Simple Start

The first phase of this project uses circuitry introduced in previous chapters, for no other reason than to show that slight rearrangement of the techniques already covered can yield any number of truly useful systems.

The hardware needed is that which turns on and off a light bulb. (The light bulb in this case happens to be surrounded by an enlarger.) (See Fig. 4-1.) This and a program which uses the Extended Functions/Memory module are all that are needed to implement all but the last feature in the above list.

With the circuitry in Figure 4-1, all you have to do is a "32 ACCHR" to turn on the light, and a "0 ACCHR" to turn it off, and in fact this is precisely how the FOCUS function (assigned to the + key) is implemented in the program provided below. But before talking more, some history describing the evolution of this system is in order.

A simple print timer

The original need developed (no pun intended) from my many tedious sessions in the darkroom. After a print was properly developed, it would have to soak in a chemical bath (called the fixer) for 2 minutes, then promptly be moved to a wash bath for 4 minutes. On busy nights there were always 1 or 2 prints in the

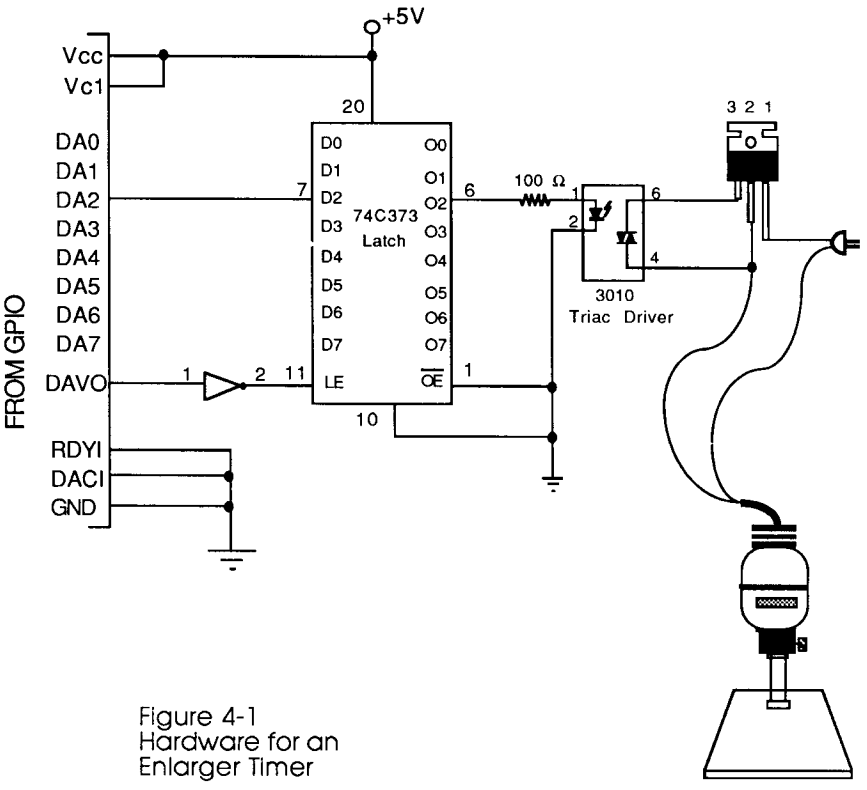


Figure 4-1
Hardware for an
Enlarger Timer

fixer and at least 3 in the wash at any given time, and mentally keeping track of when to move what print while at the same time concentrating on the creative aspects at the enlarger proved to be quite taxing. What was needed was something that would automatically keep track of the prints for me, and quietly announce when it was time for each to be moved.

Using only a 41C and a time module, this proved to be an easy software task. As soon as a print enters the fixer, the R/S key is pressed, which promptly sets 2 control alarms using the XYZALM function of the time module. After 2 minutes elapse (when it's time to move the print to the wash), the first alarm (at label A1) activates, "chirps" three times to announce that a print should be moved, and updates the display, which always shows a running count of how many prints are in the fixer and how many are in the

wash. After an additional 4 minutes (when it's time to remove the print from the wash and hang it up to dry), the 2nd alarm (at label A2) activates, "beeps" three times to announce that a print is ready to be dried, and again updates the scoreboard.

A darkroom is a rather noisy environment, which is why it is advantageous to have two different, very distinguishable sounds generated for each alarm. A unique sound which I call the "chirp" provides this instant recognizability, and is generated via a whole string of TONE 89s, the synthetic instructions occupying lines 54-61 and 66-73. (See DKRM3 listing later.) Three conventional TONE 6s provide the other warning, therefore allowing instant recognition of which print is to be moved without having to consult the 41's display.

The first two lines of the program show another advantage of synthetic instructions: the ability to set many system parameters, which are normally not changeable under program control, at once. Here its main advantage is setting the continuous-on flag (flag 44), a feat normally accomplished by manually XEQing ON (a non-programmable function).

Software for added Functionality

Extra software was added to control the duration of the enlarger lamp, thereby turning this simple capability into more of a time-saving tool. The system is controlled by the global assignments that have been PASN'd at the program's onset as illustrated in Figure 4-2.

As mentioned earlier, pressing the "+" key toggles the enlarger on and off for focusing. The "*" (MANual) key does the same thing, except for this key the calculator remembers how long the enlarger was on and stores it in R04. This way, I can make an educated guess as to the exposure on a test print. If it was right, the final print can be exposed simply by hitting "/" (AUTO) and the enlarger is activated for the exact time of the previous trial.

If my guess was incorrect, the exposure can be "tweaked" by keying the adjustment into the X register and hitting "-" (ADJust). If 2 seconds less exposure is desired, for example, I key in "2 CHS -". Hitting "/" (AUTO) then implements this new time. Once the

correct exposure has been determined, making hundreds of identical reprints from the same negative is accomplished just by hitting AUTO.

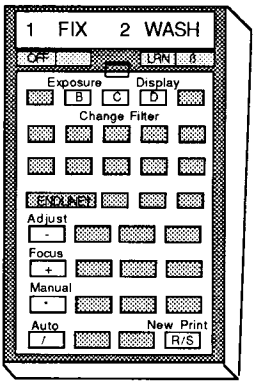


Figure 4-2:
Darkroom Controller
Key Assignments

Another darkroom task best left to a computer is the calculation of exposure adjustments when changing enlarger filters. When I feel the exposure is right but the contrast needs changing, I'd press "C" (for "Change"), and the 41 prompts me for the old and new Polycontrast filter number, and then adjusts the exposure time (it always remembers the last exposure time) accordingly. The "/" (AUTO) key implements this new time.

The program that performs all this stuff is listed below. Barcode begins on page 273.

```
01*LBL "DKRM3"
02 "++D<+++"
03 RCL M
04 STO d
05 "FOC"
06 61
07 PASN
08 "MAN"
09 71
10 PASN
11 "AUTO"
12 81
13 PASN
14 "ADJ"
15 51
16 PASN
17 CLRG
18 GTO 02
19*LBL 01
20 CLST
21 TIME
22 STO 01
23 .0215
24 HMS+
25 24
26 X<=Y?
27 XEQ 03
28 RDN
29 "^^A1"
30 XYZALM
31 CLST
32 RCL 01
33 .07
34 HMS+
35 24
36 X<=Y?
37 XEQ 03
38 RDN
39 "^^A2"
40 XYZALM
41  E
42 ST+ 02
43*LBL 02
44*LBL D
45 FIX 0
46 CF 21
47 CF 22
48 CF 29
49 " "
50 ARCL 02
51 "| FX. "
52 ARCL 03
53 "| WSH"
54 AVIEW
55 RTN
56 GTO 01
57*LBL 03
58 MOD
59 TIME
60 INT
```

```
 61 X=0?
 62 RTN
 63 RDN
 64 DATE
 65  E
 66 DATE+
 67 X<>Y
 68 .
 69 RTN
 70*LBL B
 71 FIX 6
 72 "  "
 73 RCL 04
 74 ATIME24
 75 CF 21
 76 AVIEW
 77 SF 21
 78 CF 22
 79 RTN
 80 GTO 01
 81*LBL "A1"
 82  E
 83 ST- 02
 84 ST+ 03
 85 RDN
 86 XEQ "T1"
 87 PSE
 88 XEQ "T1"
 89 PSE
 90 XEQ "T1"
 91 FS? 03
 92 RTN
 93 GTO 02
 94*LBL "A2"
 95  E
 96 ST- 03
 97 RDN
 98 TONE 6
 99 PSE
100 TONE 6
101 PSE
102 TONE 6
103 FS? 03
104 RTN
105 GTO 02
106*LBL "MAN"
107 FC?C 02
108 GTO 08
109 .
110 ACCHR
111 TONE 87
112 RCLSW
113 STO 04
114 CF 22
115 ALMNOW
116 GTO 02
117*LBL 08
118 "+i+++"
119 X<> M
120 X<> c
121 ALMNOW
122 X<> c
123 SF 21
124 .
125 STOPSW
126 SETSW
127 32
128 ACCHR
129 TONE 87
130 RUNSW
131 SF 02
132 CF 21
133 "   MANUAL"
134 AVIEW
135 SF 21
136 RTN
137 GTO 01
138*LBL E
139 CLOCK
140 GTO 01
141*LBL "FOC"
142 SF 21
143 FS? 01
144 GTO 05
145 32
146 ACCHR
147 "   FOCUS"
148 CF 21
149 AVIEW
150 SF 01
151 RTN
152 GTO 01
153*LBL 05
154 .
155 ACCHR
156 CF 01
157 GTO 02
158*LBL "AUTO"
159 FS?C 22
160 XEQ 09
161 "+i+++"
162 X<> M
163 X<> c
164 ALMNOW
165 X<> c
166 SF 03
167 SF 21
168 32
169 ACCHR
170 TONE 87
171 "^^EOFF"
172 TIME
173 RCL 04
174 HMS+
175 5 E-5
176 HMS-
177 +
178 CLST
179 LASTX
180 24
181 X<=Y?
182 XEQ 03
183 RDN
```

```
184 XYZALM
185 ALMNOW
186 GTO B
187*LBL "EOFF"
188 SF 21
189 .
190 ACCHR
191 TONE 87
192 CF 03
193 GTO 02
194*LBL 09
195  E4
196 /
197 STO 04
198 RTN
199*LBL "ADJ"
200  E4
201 /
202 RCL 04
203 HMS+
204 STO 04
205 GTO B
206*LBL "FLTR"
207*LBL C
208 "FILTER #1?"
209 PROMPT
210 2
211 *
212 10
213 +
214 XEQ IND X
215 "FILTER #2?"
216 PROMPT
217 2
218 *
219 10
220 +
221 XEQ IND X
222 X<>Y
223 RDN
224 /
225 RCL 04
226 HR
227 *
228 HMS
229 STO 04
230 GTO B
231*LBL 10
232 500
233 RTN
234*LBL 12
235 250
236 RTN
237*LBL 13
238 400
239 RTN
240*LBL 14
241 320
242 RTN
243*LBL 15
244 320
245 RTN
246*LBL 16
247 250
248 RTN
249*LBL 17
250 100
251 RTN
252*LBL 18
253 80
254 RTN
255*LBL e
256 CLA
257 61
258 PASN
259 71
260 PASN
261 81
262 PASN
263 51
264 PASN
265 FIX 4
266 CLST
267 RTN
268*LBL "T1"
269 TONE 89
270 TONE 89
271 TONE 89
272 TONE 89
273 TONE 89
274 TONE 89
275 TONE 89
276 TONE 89
277 TONE 89
278 TONE 89
279 TONE 89
280 TONE 89
281 TONE 89
282 TONE 89
283 TONE 89
284 TONE 89
285 END
```

Synthetic Text Lines:

```
 02: 68,60,132,136
118: 1,105,1,0,16
161: 1,105,1,0,16
```

Part II: Make it Better

Now that most of the tedium has been removed from my darkroom work (courtesy of my 41), I felt it was time to let my calculator take over one additional aspect: take the guesswork out of the initial exposure time by measuring the light hitting the paper and recommending exposure time and filtration. This is where some additional hardware is needed.

The A/D Converter

The chip used here, the National Semiconductor ADC0804, is your basic, no-frills, been-around-forever analog-to-digital converter which takes a variable voltage (in this case a light-sensitive cadmium sulfide photo cell) as its input and provides an 8-bit number at its output. (See Figure 4-3.)

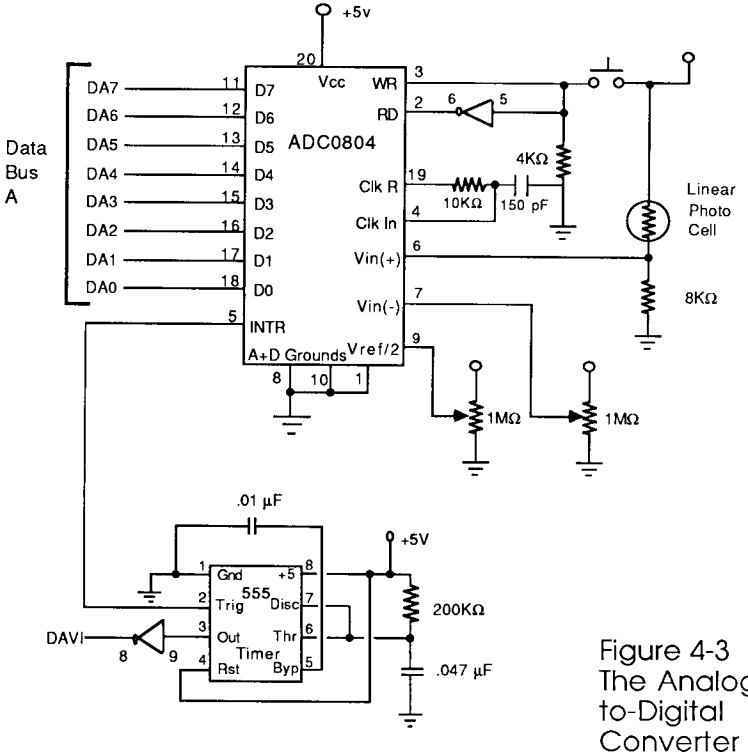


Figure 4-3
The Analog-to-Digital Converter

The underlying principles behind analog to digital converters are actually quite simple. Using a two-input element called a comparator, the converter compares the voltage supplied at the input with an internally-generated voltage that starts at 0v and continuously rises until it matches the input voltage, at which time the comparator says "Okay, they're equal". At that instant, the voltage increase halts, and an 8-bit counter (whose value is proportional to the rising reference voltage) is displayed at the A/D converter's output.

The ADC0804 uses something called differential input mode, which takes as its analog input the difference in voltage between the (+) and (-) inputs. (Pins 6 and 7.) As will be seen shortly, this feature combined with the Vref/2 pin (pin 9) allows for an incredible amount of different configurations. This is important, since it means we can use anything as input. Using external resistors, we can program the chip to scale its output up or down and allow us to measure the intensities of solar flares or fireflies; sound intensities of glass being broken or space shuttles taking off; temperature variations on the Earth or on Venus. In short, you can magnify or compress the 256-unit range to fit any form of any input you care to supply!

The lower half of Figure 4-3 also contains something called a pulse expander, comprised of a commonly available 555 IC timer and some support components. To understand its function, recall the handshaking scheme described in Chapter 1. A pulse must appear on the DAVI line telling the GPIO that the stuff it sees on the data lines is valid. Well, what wasn't mentioned is that this DAVI input will only accept pulses that last a minimum of 60 milliseconds. The pulse width of the A/D converter's STROBE line output is only 10 ms, a mere 1/6 of the width required by the GPIO.

This pulse expander does exactly what its name implies: it takes a pulse of any duration from the input marked STROBE, and outputs to DAVI a fixed length, 100 ms pulse, much larger than the required width. When doing any type of general-purpose interfacing, it is a useful circuit to have for conditioning any signals, since it debounces as well as expands its input.

For this application, I wish to measure the amount of light coming from an enlarger using a photo cell, a device whose

resistance decreases proportionally with the amount of light that hits it. The three challenges here are 1) somehow convert a variable resistance into a variable voltage for the A/D converter, 2) "program" the converter to give a full-scale swing in response to the relatively small variance in intensities coming from the enlarger, and finally 3) make the A/D converter's output meaningful to the computer that will be receiving it.

Voltage Divider

The solution to the first challenge is to use the most basic of electric circuits that uses only one other resistor: the voltage divider. In theory, if you have two resistors of equal value arranged in series across a 5-v source as in Fig. 4-4, the voltage at the midpoint would be 2.5 volts.

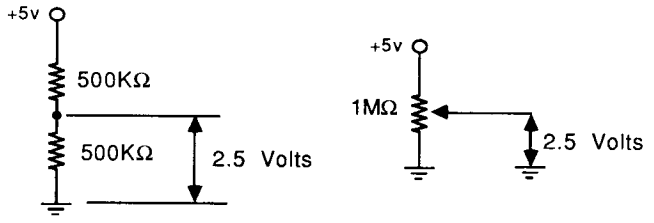


Figure 4-4
Two Equivalent
Voltage Dividers

The two-step method for deriving the midpoint voltage using two different resistors (as illustrated in Fig. 4-5) comes directly from Ohm's Law: Volts = Current X Resistance. First, calculate the total current running through these resistors:

I (=current) = V/R = 5v/1,000,000 Ohms = .005 milliamps

Then, starting from the top, we just subtract the voltage drop due to each resistor to get the voltage at that point. In this case:

5 - (.005 mA)x(300 KOhms) = 3.5 volts.

The steps involved in adapting this converter to your specific photo cell (not all behave the same way) are very straightforward:

1) Go into your darkroom with an ohmmeter and a photocell. Making sure that no stray safelight rays hit the cell, take two readings off the ohmmeter: the photocell's resistance with the lamp shut off, (call this reading "R(d)") and with the lamp turned on with no negatives or filters in place and the enlarger head all the way down (label this reading "R(l)"). These values correspond to the darkest and lightest your negative can ever be. If you find that your two readings do not differ greatly (i.e. by less than a factor of 10), try a different kind of photocell. I personally find that Cadmium Sulfide (CdS) cells work best in this environment.

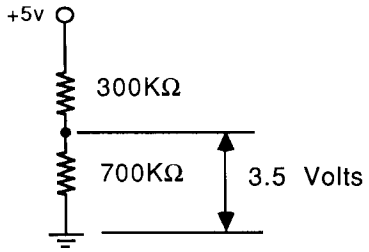


Figure 4.5
Voltage Divider Using
Unequal Resistors

2) The photo cell will be used as the top resistor in a voltage divider as pictured in Fig. 4-5. In this way, the change in resistance will directly affect the voltage appearing at the midpoint. A value of the lower resistor that will cause the greatest voltage swing for the measured resistance values must now be calculated. The formula for the voltage swing is:

$$\text{Voltage Swing} = 5\left(\frac{R_d}{R_d+X} - \frac{R_L}{R_L+X}\right)$$

where X is the value of the bottom resistor. Try "plugging in" several values of X until the voltage swing hits a maximum value. (You may wish to invest in a fine pocket calculator to help speed up

the maximization of the above expression.)

Finding the proper value for the bottom resistor will increase the sensitivity of the A/D Converter for your particular environment, and allows you to go on to the next step, which is:

Building the Circuit

Figure 4-6 shows the entire circuit for the darkroom controller. For ease of use, this circuit is contained in two separate housings: the base unit, which contains the power supply, A/D converter, triac, IL Converter, 555 timer, and AC receptacle for the enlarger; and the "RAT" (which is sort of like a mouse except the trackball on the bottom has been replaced by a photocell on top), which contains the pushbutton switch and the photocell. The rat is attached to the main housing by a three-conductor cable, which carries the wires eminating from the dashed box in Figure 4-6.

Because this project involves turning AC devices on and off, and because it is to be used in the vicinity of water and chemicals, it is important to take some extra precautions when building the enclosure and placing the components. Be certain all AC connections are well insulated; building it into a plastic box is extra insurance against accidental shock. For extra safety, add a fuse with the same rating as the enlarger to the power cable.

Calibrating the Circuit in the Darkroom for the First Time

After building the hardware and loading the DKRM4 program below, the final step is to calibrate the A/D converter for your particular enlarger. To do this, you need a set of calibrated reference negatives. This is easy to obtain; just get a photographic 18% grey card and, using a 35MM camera, take 7 pictures of it, each frame having one stop more exposure than the last. Start with correct exposure, then overexpose one, two, and three stops, and then underexpose one, two, and three stops. An unexposed frame (shoot with the lens cap on) is also needed for calibration. It's helpful to mark the negatives with their respective exposures,

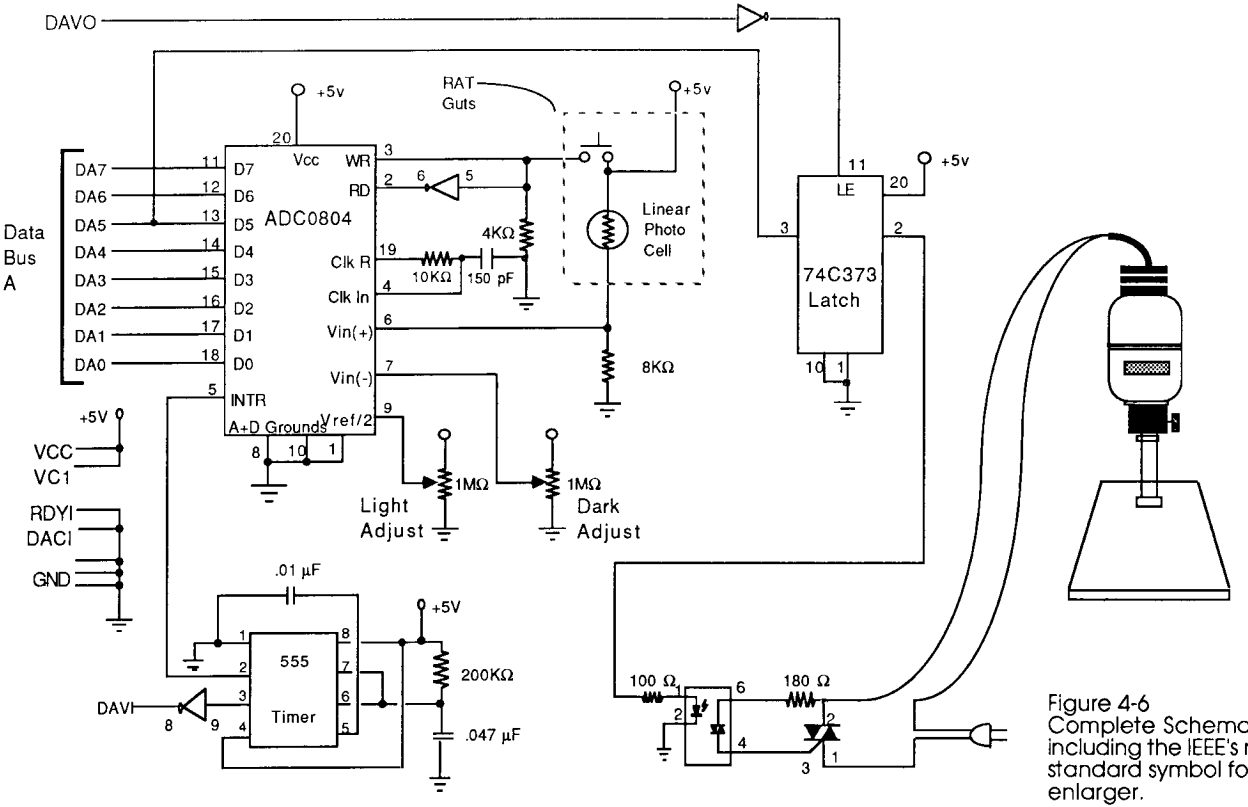


Figure 4-6
Complete Schematic, including the IEEE's new standard symbol for an enlarger.

so when comparing "corrected" prints later on you'll know what exposure you started with.

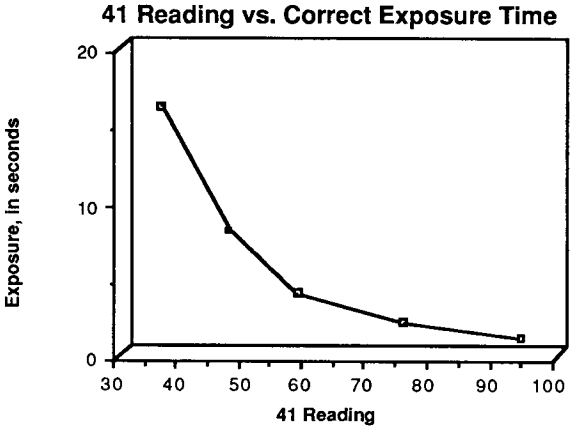
1) First, position the Vref/2 potentiometer (the one connected to pin 9 of the ADC) at the halfway position, setting that pin at 2.5 volts. (Disconnecting pin 9 achieves the same thing.)

2) With the photocell safely shielded from the safelight, take a reading of the ambient light using the INTR routine from the program DKRM4. Adjust the Vin potentiometer (the one connected to pin 7) until the HP-41 just reads zero. This pot adjusts the offset, and insures that both V+ and V- are at the same voltage when the reading should ideally be zero.

3) Turn the enlarger on. With the housing lowered all the way and with the unexposed negative in the carrier, slowly adjust the Vref/2 potentiometer until the readings just hit 255, its full-scale value. Just seeing "255" off the 41's display is not enough; the ADC may be displaying an overflow value without telling you. Because we're making use of the A/D converter's full scale to represent our relatively narrow range of light intensity, we can now make very accurate readings.

4) Determine the Exposure Curve. Take the 7-negative strip of grey card exposures described earlier and make an accurate print of each frame, without filters, using the old-fashioned trial-and-error method. This will yield 7 identical photos of grey cards, so it is best to mark each photo with initial exposure (-1, +2, etc.), enlarger time, and A/D reading. Here are the numbers I came up with:

EV	41 Reading ↓	Exposure (sec)
-2	94	1
-1	75	2
0	58	4
+1	47	8
+2	36	16



This table is used to determine the proper exposure given a reading of an 18% grey area of the desired negative. The equation describing this graph turns out to be:

$$\# \text{ of seconds} = 5.38E5 \times (\text{reading})^{-2.9}$$

Again, the numbers and the equations will be different for every cell, resistor, and enlarger combination. Those displayed here are only examples.

5) Make adjustments to the filtration guides. When two or more readings are taken (lightest, darkest, and 18% grey), the calculator recommends a polycontrast filter # and an exposure time already compensated for that filter. The rules this program follows for determining the filter are as follows:

Range (lightest-darkest)	Filter #
< 35	4
36-45	3.5
46-55	3
56-65	2.5
66-75	2
76-85	1.5
86 >	1

As you gain more experience with the darkroom controller, these numbers may be tweaked to meet your personal preferences. It is an easy thing to do; the constants are stored as comparison numbers at LBL 33. But since this unit's only purpose is to give a general recommendation, these numbers should suffice.

Complete Instructions for the Darkroom Controller

1) With all the hardware hooked up, XEQ DKRM4 (or DKRM3 if you're using the simplified version without the A/D converter.)

2) To turn the enlarger lamp on for focusing, hit the FOCus (the "+") key. To turn the enlarger off, hit the FOCus key again.

3) Once the image has been sized and focused, there are 3 ways to tell the unit how long to operate:

A) Using the 41's keyboard, punch in the desired number of seconds and hit AUTO (the / key). This time is implemented and stored in R04.

B) Use the MANual (the *) key to turn the lamp on. When you feel the print has absorbed enough light, hit the MAN key again to turn the enlarger off. The duration just implemented is automatically measured and stored in R04 for future use.

C) (This applies only to the A/D converter version.) Use the "RAT" to measure the light coming from the negative and have the 41 calculate the proper exposure. (See below, "Using the RAT".)

4) Develop the print in the normal fashion. When the print enters the fixing bath, hit the R/S key. This sets 2 alarms which will automatically remind you when the print should change baths.

With the RAT, the above steps should yield a technically acceptable print. Many adjustments, however, may be necessary in order to obtain an artistically excellent one. To make these adjustments, the following steps should be used:

5) To 'tweak' the enlarger time, key in the number of seconds to add to R04 and hit ADJust (the - key). For example, if you feel the print needs 2 seconds less to make it perfect, key in 2 CHS ADJ and the time is altered. Hitting AUTO implements the new time.

6) If a different filter for contrast is desired, the new exposure time can automatically be calculated. Press "C" (standing for Change filter) and answer the prompts for "Filter #1" and "Filter #2". (DKRM4 will only ask for filter #2, since it automatically keeps track of filter #1.) A new exposure time will be calculated based on the time already stored in R04.

7) To view the current FIX-WASH scoreboard, press "D" (for Display) at any time.

8) To view the current enlarger time and filtration, press "B".

9) After an incredibly productive evening, XEQ e to remove all the global assignments and return the display to something normal.

Using the RAT

The software supporting the RAT (it's a much more descriptive term than, say, a MOUSE) allows you to measure different aspects of the negative and calculate the proper exposure and filtration in different ways.

10) Turn the enlarger lamp on by pressing the FOCus (the +) key once. (Clicking once on the RAT's button does the same thing.)

11) By making one reading on a portion of the negative that is the equivalent to 18% grey, the 41 will calculate the proper exposure.

12) By making two readings on the lightest and darkest portions of the negative, the 41 will determine the negative's density range and recommend a polycontrast filter to match the paper's range

with that of the negative. The exposure is calculated based on the midpoint between the measured spots.

13) By making three (or more) readings on the lightest, darkest, and "18%-est" areas of the negative (in any order), the 41 will recommend a polycontrast filter number based on the negative's extremes, and an exposure based on the average of all readings.

14) After all the readings are taken, hit the FOCus button to shut off the enlarger and start the calculations going. The new exposure time and filtration are then prominently displayed.

Phase II Software

Here's the software. As always, barcode for this program can be found in Appendix A, pg. 277.

```
01*LBL "DKRM4"
02 "`<O +O +O +"
03 RCL M
04 STO d
05 7
06 BSIZEX
07 0
08 PT=
09 130
10 X-BUF
11 1
12 LAD
13 0
14 DDL
15 1
16 OUTBUFX
17 UNL
18 3
19 ENTER^
20 64
21 WREG
22 "FOC"
23 61
24 PASN
25 "MAN"
26 71
27 PASN
28 "AUTO"
29 81
30 PASN
31 "ADJ"
32 51
33 PASN
34 ~REG 05
35 CLRG
36 GTO D
37*LBL 20
38 RTN
39 CLST
40 TIME
41 STO 01
42 .0215
43 HMS+
44 24
45 X<=Y?
46 XEQ 03
47 RDN
48 "^^A1"
49 XYZALM
50 CLST
51 RCL 01
52 .07
53 HMS+
54 24
55 X<=Y?
56 XEQ 03
57 RDN
58 "^^A2"
59 XYZALM
60  E
61 ST+ 02
62*LBL D
63 FIX 0
64 CF 21
65 SF 18
66 CF 22
```

```
67 CF 29
68 " "
69 ARCL 02
70 "> | FX. "
71 ARCL 03
72 "> | WSH"
73 AVIEW
74 AUTOIO
75 GTO 20
76*LBL C
77 "FILTER #2?"
78 PROMPT
79 RCL 00
80 2
81 *
82 10
83 +
84 XEQ IND X
85 RCL Z
86 2
87 *
88 10
89 +
90 XEQ IND X
91 X<>Y
92 RDN
93 /
94 RCL 04
95 HR
96 *
97 HMS
98 STO 04
99 GTO B
100*LBL 03
101 MOD
102 TIME
103 INT
104 X=0?
105 RTN
106 RDN
107 DATE
108  E
109 DATE+
110 X<>Y
111 .
112 RTN
113*LBL B
114 XEQ 04
115 GTO 20
116*LBL 04
117 FIX 6
118 RCL 04
119 CLA
120 ATIME24
121 "> |  #"
122 FIX 1
123 ARCL 00
124 ASHF
125 32
126 X-AL
127 CF 21
128 AVIEW
129 SF 21
130 CF 22
131 AUTOIO
132 RTN
133*LBL "A1"
134  E
135 ST- 02
136 ST+ 03
137 RDN
138 XEQ "T1"
139 PSE
140 XEQ "T1"
141 PSE
142 XEQ "T1"
143 FS? 03
144 RTN
```

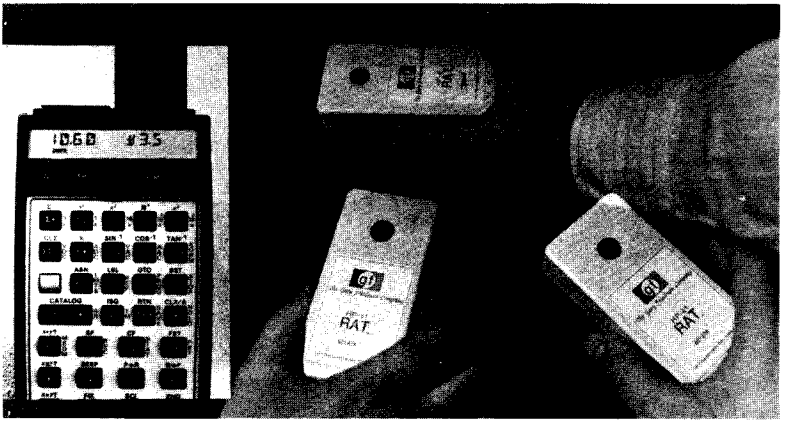
And now, a walk-through of the Darkroom Controller's use.

Follow these simple steps for faster, easier printing!

```
145 GTO D
146*LBL "A2"
147  E
148 ST- 03
149 RDN
150 TONE 6
151 PSE
152 TONE 6
153 PSE
154 TONE 6
155 FS? 03
156 RTN
157 GTO D
158*LBL "MAN"
159 MANIO
160 SF 21
161 CF 18
162 FC?C 02
163 GTO 08
164 .
165 ACCHR
166 TONE 87
167 RCLSW
168 STO 04
169 CF 22
170 ALMNOW
171 GTO D
172*LBL 08
173 "O +iO +O +O +"
174 X<> M
175 X<> c
176 ALMNOW
177 X<> c
178 SF 21
179 .
180 STOPSW
181 SETSW
182 32
183 ACCHR
184 TONE 87
185 RUNSW
186 SF 02
187 CF 21
188 "   MANUAL"
189 AVIEW
190 SF 21
191 GTO 20
192*LBL "FOC"
193 MANIO
194 SF 21
195 SF 18
196 FS? 08
197 GTO 05
198 32
199 ACCHR
200 AUTOIO
201 3
202 ENTER^
203 64
204 WREG
205 CL~
206 "   FOCUS"
207 CF 21
208 AVIEW
209 SF 08
210 GTO 20
211*LBL 05
212 .
213 ACCHR
```

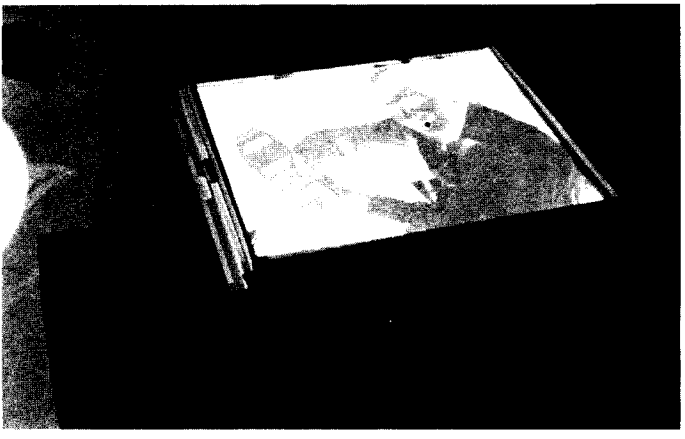
Step 1 Hit the FOCus key and focus as usual.

```
214 CF 08
215 RCL 10
216 X=0?
217 GTO D
218 GTO 22
219*LBL "AUTO"
220 MANIO
221 SF 21
222 CF 18
223 FS?C 22
224 XEQ 09
225 "O +iO +O +O +"
226 X<> M
227 X<> c
228 ALMNOW
229 X<> c
230 SF 03
231 SF 21
232 32
233 ACCHR
234 TONE 87
235 "^^EOFF"
236 TIME
237 RCL 04
238 HMS+
239 5 E-5
240 HMS-
241 +
242 CLST
243 LASTX
244 24
245 X<=Y?
246 XEQ 03
247 RDN
248 XYZALM
249 ALMNOW
250 GTO B
251*LBL "EOFF"
252 MANIO
253 SF 21
254 .
255 ACCHR
256 TONE 87
257 CF 03
258 GTO D
259*LBL 09
260  E4
261 /
262 STO 04
263 RTN
264*LBL "ADJ"
265  E4
266 /
267 RCL 04
268 HMS+
269 STO 04
270 GTO B
271*LBL 10
272 0
273 STO 00
274 RDN
275 500
276 RTN
277*LBL 12
278 1
279 STO 00
280 RDN
281 250
282 RTN
```



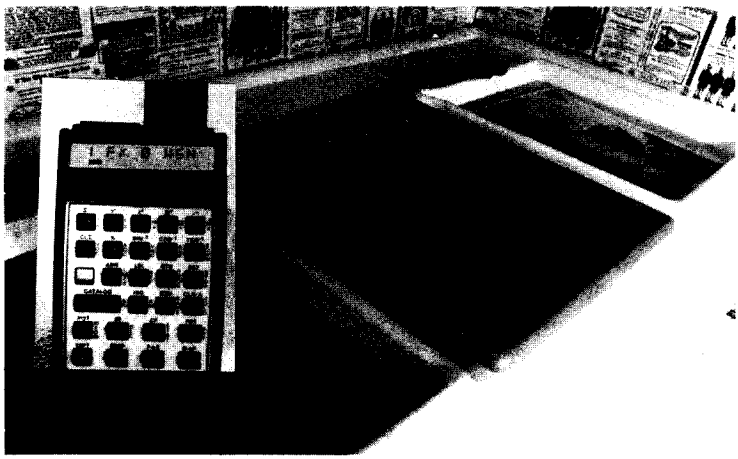
Step 2 Using the RAT, take density readings of the brightest, darkest, and "18% grey-est" areas of your negative. The 41 will recommend a contrast filter and exposure time.

```
283*LBL 13
284 1.5
285 STO 00
286 RDN
287 400
288 RTN
289*LBL 14
290 2
291 STO 00
292 RDN
293 320
294 RTN
295*LBL 15
296 2.5
297 STO 00
298 RDN
299 320
300 RTN
301*LBL 16
302 3
303 STO 00
304 RDN
305 250
306 RTN
307*LBL 17
308 3.5
309 STO 00
310 RDN
311 100
312 RTN
313*LBL 18
314 4
315 STO 00
316 RDN
317 80
318 RTN
319*LBL 22
320 MEAN
321 XEQ 31
322 RCL 10
323 1
324 X=Y?
325 GTO B
326 RCL 11
327 RCL 12
328 XEQ 32
329 GTO B
330*LBL 31
331 XEQ "EXPOSUR"
332 1 E4
333 /
334 STO 04
335 .
336 STO 00
337 RTN
338*LBL 32
339 -
340 XEQ 33
341 500
342 X<>Y
343 /
344 RCL 04
345 HR
346 *
347 HMS
348 STO 04
349 RTN
350*LBL 33
351 95
```



Step 3 With paper in the easel and filter in place, press AUTO to automatically expose for the recommended time.

```
352 X<=Y?
353 GTO 12
354 RDN
355 85
356 X<=Y?
357 GTO 13
358 RDN
359 75
360 X<=Y?
361 GTO 14
362 RDN
363 65
364 X<=Y?
365 GTO 15
366 RDN
367 55
368 X<=Y?
369 GTO 16
370 RDN
371 45
372 X<=Y?
373 GTO 17
374 RDN
375 GTO 18
376*LBL e
377 CLA
378 61
379 PASN
380 71
381 PASN
382 81
383 PASN
384 51
385 PASN
386 FIX 4
387 CLST
388 RTN
389*LBL "INTR"
390 CLD
391 FS? 08
392 GTO 02
393 MANIO
394 SF 21
395 32
396 ACCHR
397 SF 08
398 CL~
399 1 E9
400 STO 12
401 .
402 STO 11
403 AUTOIO
404 RTN
405*LBL 02
406 1
407 TAD
408 INBUFX
409 UNT
410 BUF-XB
411 INSTAT
412 RDN
413 FS? 01
414 GTO 02
415 CF 00
416 128
417 X<>Y
418 X=Y?
419 GTO 37
420 255
```



Step 4 Develop the print in the usual manner. Hit the R/S key when it gets to the fixer. The 41 will log the print and keep track of it from here.

```
421 X<>Y          444*LBL "T1"      467 *
422 X=Y?          445 TONE 89       468 END
423 GTO 37        446 TONE 89
424 TONE 89       447 TONE 89
425 RCL 11        448 TONE 89
426 X<>Y          449 TONE 89
427 X>Y?          450 TONE 89
428 STO 11        451 TONE 89
429 RCL 12        452 TONE 89
430 X<>Y          453 TONE 89
431 X<Y?          454 TONE 89
432 STO 12        455 TONE 89
433 ENTER^        456 TONE 89
434 ~+            457 TONE 89
435 RDN           458 TONE 89
436 RTN           459 TONE 89
437*LBL 37        460 TONE 89
438 TONE 13       461 TONE 89
439 CF 21         462 RTN
440 " TRY AGAIN" 463*LBL "EXPOSUR"
441 AVIEW         464 -2.9
442 SF 21         465 Y^X
443 RTN           466 537933.4
```

Step 5 Go about your business. The 41 will 'chirp' when it's time to move the print to the wash, and later 'beep' when it's time to hang it up to dry. This system will keep track of as many photos in process as you have memory.

Analysis:

LINES 1-36

General initialization. All system flags are set synthetically in lines 2-4. The I/O buffer required by the IL Development ROM is set up in lines 6-8. Lines 9-17 configure the GPIO's first control register (R0) to flag a service request whenever there's data in the buffer. Lines 18-21 configure the 41's IL chip to send out IDYs when idle (see Chapter 6 for how service request and IDYs work.), and finally lines 22-33 assign the global labels necessary for use.

LINES 37-61

Flow control always returns to LBL 20 and stops. Pressing R/S after a print has entered the fixer automatically continues execution, and sets up 2 interrupting alarms: one for 2 minutes 15 seconds in the future (line 42), the other for 7 (line 52). Other checking is done (at LBL 03) in case the current time is close to midnight and the prints may have to be moved tomorrow. (You can tell my usual darkroom hours!) Lines 60-61 increment Register 2, which keeps track of the number of prints in the fixer.

Step 6 Examine the print. If desired, use the system's features to tweak the exposure or change the filter with ANSI speed compensation.

LINES 62-75

LBL D (for Display) can be called from the keyboard as well as other subroutines. This simply displays the number of prints in each bath, as kept track of by regs. 02 and 03.

LINES 76-99

LBL C stands for Change filter, and calculates the new exposure value when changing polycontrast printing filters. The ANSI paper speeds have been stored at labels 10-18, which are moved into the X register at lines 84 and 99, "XEQ IND X". The old time is multiplied by the ratio of these 2 numbers and stored away into R04 in H.MS format, ready for action by hitting "AUTO". (/ key.) Calling LBL B displays the current time and filter number, so it may be recorded for future use.

LINES 100-112

Called by the alarm setting routine (LBL 20) when the alarm has to go off after 12:00 midnight. This subroutine advances the date and returns with the stack contents in the proper places.

Step 7 When finished, be proud of your results which were produced using a minimum number of steps!

LINES 113-132

LBL B calls LBL 04, and the two have been separated because calling routines need control passed back and finger execution (pressing the "B" button) does not. LBL 4 simply displays the current enlarger time stored in R04 and the current filter number stored in R00.

LINES 133-145

LBL A1 is the "warning" program called by the first alarm, telling the user that it's time to move a print from the fix to the wash. After decrementing R02 and incrementing R03 (reflecting the fact that a print is to be moved), the audible chirping routine "T1" is called 3 times, and then the fix-wash scoreboard is displayed (GTO D).

LINES 146-157

This is another warning routine called by the second alarm. R03 (the number of prints in the wash) is decremented and 3 TONE 6's are sounded, telling the user to remove the print from the wash and hang it up to dry. The new scoreboard is displayed (GTO D).

LINES 158-191

This is the MANual routine, called when the * key is pressed. It simply turns the enlarger on and measures the time elapsed until the * key is hit again. First, flag 2 is checked. If it is clear, then the enlarger is off and we branch to the 'on' routine, LBL 08. Otherwise, the enlarger is shut off (lines 164-165), the stopwatch time is stored in R04, the suspended alarms are re-activated (ALMNOW) and we return. LBL 08 starts off by synthetically suspending all alarms via lines 173-177 so there's no potential for interruption. The stopwatch is reset and started, the enlarger is turned on, flag 2 is set showing the MANual operation has started, and we GTO 20, the standard waiting spot.

LINES 192-210

LBL FOCus does the same job as MANual except no timing

takes place. First, flag 8 (indicating if this is the OFF cycle rather than the ON cycle) is checked; if it is we GTO 05 and shut it off, otherwise the enlarger is turned on. Lines 201-204 reset the IL Module for automatic IDYs (in case the 41 was turned off since the program was initialized). The statistical registers are cleared, the word "FOCUS" is displayed, and we wait.

LINES 211-218

In LBL 05, we try to determine what was done with the "RAT" while FOCus mode was on. First, the enlarger lamp is shut off, flag 8 is cleared, and then R10 is checked for how many A/D readings were taken. If R10=0 (no readings), we exit. Otherwise, we go on to LBL 22 for further processing.

LINES 219-250

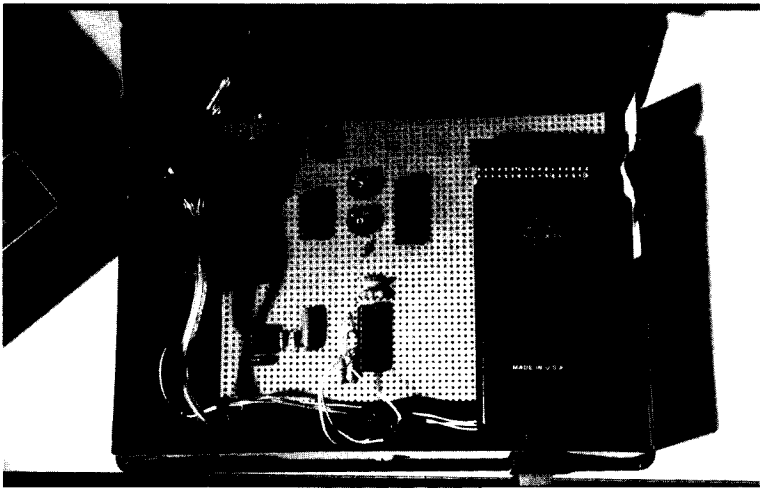
This is not a toggle function. AUTOmatic takes the time stored in R04 and turns the enlarger on for exactly that duration. First, flag 22 is checked to see if an overriding duration was manually punched into the X register. If it was, that time is stored into R4 and implemented. Either way, the pending alarms are synthetically suspended (lines 225-229) and the enlarger is turned on. Lines 235-248 then set an interrupting alarm that will turn the enlarger off. (Lines 239-240 subtract a small amount of time to compensate for the time it takes to set the alarm, thus guaranteeing a true-to-request duration.) Line 249 lastly restores all the suspended alarms, so audible reminders can still occur during a critically timed exposure.

LINES 251-258

This is the routine we branch to when the alarm set in the AUTO function comes due. It simply turns the enlarger off (254-255), clears flag 3 indicating AUTO mode has been terminated, and exits.

LINES 264-270

The ADJust function (the "-" key) takes the number of seconds in the X register, converts it into the proper format and adds it to the normal time stored in R04.



Inside the controller is an IL Converter and a small, unregulated power supply. A 5v regulator was added to protect the circuitry. The front panel (bottom) contains the IL receptacle, 3-pin miniature phone jack for the RAT, and a master power switch. The enlarger plugs into an AC receptacle mounted on top of the case (not shown).

LINES 271-318

Eight subroutines used when calculating new exposure times for filtration changes. Each label first stores its respective filter number in R00, drops the stack, and puts its effective ANSI paper speed into the X register and returns. LBL 10 represents the enlarger without a filter; LBLs 12-18 accommodate the seven filters, from #1 through #4 (in half-steps).

LINES 319-337

If at least one reading was taken with the RAT, we end up here. The exposure is determined by taking the average of all the readings and XEQing 31, which converts it to the number of seconds and stores it into R04. If only one reading was taken (lines 323-325) that's all we can do; otherwise the contrast range is computed by recalling the highest and the

lowest values read (stored in R11 and R12 respectively) and XEQing 32.

LINES 338-349

The required filter value is determined by taking the difference between the highest and lowest values read and XEQing 33, which is a look-up table. This subroutine changes the filter value in R00 and returns with the ANSI paper speed in the X register. Lines 341-348 then calculate the new exposure time required when using this new filter by taking the old time (R04) and multiplying it by the ratio of the old and new paper speeds.

LINES 350-375

LBL 33 is a look-up table which takes the range of RAT readings as input and compares it against the arbitrary thresholds defined earlier. If the number is within a given range it branches to one of the eight filter subroutines in lines 271-318.

LINES 376-388

This is an odd place for a "cleanup" subroutine, but that's precisely what LBL e does. Upon completion of a darkroom session, this routine gets rid of all the global labels and "FIXes" the display.

LINES 389-404

This is the interrupt routine jumped to every time the RAT button is pressed. First flag 08 is checked to see if the enlarger has already been turned on via the FOCus routine. If not, the enlarger is turned on and various registers are set up: the sigma registers are cleared (line 398), a high number is stored in R12, a low number is stored in R11, and flag 8 is set so next time the button is pressed a reading will be taken.

LINES 405-436

The number is read from the A/D Converter here. The GPIO is made a talker, a byte is transferred to the buffer and then to the X register, and the GPIO is immediately queried to see if more data awaits. If there is it was probably due to switch

bounce, so we disregard the old value and start again. (Lines 411-414.) We then check for "bad" values: if the number just read was either 128 or 255, it was probably an error and we branch to LBL 37 to handle it. By the time we get to line 424, the number is valid and a happy-sounding TONE 89 is issued for feedback. The remainder of the routine (lines 425-436) tests the number to see if it's either the lowest or highest value read so far, and then "sigma-plus"es it. The reading value is retained in the X register in case some other routine needs it.

LINES 437-443

The most complex error-handling routine in existence. Simply sounds a low negative-reinforcement tone, displays "TRY AGAIN", and exits.

LINES 444-462

This is the "chirp" sound used to give audible warning of a print that needs to be moved from the fixer to the wash. It is comprised of a string of TONE 89s, and is called three times from the "A1" interrupting alarm.

LINES 463-468

This subroutine will be different for every user. It consists of the equation that was derived in step 4, Determine the Exposure Curve.

Software Techniques

There are 2 significant techniques used in DKRM4 that warrant discussion. The first is the suspension of alarms in the ALMCAT, and the second deals with a no-overhead way of responding quickly to an event on the loop, such as someone pressing the button on the RAT.

Since the AUTO function is designed to reproduce a previous enlarger time EXACTLY, any alarms activating during an exposure run the risk of altering that time, since the 41 may be busy handling the interrupt rather than shutting off the enlarger.

To insure repeatable results, it is important to guarantee that no such interruptions occur.

One method for doing so is to fool the time module into thinking that there aren't any alarms. Using synthetics, it is possible to move the program/data divider, called a curtain, past the buffer area where alarms are normally kept, so the time module doesn't see them. Performing an ALMNOW in this condition tells the 41 "Look Ma! No alarms!" and makes it safe to move the curtain back without fear of the alarms activating. Other time-crucial activities can now be performed safely, and as soon as you are done another ALMNOW will force the 41 to examine the alarm buffer and make it realize that "Whoa! There are alarms here after all!" and they will start to activate in sequence. The first 5 lines of LBL 08 (line 172) shows the instructions required to do this. [For further details on this technique, refer to reference 2.]

Another thing I consider to be "really neat" about this application is that while the calculator's sitting idle and not running a program, I can grab the "RAT", press the button, and the 41 will suddenly start to execute a program! (Recall that in Chapter 1 it was pointed out that in order for the 41 to respond to any outside activity, a running program must continually look for the event. This has the side effect of slowing everything else down while at the same time increasing battery consumption significantly.)

To nullify this problem a neat little trick was used, and I shall now offer a quick demonstration of how it works. Take a 41, an IL Development ROM, and the 82162A IL Thermal printer and enter this initializing program:

```
01 LBL "SETUP"
02 55
03 FS? 55
04 XROM IF
05 SF 18
06 0
07 ENTER
08 64
09 WREG
10 3
11 ENTER
12 64
13 WREG
14 END
```

and this routine:

```
01 LBL "INTR"
02 BEEP
03 IDY
04 END
```

Now run the SETUP program, and press either the 'Print' or 'Paper Advance' keys and hear a BEEP. Congratulations! You've just reassigned the printer keys!

Well, actually the keys weren't reassigned at all; rather the microprocessor in the 41's IL Module was told to constantly search the loop for devices that "needed attention" (set the service request flag on a passing message frame). If one is found, and if the 41 isn't running a program, a program named "INTR" is run. This feature is mentioned (slightly) in the Development ROM's User Manual, Section 2 and Appendix C.

More Details

Using the WREG (Write Register) command in the DevIL ROM, it is possible to access certain registers of the HP-IL chip. In the above example, the chip was told to constantly send an IDY (Identify) message around the loop. If any loop device needs service (i.e., if a printer button is pressed), it alters the IDY message slightly and passes it on.

As you probably know, the IL Module performs its error checking by comparing the messages returning from the loop with those sent. Once the setup routine above has been run, every time a message comes back that is different from the one sent, the 41 runs a program called INTR (Interrupt), which will determine the cause of the interrupt and perform some specified function.

This means that INTR will start to run under the following conditions:

1) The 41 is in idle mode (CLOCK or programs not running) AND

2a) A transmission error has occurred (a frame came back not-as-sent), OR

2b) ANY device on the loop has requested service by altering the IDY message.

After determining the cause, INTR can then perform any routine or function, and then return back to idle mode once again. (Service Requests are covered more thoroughly in Chapter 6.)

In the general case, the interrupt handling routine should look more like this:

```
01 LBL "INTR"
02 FRNS?
03 RFRM
04 INSTAT
05 FS? 01
06 BEEP
07 FS? 02
08 TONE 7
09 IDY
10 END
```

Lines 2 and 3 check for the transmission error (Frame Received Not as Sent?) and reads the frame (RFRM) to clear the error. The rest of the program checks a status word from the printer to determine which button was pressed. That way, each button can perform a different function. The disadvantage here is the button must be held down to both generate an interrupt and be recognized through INSTAT.

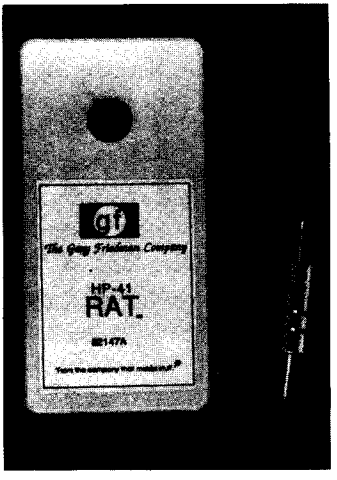
In the DKRM4 program, not much of this extra checking is performed since the loop configuration is known. As soon as an interrupt occurs, it simply turns the enlarger on (if it isn't on already), reads an A/D value from the IL Converter, and goes back to being idle.

The technique does have some undocumented restrictions, however. For starters, the 41 must be in AUTOIO mode and flag 44 (the continuous ON flag) must NOT be set. You must not be in PRGM mode when the interrupt occurs, and your program pointer must not be in ROM. When trying the above printer example make

sure flag 55 is clear, otherwise PRX will appear in the display rather than cause an interrupt.

Conclusion

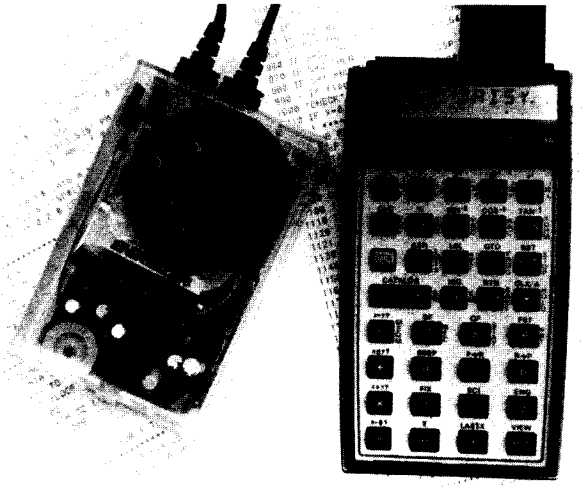
This darkroom application is only an example. This chapter is really about how a basic analog to digital converter works and how to tailor it to your needs. It also serves to give an excellent philosophical example of what computers ought to be used for: relieving users of the 'dog work' and leaving them free to concentrate on the creative matters.



References

"Quality Enlarging with Kodak B&W Papers", Eastman Kodak Co. 1982

"Suspending Alarms" Tapani Tarvainen, PPC Journal, August 1983



Chapter Five

SPEECH SYNTHESIS

"The HP-IL system exemplifies the phrase 'What Goes Around, Comes Around'".

--Clifford Stern

I've always been a fan of synthetic speech, so when the opportunity came a few years ago to actually make a portable system, I jumped at the chance and built what was at the time the smallest speech development system known. This chapter describes that system, discusses some unusual interfacing requirements, and gives several examples of programming (and even some singing!) using a single speech synthesis I.C.

Most people are probably familiar with computers that can talk, but few bother to make the distinction between two very different approaches to computer speech: digitized and synthesized. Digitized speech works much like a tape recorder. It starts with a human speaking into a microphone, but instead of being stored on magnetic media, it is digitized, analyzed, homogenized, mathematically compressed and stored in computer memory. The inverse of this process is employed to retrieve the information and "play it back". Because it starts with a human source, digitized speech is famous for sounding very much like a human. The price paid for this, however, is the large amount of data needed to represent the voice, its qualities, and inflections. Data compression can help reduce that amount by extracting only the "characteristic features" and compressing that data to fit into a small space, but the voice quality becomes heavily degraded as a result.

The other method, synthetic speech, doesn't start with a human voice at all. Using Fourier analysis, all the different sounds of the English language are recorded, reduced to mathematical equations, and simulated by digital filter techniques. The result is that any combination of English language sounds can be strung together to make words and phrases from relatively little data, giving the unit an unlimited vocabulary without the usual large amount of memory! Synthetic speech, unfortunately, also has its drawbacks: it very often talks with a very heavy Swedish-sounding accent.

The Votrax Co., a division of Federal Screw Works (I'm not kidding!) has been one of the early pioneers in synthetic speech for personal computers as well as mainframes. Their offering is the SC-01 chip, a primitive sounding (by today's standards, anyway) phoneme-based chip that represents a mathematical model of a human vocal tract.

Votrax has broken the English language up into 64 different sounds, or phonemes, all of which are illustrated in Fig. 5-1. These phonemes consist of vowels, consonants, blends, combinations, and "no-sounds". You can have the speech chip say any one of these phonemes by putting its respective address onto the chip's address bus and grounding the chip's STB (Strobe) line momentarily. If you took a whole bunch of these phonemes and fed them in one at a time very quickly, you might hear something that sounds like a word. For example, to say "Thank you", you should feed the chip the following sequence of addresses:

PHONEME SYMBOL:	TH	A1	Y	N	K	-	Y	U	W	-
PHONEME CODE (ADDRESS):	57	6	41	13	25	3	41	40	45	3

It would seem a natural that, given this chip's 8-bit bus interface, both the 41 and 71 ought to be talking.

ADDRESS	SYMBOL	DURATION (ms)	SAMPLE WORD
00	EH3	59	jacket
01	EH2	71	enlist
02	EH1	121	heavy
03	PA0	47	(--)
04	DT	47	butter
05	A2	71	made
06	A1	103	made
07	ZH	90	azure
08	AH2	71	honest
09	I3	55	inhibit
10	I2	80	inhibit
11	I1	121	inhibit
12	M	103	mat
13	N	80	sun
14	B	71	bag
15	V	71	van
16	CH(*)	71	chip
17	SH	121	shop
18	Z	71	zoo
19	AW1	146	lawful
20	NG	121	thing
21	AH1	146	father
22	OO1	103	looking
23	OO	185	book
24	L	103	land
25	K	80	trick
26	J(*)	47	judge
27	H	71	hello
28	G	71	get
29	F	103	fast
30	D	55	paid
31	S	90	pass

ADDRESS	SYMBOL	DURATION (ms)	SAMPLE WORD
32	A	185	day
33	AY	65	day
34	Y1	80	yard
35	UH3	47	mission
36	AH	250	mop
37	P	103	past
38	O	185	cold
39	I	185	pin
40	U	185	move
41	Y	103	any
42	T	71	tap
43	R	90	red
44	E	105	meet
45	W	80	win
46	AE	185	dad
47	AE1	103	after
48	AW2	90	salty
49	UH2	71	about
50	UH1	103	uncle
51	UH	185	cup
52	O2	80	for
53	O1	121	aboard
54	IU	59	
55	U1	90	you
56	THV	80	the
57	TH	71	thin
58	ER	146	bird
59	EH	185	get
60	E1	121	be
61	AW	250	call
62	PA1	185	(--)
63	STOP	47	(--)

* "T" must precede "CH" to produce CH sound.
* "D" must precede "J" to produce J sound.

Figure 5-1
Phoneme Chart

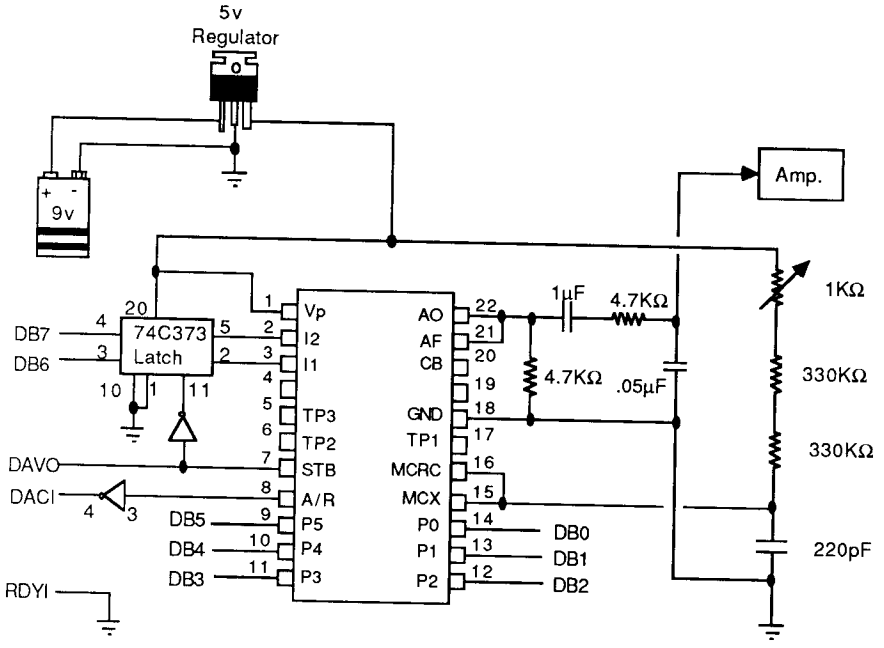


Figure 5-2
Diagram of
Speech Synthesizer
and Low-Pass Filter

The Circuit

Figure 5-2 shows a method of hooking up the speech chip to the IL Converter. Since this chip was designed to interface with a microprocessor bus, no separate latch is needed. The 74C373 8-bit latch pictured is used instead to facilitate inflection, which we will cover later. There is an amplifier attached to the SC-01's 2 output pins; this can consist of a tape recorder with an Auxiliary input and Monitor mode. Alternatively, there also exist several self-contained low-fidelity modules available from local electronics stores; one such module has been included in the figure along with a low-pass filter to increase the sound quality at moderate-to-high volumes.

The values of the resistors and capacitors determine the

reference frequency needed to generate intelligible speech, and the valid passband is very narrow. (Putting your finger across the 220 pF capacitor, for example, is enough to slow it down to the point where it starts doing its H.A.L. 9000 impersonation.) These values are not crucial; the only sensitive ones are those that are attached to pins 15 and 16. In general, the values for these parts should be chosen so that

$$\frac{1}{R_{total} \times C} = 7{,}000$$

where R is the resistance in Ohms and C is the capacitance in Farads. Most people hooking up a speech chip for the first time have a great deal of trouble just getting it to say "Ahhhh", because finding the correct resistor/capacitor combination to fulfill its narrow tolerances can be difficult without pulse counters, capacitance meters, etc. Don't worry; use variable resistors and try lots of similarly-rated capacitors (they tend to deviate from the marked value the most) and it shouldn't take more than a day to hear a sound.

I decided that futuristic packaging is the way to go, so I took two clear audio cassette cases, glued them back to back, and had enough room inside to fit all the necessary components: an IL Converter (minus its case), amplifier, speaker, 9v battery (and therefore 5v regulator to get the voltage down to a non-destructive level), speech chip and its interface ICs.

How to Drive it with Software

So how do we get this thing to talk? Let's try it with the 41:

```
    DO                          COMMENTS

    MANIO           We're not talking to a printer.
  5 ACCHR           Hear "A"
 60 ACCHR           Hear "E"
  3 ACCHR           No sound.
```

As a second example, fill the alpha register with this string (refer to Chapter 1 if you forgot how), SF 17, and OUTA:

```
H    EH1    L    O    W    -                    "HELLO"
27    2     24   38   45   3
```

Notice that the last phoneme was a "stop", which keeps the speech chip from making the "W" sound forever. Some other interesting phrases it can say are:

"Hewlett Packard":

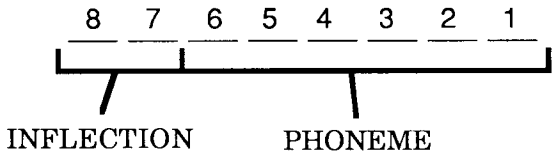
```
H    Y    U1    L    EH1    T    -
27   41   55    24    2     42   3

P    AE    K    ER    D    -
37   46    25   58    30   3
```

"Shall we play a game?"

```
SH    AE    L    -    W    E1    Y    -
17    46    24   3    45   60    41   3

P    L    AE    Y    -    A    Y    -
37   24   46    41   3    32   41   3

G    A1    AY    Y    M    -
28    6    33    41   12   3
```

Adding Inflection

This all sounds hopelessly monotone, making this an ideal time to introduce the inflection concept. The SC-01 chip was designed to operate on 8-bit machines, allowing a byte to be broken down as follows:



The two most significant bits determine the pitch, or inflection, at which the phoneme is spoken. The greater the inflection bits, the higher the pitch. Therefore, if you wish to increase the pitch of any phoneme, simply add multiples of 64 to the phoneme's address. For example, to hear it sing "A" in 4 musical keys:

```
                DO                          COMMENTS

                              MANIO
                    5         ACCHR         Inflection level =0
      5 + 64 = 69             ACCHR         Inflection level =1
     69 + 64 =133             ACCHR         Inflection level =2
    133 + 64 =197             ACCHR         Inflection level =3
                    3         ACCHR         Stop
```

Adding inflection levels to your speech can make it much more intelligible. For example, we can make "Shall we play a game?" sound more like a question:

```
 SH   AE   L   -   W   E1   Y   -
 17   46   24  3   45  60   41  3

 P   L   AE   Y   -   A   Y   -
101  88  110  105 67  32  41  3

 G   A1   AY   Y   M   -
 28   6   97  105  76 131
```

What a difference! Notice that inflection was added to the STOP

phoneme as well. Although STOP doesn't produce any sound, its inflection will determine how the previous phoneme will end. In this example, it is responsible for the question-like quality at the end of game?

Here are some other examples:

"Go ahead...Make my day!"

```
G    OO1   O1    -    UH1   H    EH1   EH3   D    -
28   22    53    3    50    27   2     64    94   3

M    A1    AY    Y    K    -    M    AH1   EH3   Y    -
76   70    97    105  89   67   76   85    64    105  67

D    A1    I3    Y    -
94   70    137   105  3
```

"Happy Birthday to you....":

```
H    AE1   EH3   P    Y    -
91   111   64    101  105  67

B    ER    R     TH   D    A1   I3   Y    -
78   186   107   121  94   6    9    41   3

       T    U1    U1    U1    U1    --    Y    U1    U1    U1    U1    -
(1st Time)
      170   183   183   183   183   131   105  119   119   119   119   67
(2nd Time)
      234   247   247   247   247   195   169  183   183   183   183   131
```

"Starbase Operations":

```
S    T    AH1   R    B    A2   AY   S    -
95   106  85    107  78   69   97   95   67

AH1   P    ER    A2    AY    SH    UH3   N    S    -
85    101  122   69    97    81    35    13   31   3
```

The latter phrase, from the beginning of a two-part Star Trek episode, sounds best when programmed into a loop and played back while adjusting the potentiometer in Fig. 5-2 to extremes.

10 Point Mystery Phrase

76, 85, 105, 67, 229, 235, 230, 92, 107, 175, 140, 122, 67.
170, 176, 176, 170, 131, 76, 108, 67, 106, 119, 119, 67.
159, 138, 138, 141, 156, 131, 96, 97, 67, 223, 176, 176, 141, 156, 131.
94, 96, 33, 15, 3, PSE
109, 86, 94, 67, 34, 55, 55, 3, 88, 72, 105, 89, 67.
42, 55, 55, 3, 91, 108, 107, 67, 39, 170, 131. PSE
180, 180, 183, 131, 25, 32, 33, 3. PSE
222, 224, 224, 224, 224, 225, 210, 172, 172, 172, 131.
94, 96, 96, 96, 96, 97, 82, 44, 44, 44, 3.
92, 75, 73, 79, 67, 140, 188, 169, 131, 233, 244, 244, 235, 195.
111, 111, 111, 64, 77, 95, 186, 131, 30, 40, 40, 40, 3.

NOTE: For the most accurate reproduction, one should hold two fingers across the 220pf capacitor's leads while the above mystery phrase is being played.

A moment to reflect. The 41, to begin with, is slow. Not helping the situation any is the fact that the SC-01 chip uses only one line for handshaking, while the GPIO likes to use two. If this leftover handshake line (RDYI = Ready Data In) is tied to ground instead, data flow to the chip will occur but with the following drawback: the GPIO's 32 register transfer buffer is disabled; so after an OUTA is executed, program control will not be returned until the speech chip says the last phoneme.

Because of this, when long speeches (such as the 10 point Mystery Phrase) are spoken, the alpha register must load and output groups of 24 phonemes quickly if noticeable gaps in speech are not to occur. This necessitates the use of synthetic text lines on the 41 for speedy alpha filling. (A doubled clock speed on said machine also helps considerably.) The only problem with this method is that EH3 (address = 0) will not be sent with an OUTA

because it is interpreted as a null. EH1 or EH2 should suffice as a substitute.

Unfortunately, these speed problems also hold true for the 71: Since the buffer is bypassed, program execution will not continue until the entire phrase is spoken. Slowness on this machine, though, should be much less noticeable.

Sending text strings on the 71 is a much easier task and can be done in this fashion:

```
10 A = DEVADDR("%64")
20 SEND UNT UNL MTA LISTEN A
30 SEND DATA 91,111,64,101,105,67,78,186,107,121,94,6,9,41,3
40 SEND DATA 170,183,183,183,183,131,105,119,119,119,119,67
50 SEND DATA 91,111,64,101,105,67,78,186,107,121,94,6,9,41,3
60 SEND DATA 234,247,247,247,247,195,169,183,183,183,183,131
70 SEND UNT UNL
```

This short program has it sing "Happy Birthday". The equivalent program on the 41 looks like this:

```
01*LBL "HAPPY"
02 SF 17
03 "[o@eiCN+ky^++)+"
04 OUTA
05 "++++++++iwwwwC"
06 OUTA
07 5
08 LN
09 E^X
10 LN
11 E^X
12 "[o@eiCN+ky^++)+"
13 OUTA
14 "+++++++++++++"
15 OUTA
16 PSE
17 GTO "HAPPY"
18 END
```

(The bytes for the four synthetic text lines can be found in lines 30-60 of the 71 program above.) Barcode for "HAPPY" and other phrases appears on page 282.

For those of you who prefer thinking software, the program on the next page is designed to recite the correct time. It requires a time module and an XF/M module with the following ASCII file named "TIME" loaded into it, which contains the necessary vocabulary. First, the program (Barcode is provided on page 284):

```
01*LBL "TIMED"
02 CF 21
03 TIME
04 FIX 2
05 "  "
06 ATIME
07 AVIEW
08 STO 01
09*LBL 04
10 RCL 01
11 INT
12 12
13 X<=Y?
14 SF 05
15 X<>Y
16 X>Y?
17 -
18 X=0?
19 12
20 "TIME"
21 SEEKPTA
22 GETREC
23 RCL 01
24 FRC
25 10
26 *
27 ENTER^
28 INT
29 X=0?
30 GTO 01
31 2
32 X<=Y?
33 GTO 03
34 RCL Z
35 10
36 *
37 INT
38 SEEKPT
39 ARCLREC
40*LBL 02
41 SF 17
42 OUTA
43 FS? 05
44 "elC++++"
45 FC?C 05
46 "`aCAA++"
47 OUTA
48 DATE
49 DOW
50 FS?C 08
51 XEQ 05
52 24
53 +
54 SEEKPT
55 GETREC
56 SF 17
57 OUTA
58 RTN
59*LBL 01
60 RDN
61 10
62 *
63 INT
64 X=0?
65 GTO 02
66 0
67 SEEKPT
68 ARCLREC
69 X<>Y
70 SEEKPT
71 ARCLREC
72 GTO 02
73*LBL 03
74 RDN
75 18
76 +
77 SEEKPT
78 ARCLREC
79 RDN
80 FRC
81 10
82 *
83 INT
84 X=0?
85 GTO 02
86 SEEKPT
87 ARCLREC
88 GTO 02
89*LBL 05
90 RCL 01
91 1 E2
92 *
93 FRC
94 10
95 *
96 END
```

Synthetic Text Lines:

```
44: 101,108,67,129,129,12,3
46: 96,97,67,65,65,12,3
```

ASCII FILE "TIME"

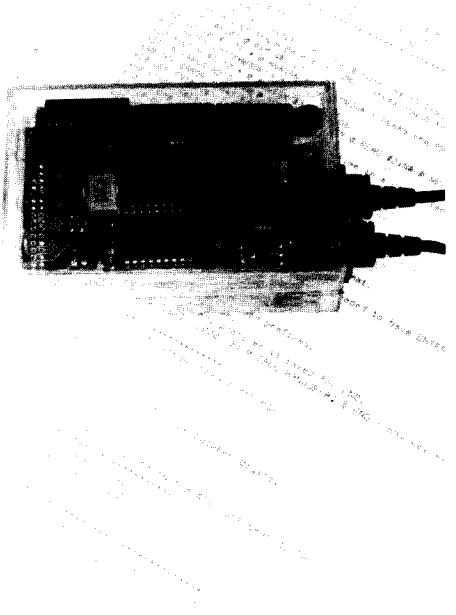
REC. #	SAYING	ASCII DATA
0	ZERO	18,44,43,53,55,3.
1	ONE	173,178,13,13,3.
2	TWO	170,168,45,3.
3	THREE	185,171,44,41,3.
4	FOUR	157,180,180,43,3.
5	FIVE	29,21,0,9,41,15,3.
6	SIX	31,39,25,31,3.
7	SEVEN	159,130,143,1,13,3.
8	EIGHT	32,33,42,3.
9	NINE	13,21,41,13,3.
10	TEN	106,65,1,13,3.
11	ELEVEN	44,24,66,79,0,2,13,67.
12	TWELVE	42,45,2,24,15,67.
13	THIRTEEN	121,122,106,172,13,3.
14	FOURTEEN	157,117,107,106,172,13,3.
15	FIFTEEN	93,103,93,170,236,13,3.
16	SIXTEEN	95,103,89,95,106,172,13,3.
17	SEVENTEEN	95,66,79,65,77,106,172,13,3.
18	EIGHTEEN	96,97,106,44,13,3.
19	NINETEEN	77,85,105,77,106,44,13,3.
20	TWENTY	106,109,66,77,106,108,41,3.
21	THIRTY	121,122,106,44,41,3.
22	FORTY	157,180,180,43,42,44,41,3.
23	FIFTY	29,39,29,42,44,41,3.
24	SUNDAY	223,242,227,205,205,158,32,33,3.
25	MONDAY	204,242,227,205,205,158,32,33,3.
26	TUESDAY	170,169,168,146,94,32,33,3.
27	WEDNESDAY	37,194,222,205,223,158,32,33,3.
28	THURSDAY	185,186,171,146,94,32,33,3.
29	FRIDAY	221,235,213,233,94,32,33,3.
30	SATURDAY	223,238,170,58,94,32,33,3.

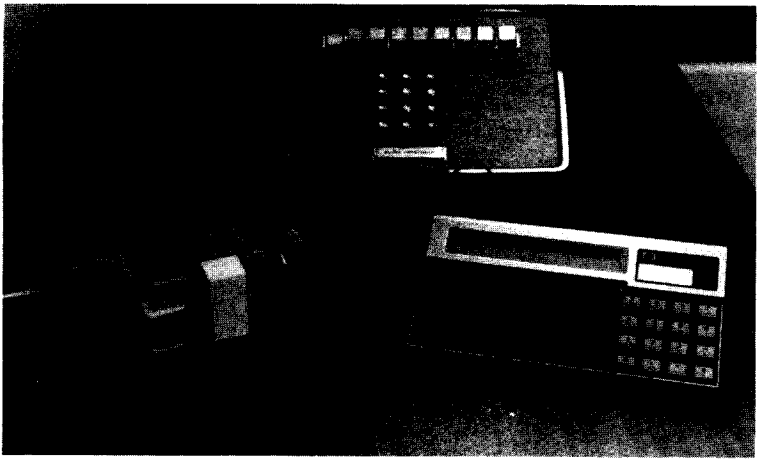
$$\frac{210 \text{ Chars.} + 31 \text{ Regs.} + 1}{7} = 34.5 = 35 \text{ Registers}$$

There's only one way to input the "TIME" file: take each of the above numbers, toss it into ALPHA via XTOA, and at the end of each line INSREC.

Programs like this can be included in verbal alarm clocks, which also capitalize on the 41's time module capabilities. And since the whole setup is programmable, it's not too much trouble to implement an oversleepers anonymous mode, where the unit actually gets abusive if the 'snooze' function is called upon too many times.

In a couple of chapters, we'll use the SC-01 speech chip to greet your callers in a most sophisticated telephone answering machine. But that shouldn't stop anyone from coming up with other practical applications (such as indestructible Trans-Ams with 4-character acronyms).





Chapter Six

INTELLIGENT AUTODIALER

"Computers are not intelligent. They only think they are."
-Anonymous

This chapter and the next may seem to deal with fun things one can do with telephones, but actually they are a sample of all the different things that can be done using the few techniques introduced in previous chapters.

The intelligent autodialer is a device I originally designed out of my own frustration with our office's telephone system. Dialing three digits just to get an outside line, then another ten digits for the number (plus waiting for the Dimension™ system to redial the number for you) only to find a busy signal got to be very annoying. This system would always listen in on the line and would know what number I dialed on the telephone's keypad. If it was busy, I'd just press a button and the 71 would keep trying the last number it heard until it broke through. Soon, using exactly the same hardware, the system grew to include a Rolodex-type phone directory (where you search for the name and it dials the number), and a phone usage monitor (where the system keeps a log of all outgoing calls and their dates and times).

The most useful and unique aspect of this system is its ability to detect busy signals and automatically redial the number without human intervention, normally a difficult thing because of the phone companies' varying representation frequencies and signal strengths. Among those who will <u>really</u> appreciate this feature are teenagers who participate in radio station contests that award a billion dollar prize to the 13th caller!

First, every individual aspect of the system will be described in detail, so you can get an understanding of how things interact and will be able to design systems that meet your own needs, should you desire to do so. Next, I'll describe the system software and complete schematics. Finally Chapter 7, using almost identical techniques, will prove once and for all that the 41 is just as capable a system driver as its horizontal successor (the 71) by making it do more things than any calculator should be allowed to do by law.

Telephone Line Interface

This is a sticky topic, but must be addressed since two chapters rely on it. Before 1955, it was illegal for anyone to attach anything besides phone company equipment to the phone lines. Two Supreme Court decisions, the "Hush-A-Phone" case in 1955 and the Carterfone case in 1968 altered the laws to allow foreign equipment to be attached to the phone company's lines via an expensive interface called a coupler. The coupler's stated purpose was to provide insurance against someone's Chen Fu brand answering machine vaporizing central office equipment, but the phone company required it even for machines that were FCC approved.

Couplers were phased out in the late '70s (as were their ridiculous surcharge) when the FCC rules were altered in 1975 to allow direct connection of any FCC-approved device to the phone lines. Today, in the post-AT+T breakup era, it's suddenly OK to attach a $10.00 disposable phone without a charge of any kind! (It must be FCC approved, of course.)

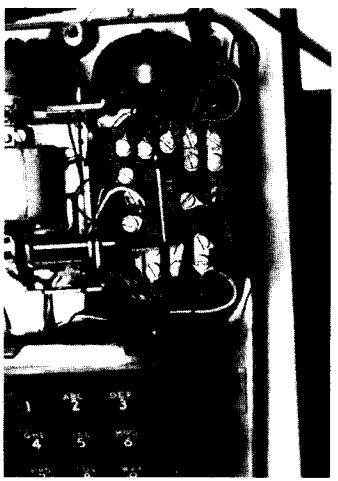
It is still mandatory, however, that any equipment attached to the phone lines meet these said requirements. Since legitimate telephone-line interfaces aren't a commonly available off-the-shelf component, I solved the problem by taking an old telephone and removing its hybrid transformer, which looks like a rectangular cube with a wiring block on top. This transformer acts as the telephone's interface; it takes the two incoming wires from the central office and splits them into four needed for the mouthpiece and the earpiece. (See Fig. 6-1.) Best of all, every telephone has

this component, so a little bit of scavenging at surplus shops or garage sales will yield you a perfectly legal and efficient telephone interface!

Generally, using the "R" contact as a common ground, telephone audio can be heard through "B", and sounds fed to the phone line should go through "GN". The terminals normally used for the dial contacts, "F" and "RR", will answer the phone if shorted out and hang up if not connected. An opto-isolator attached to these two contacts makes that task easy.

Hardware Overview

Just to get familiar with this circuitry, here's an overview of the components and how they work together. Fig. 6-2 (next page) shows the entire circuit, which provides all the input necessary for the software to make its decisions. Notice that this is the first time



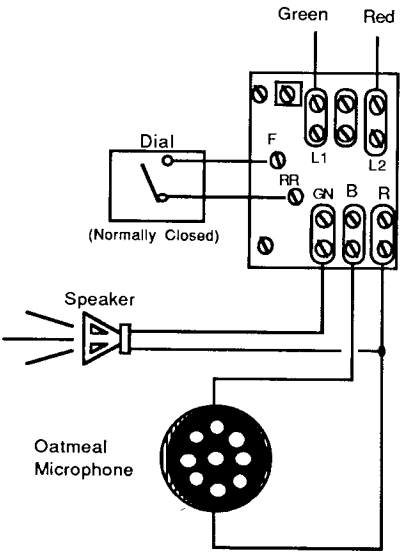


Figure 6-1
Telephone
Hybrid
Transformer

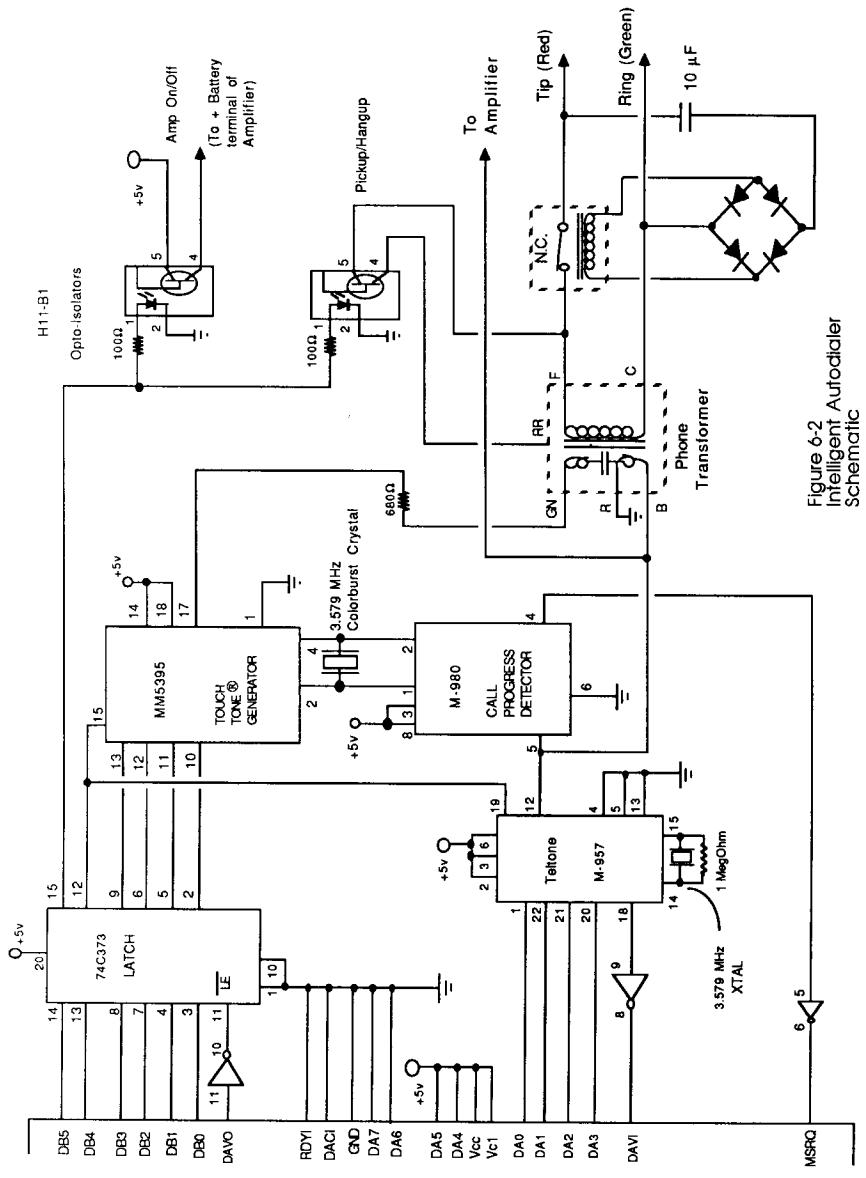


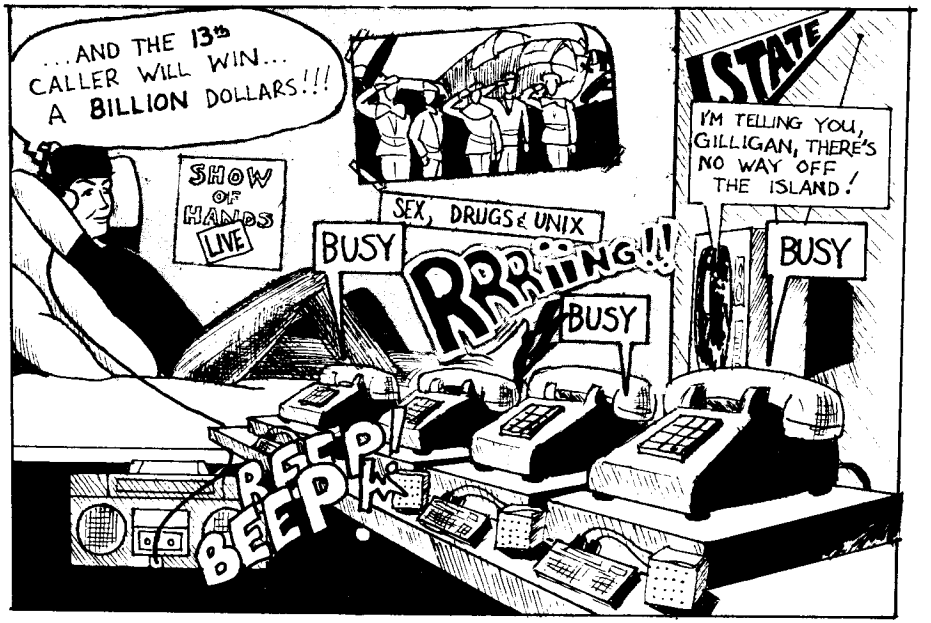
Figure 6-2
Intelligent Autodialer
Schematic

we take advantage of the GPIO's 8-bit uni-directional mode: output goes to Data Bus B, and input comes from Data Bus A.

The Touch Tone dialer portion should look familiar; it's the same circuit used in Chapter 2, but without the elaborate handshaking hardware. (More on that later.) Below the Touch Tone generator we add another chip which performs the inverse function; it takes the sounds, decodes them, and delivers a binary code representing the digit pressed. This chip is needed for the LND (Last Number Dialed) feature, so the computer can mimic the last number manually dialed on a normal phone.

A chip that detects sound on the phone line is also needed so the host computer can time the duration of the noise and tell whether its ringing, busy, or dead. This gets its input from the same place as the Touch Tone decoder chip above it, but delivers its output not to the data bus, but to the MSRQ (Manual Service Request) line of the GPIO.

Finally, the two Darlington-pair opto-isolators answer and hang up the phone, and turn on an optional audio amplifier so the



user can monitor the progress of the call along with the HP71. (This is more of an anti-paranoia device to help the user learn to trust in the 71's decision-making process. After all, an expert system's advice is of little value if the user doesn't trust it!)

Touch Tone Decoding

Recall that in Chapter 2 we discussed how to hook up the MM5395 Touch Tone chip to generate the frequencies needed for automatic telephone dialing. Here we use the chip again, except the special timer circuitry to control the handshake lines isn't needed. Instead, we take advantage of the finer control offered by the 71, and regulate the Touch Tone output rate via software rather than hardware.

Back in the olden days, decoding these Touch Tone signals required seven or eight temperature-sensitive phase-locked loop passband filters all critically adjusted via RC constants. Now, there's a single IC, the M-957 by Teltone, which only requires a standard 3.579 MHz crystal and a resistor to do the identical job, making Touch Tone decoding even easier than learning RPN.

Figure 6-3 shows how this neat little component is attached to the hybrid transformer. Normally, this chip will be monitoring all

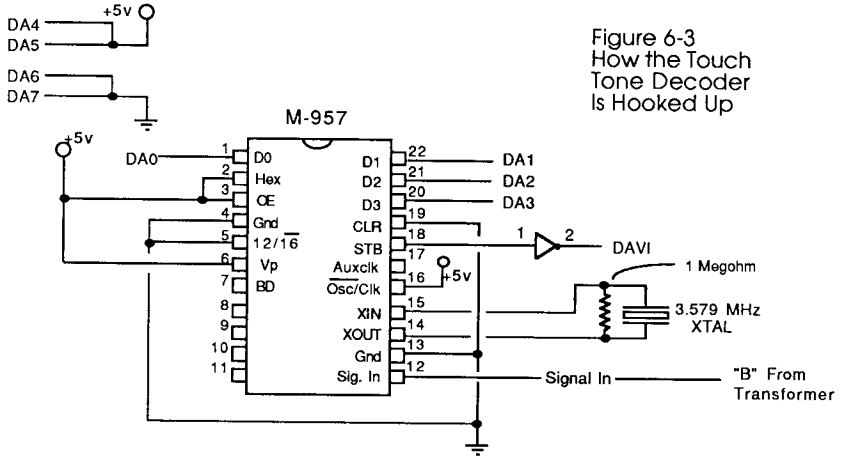


Figure 6-3
How the Touch
Tone Decoder
Is Hooked Up

audio activity on the phone line. If it hears any one of the 16 valid Touch Tone codes, it produces a 4-bit output on its data pins and pulses its strobe output high to tell anything that's listening that its data pins have a valid output.

Notice the remaining four data lines: half are tied to positive, and half are grounded. This is due to a discrepancy between data representations: the M-957 chip gives its output in binary form, while the 41 and 71 computers prefer to get their information in ASCII coded format. (See below for table of discrepancies.) Notice that for the digits 1 through 9, the only difference between the binary and ASCII representations is that bits 4 and 5 in the ASCII words are set to "1". Zero through D aren't as convenient; the ASCII characters don't correspond to the digit entered. This is especially troubling with the zero, which MUST be available as a number. The simplest solution to this problem is to simply tie bits 4 and 5 of the data bus high, allowing digits 1-9 to appear to the computer as their corresponding ASCII codes, and instruct the computer to replace any colons (:) in the incoming numeric string with a zero. Similar correction for *, #, A, B, C, and D are also necessary.

INPUT	LOW FREQ	HIGH FREQ	HEX FORMAT	ASCII FORMAT	ASCII CHAR.
1	697	1209	0000 0001	0011 0001	1
2	697	1336	0000 0010	0011 0010	2
3	697	1477	0000 0011	0011 0011	3
4	770	1209	0000 0100	0011 0100	4
5	770	1336	0000 0101	0011 0101	5
6	770	1477	0000 0110	0011 0110	6
7	852	1209	0000 0111	0011 0111	7
8	852	1336	0000 1000	0011 1000	8
9	852	1477	0000 1001	0011 1001	9
0	941	1336	0000 1010	0011 1010	:
*	941	1209	0000 1011	0011 1011	,
#	941	1477	0000 1100	0011 1100	<
A	697	1633	0000 1101	0011 1101	=
B	770	1633	0000 1110	0011 1110	>
C	852	1633	0000 1111	0011 1111	?
D	852	1633	0000 0000	0011 0000	0

Embedded Control Bits

One aspect of the generator chip should be mentioned at this time. In order to achieve the features described in the first paragraph, the circuitry must somehow determine whether the outgoing data is meant for the touch-tone generator or the opto-isolators, which control the hookswitch status and the monitor amplifier. We solve this problem by noting once again the Touch Tone discrepancy table. This table also represents the way the 71 sends out ASCII digits; the lower four bits represent the binary digit transmitted, and bits 4 and 5 are always set to "1", and never get routed to the dialing chip. These bits do not go to waste; they are used to control two functions.

Bit 4 is used as a "chip enable" input, hooked up so the chip will only generate Touch Tones when this line is set to "1". This is needed so we can tell it to stop transmitting; just output a word with this bit set to "0" and it shuts up. (Normally, sending it a "0" sends an ASCII 0 which has bits 4 and 5 set, and therefore dials a "D", the 16th button.) Bit 5 is tied to two opto-isolators. When this bit is set it picks up the phone (opto-isolator #1, which is connected to the hybrid transformer), and turns on an optional audio amplifier so the user can hear what's going on. (Opto-Isolator #2.) The sample program below shows how this technique is implemented.

```
10 ! This program picks up the phone, dials the
20 ! telephone number appearing in T$, and hangs up.
25 ! Really rude, huh?
30 ENDLINE""                  ! Eliminates automatic
                              ! carriage return/line feed
                              ! normally appended to
                              ! output.
40 A=DEVADDR("GPIO")          ! Find the GPIO's address
                              ! on the loop.
50 SEND MTA LISTEN A          ! Makes the GPIO a
                              ! listener; the 71 a talker.
60 OUTPUT :LOOP; CHR$(32)     ! Sets bit 5 high, picking
                              ! up the phone.
```

```
70 FOR X=1 TO LEN(T$)          ! Loop as many times as we
                               ! have digits to dial.
80 OUTPUT :LOOP;T$[X,X]        ! Send only 1 ASCII
                               ! character.
90 WAIT .02                    ! Let the signal stay for
                               ! (at least) .02 seconds.
100 SEND DATA 32               ! Turns only bit 5 on.
                               ! Keeps phone off-hook;
                               ! tells Touch Tone chip to
                               ! shut up.
110 WAIT .02                   ! Keep silent for at least
                               ! .02 seconds.
120 NEXT X                     ! Dial next digit.
130 OUTPUT :LOOP; CHR$(0)      ! Shuts everything off.
                               ! (SEND DATA 0 would have
                               ! worked, too.)
140 SEND UNT UNL               ! Cancels current talker and
                               ! listener status.
150 END                        ! Gratuitous exit.
```

Notice that throughout the above program, bit 5 was always set during the dialing process, but bit 4 (acting as "chip enable" for the dialing chip) was set only when we were dialing a digit. We use this property to solve a small but potentially disastrous situation: having the Touch Tone decoder do its thing at the same instant we're dialing numbers. If this were allowed to happen, the sound generated by the dialer chip would instantly be decoded and fed back to the GPIO a few milliseconds later. Because of a design limitation of all three 8-bit ports, data coming and going at the same time will "lock up" the port, effectively disabling it.

This problem is easily solved by routing bit 4, normally used for the generator's Chip Enable input, to the CLEAR input of the M-957 decoder chip as well, setting it to its DISABLE state whenever we're dialing digits. This connection appears in the full circuit schematic shown earlier.

Call Progress Detector

Having a machine detect whether a line is busy or not is much more difficult than one might initially think. Frequencies used to signal ringing and/or busy are non-standard; and their volume is certainly nothing to count on. Only one property remains consistent throughout all American switching systems: the rhythm.

Figure 6-4 shows the possible signals that we must recognize.

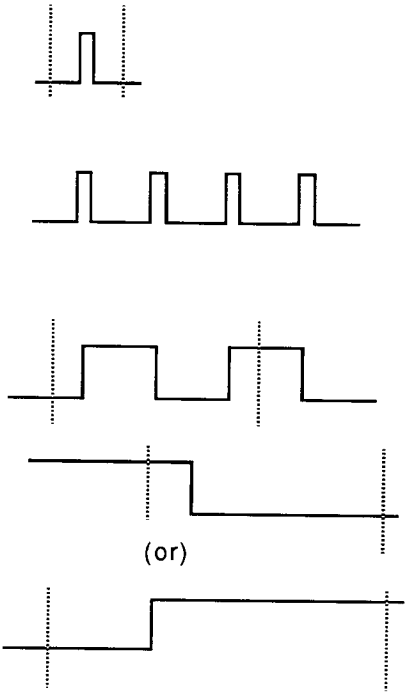
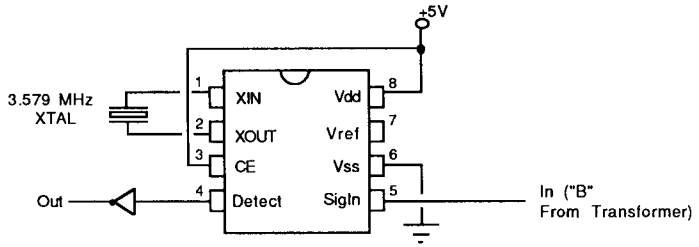


Figure 6-4
Possible Sounds



M-980
Call Progress Tone Detector

The dotted lines show the minimum signal characteristic needed to determine what it is. For example, it is not necessary to listen to the entire ring to determine that it is a ring; if we first hear silence and then a sound for more than a certain amount of time, then we conclude that it must be a ring.

Given this binary time-domain input, it is up to the controlling computer to time the duration of the silence/no silence periods and draw a conclusion as to the status of the connection.

To detect whether the line is busy or not, we therefore need an additional component that will tell a controlling computer what's happening on the line. Teltone, the same company that makes the M-957 DTMF (Dual Tone Multi-Frequency) decoder chip described above, also makes the M-980 Call Progress Tone Decoder chip, as shown above. Don't be fooled by its glamorous name; this chip is little more than a bandpass filter, and will output a "1" if it hears a sound and a "0" if it doesn't. This output is attached to the GPIO's MSRQ (Manual Service Request) line, and it is up to the controlling computer to retrieve this line's status by asking the GPIO for a STATUS byte. The computer must be very fast (the 71 barely makes it using BASIC, and FORTH is slower for any IL activity) in order to have a high enough sampling rate to separate rings, busies, and random noise. Here's where careful programming saves the day.

The diagram in Fig. 6-5 (next page) resembles a state table, and it represents an algorithm which allows you to recognize incoming patterns in a stream of incoming data. All of the algorithms for HP-IL are specified using a more complex version of state tables

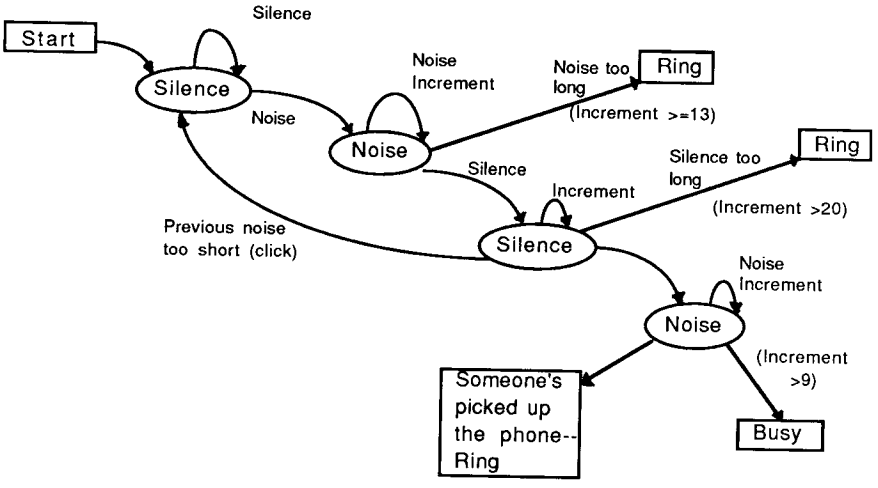


Figure 6-5
Algorithm for
Determining a
Busy Signal in the
Shortest Possible
Time

because they describe the process much more simply than an equivalent flowchart would. In this application it will, in the shortest time possible, measure the duration of the sounds and silences and determine which of the six possible patterns it is hearing.

The subroutine works like this: as soon as the number is dialed, the program sits and waits for a sound to occur. When that happens, we branch to the circle labeled "noise" and we take note of the noise's duration. If it exceeds a certain threshold, then it must be a ring so we exit. If not, then it could be either the tail end of a ring or a busy pulse, and we branch down to the next circle, where the duration of the silence is measured. At this point, a long period of silence indicates a ring; a short period must be a busy. Using this algorithm, the 71 can determine the status of a connection in as little as half a cycle!

Anyone with any experience with HP-IL is probably wondering whether the 71 can take sound samples fast enough to do this job adequately. Well, the answer is "yes" if you know how to utilize HP-IL's low-level features.

The only way to determine the status of the MSRQ line is to request a status word from the GPIO, and then check to see if bit #5 is set. (The GPIO's owner's manual has more details about interpreting the STATUS word.) But the 71's SPOLL("GPIO") (Status POLL) function, the only means of requesting this information, insists on sending out this lengthy HP-IL message sequence every time the function is called:

```
UNL       Unlisten.  Disable all listeners.
TAD 01    Find out what's on the loop:  Make the
SAI       first device a Talker and Send Accessory
          ID.
DAB 35    1st device responds with Data Byte 35;
          it's a printer.
TAD 02    Make 2nd device a Talker and Send its
SAI       Accessory ID.
DAB 16    2nd device responds with 16 (Mass
          Memory)
TAD 03    Same thing with the 3rd device.
SAI
DAB 64    64 means it's interface class; we found
          it.
UNL       That's all on the loop.  Stop listening.
TAD 03    Make device #3 a Talker.
SST       Send Status.
DAB 1     GPIO responds with a Data Byte 1.
          (Nothing's happening.)
UNT       Untalk.  Disable all talkers.
UNL       Unlisten.  Disable all listeners.
```

We can improve the situation substantially by telling the 71 where the device is:

```
A=DEVADDR("GPIO")   ! This only has to be done once.
SPOLL(A)
```

and the sequence gets shortened to:

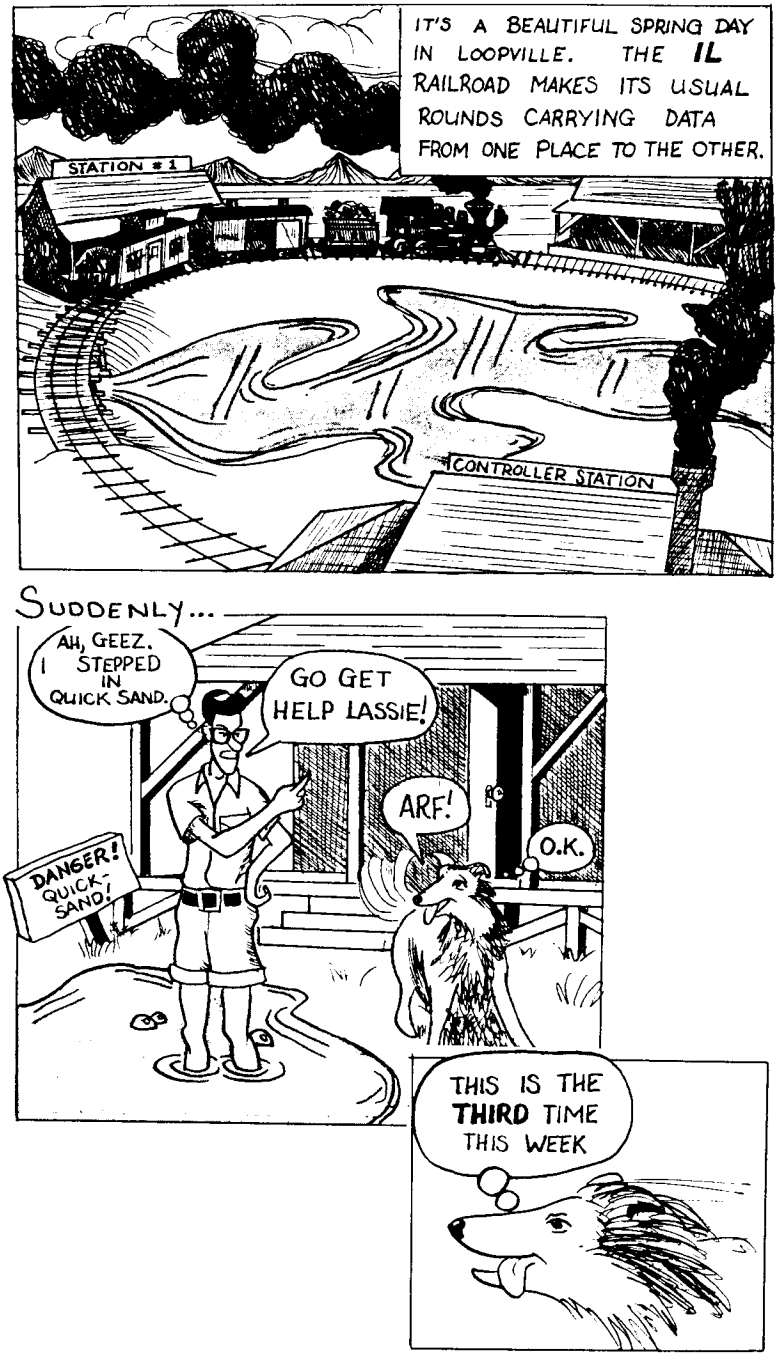
```
UNL
TAD 03
SST
DAB 01
UNT
UNL
```

An improvement, but still too much overhead. Clearly, in our controlled situation where a number's just been dialed and the loop configuration hasn't changed and no other device is going to be utilized, this is a big waste. Our aim is to take as many samples per second of the MSRQ line as we can. Fortunately, we can increase our sampling rate by making use of HP-IL Service Requests.

Service requests work sort of like bums on trains: they just hitch a ride on a vehicle that's making its rounds anyway. As any data frame is being passed around the loop, any device who needs service ("Yoo Hoo, Mr. Loop Controller! I have some data for you!") can simply affix this message to the data frame as it passes by. When this frame makes it back to the loop controller (as all frames must), the controller notices it and says "Hmmm... someone out there is trying to get my attention, but I don't know which device it is.", and immediately starts polling each device to find out which one sent the message.

In our tightly controlled environment we don't need to know who sent the message; we know to be expecting one from the GPIO as soon as a number is dialed. And since during this period of anticipation there is no data being sent, we will instead use a dummy data byte: the IDY (Identify) frame.

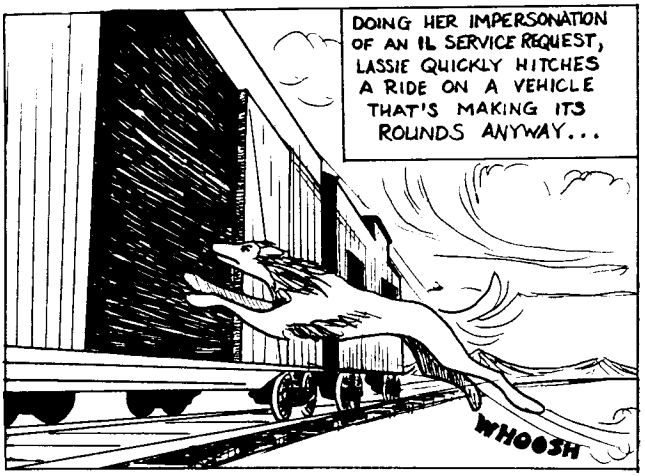
The IDY frame is (in this usage) analogous to empty railroad cars being sent out for the sole purpose of collecting bums, hobos,

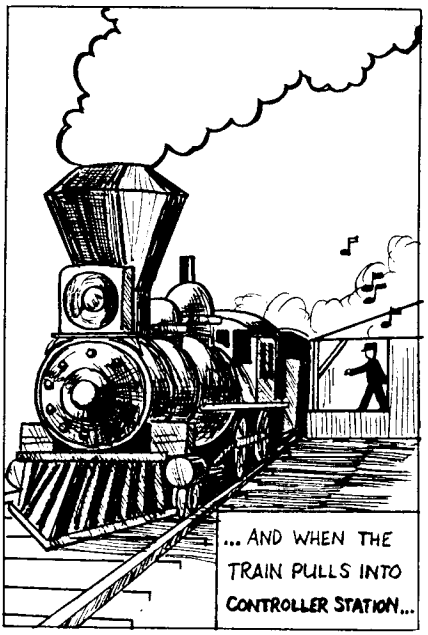

IT'S A BEAUTIFUL SPRING DAY IN LOOPVILLE. THE IL RAILROAD MAKES ITS USUAL ROUNDS CARRYING DATA FROM ONE PLACE TO THE OTHER.
STATION # 1
CONTROLLER STATION
SUDDENLY...
AH, GEEZ. I STEPPED IN QUICK SAND.
GO GET HELP LASSIE!
ARF!
O.K.
DANGER! QUICK-SAND!
THIS IS THE THIRD TIME THIS WEEK

vagabonds, and service requests. In the sample program below, we first warn the computer that any minute now we'll be expecting an interrupt frame to come in, and when it does we should branch to a certain line number. (Refer to lines 1100-1120.) Then, with the ENABLE INTR 8 command, we tell the 71 what kind of interrupts to respond to. (8 means to pay attention only to service requests, and ignore all other frames such as those that might be sent by other controllers.) Finally, we loop and send IDYs forever until a device requests service, in which case program control will automatically branch elsewhere and do something interesting. The advantage of this method is that we can instantly tell if another device needs service without the usual large overhead. All that is needed is a single frame - IDY - to assess whether there's sound on the line or not.

For this application, the GPIO has been configured by the DDL 0 command to send service requests whenever its MSRQ line goes low, corresponding to a noise on the telephone line. Because we also need to know how long there's been noise, we keep track of every time we loop while waiting for an IDY to come back positive.

```
1060 SUB BUSY(Y,A) ! Y returns yes/no, A=GPIO address
     from calling program.
1070 ! Ring: Y=0; Busy: Y=1.
1080 Y=0
1090 IF Y>=6 THEN Y=1 @ END     ! If we have too many
     clicks, assume it's a reorder and exit.
1100 ON INTR GOTO 1130
1110 X=0 @ ENABLE INTR 8
1120 SEND IDY @ X=X+1 @ IF X>300 THEN Y=1 @ END ELSE
     GOTO 1120     ! Wait for noise.
1130 ON INTR GOTO 1140 @ X=0
1140 X=X+1 @ ENABLE INTR 8
1150 IF X>=13 THEN Y=0 @ END ! If we hear a long
     noise, we have a ring.
1160 ! A noise will loop back to 1140; silence will
     drop to 1180.
1170 SEND IDY
1180 IF X<3 THEN Y=Y+1 @ GOTO 1090 ! Ignore short
     bursts but recognize reorders.
```


DOING HER IMPERSONATION OF AN IL SERVICE REQUEST, LASSIE QUICKLY HITCHES A RIDE ON A VEHICLE THAT'S MAKING ITS ROUNDS ANYWAY...
WHOOSH


... AND WHEN THE TRAIN PULLS INTO CONTROLLER STATION...


"I'LL JUST TAKE A LOOK IN HERE, AND..."

```
1190 X=0 @ ON INTR GOTO 1220
1200 ENABLE INTR 8
1210 SEND IDY @ X=X+1 @ IF X>20 THEN Y=0 @ END ELSE
1210 ! exit on a long pause.
1220 IF X<9 THEN Y=0 @ END ELSE Y=1 @ END
1230 ! If X<9 then other party must have picked it up
       after 1st ring.
```

Refinements were made that allow detection of a "re-order" (which is telephoneese for that fast busy signal you hear when all circuits are busy), and for the common case when the phone rings less than one time and someone answers immediately. This subroutine will detect that correctly, too!

Rolodex Function

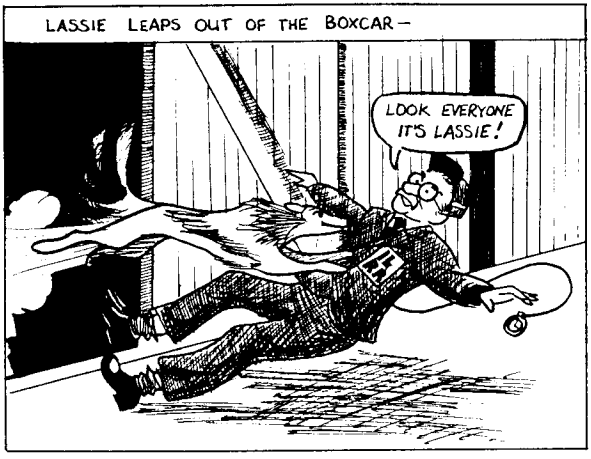
Since I've always been a fan of point-and-dial type features, I decided to implement one in this system. While the application is running (which ideally should be all the time), the cursor keys are used to scroll through a data file of your most frequently called numbers. When the desired name is displayed on the screen, hitting the "D" key automatically fetches the number associated with the name, and dials it.

If flag 1 is set, the subroutine "PREPROC"essor is called before the number is dialed. This flag indicates that I'm at my office, and will automatically dial the correct prefixes to access the appropriate outside line. Your office environment may vary, but mine requires the following rules:

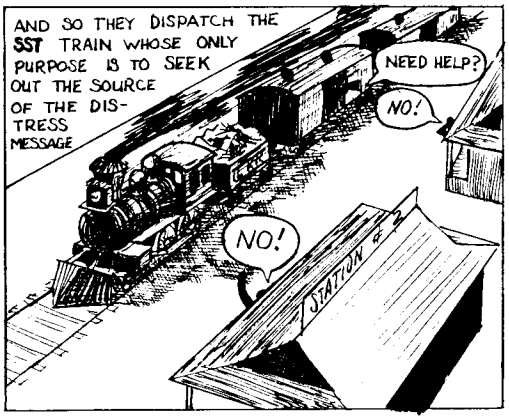
--If it's within the area code, dial "9" first.
--If the area code is 213 or 714, dial "91" first.
--If the area code is not 213, 714 or 818, dial "8" first.

The advantage of doing this with a subroutine is that I can run this at home with flag 1 clear, and it will dial "1" before any 10-digit string, just like normal people do.

The external file containing the names and numbers of your


LASSIE LEAPS OUT OF THE BOXCAR—
LOOK EVERYONE IT'S LASSIE!


SOMEONE MUST NEED HELP!
IL RR


AND SO THEY DISPATCH THE SST TRAIN WHOSE ONLY PURPOSE IS TO SEEK OUT THE SOURCE OF THE DISTRESS MESSAGE
NEED HELP?
NO!
NO!
STATION #2


HMMM... MIGHT BE IMPORTANT. BETTER GRAB IT!
YES!

friends is a simple text file named PHBOOK, and can be created using the EDTEXT function found in the FORTH/Assembler ROM. It is laid out like this, where the names alternate with the numbers:

```
Issac Asimov
2135552310
Carl Sagan
8051357911
Billy Hewlett
1248163
  :
  :
```

(Notice that "1" does not preced the area code. The "1" is automatically appended by the "PREPROC"essor routine. That way, whether this device is used in the home or office environment, PREPROC can dial the appropriate prefixes.)

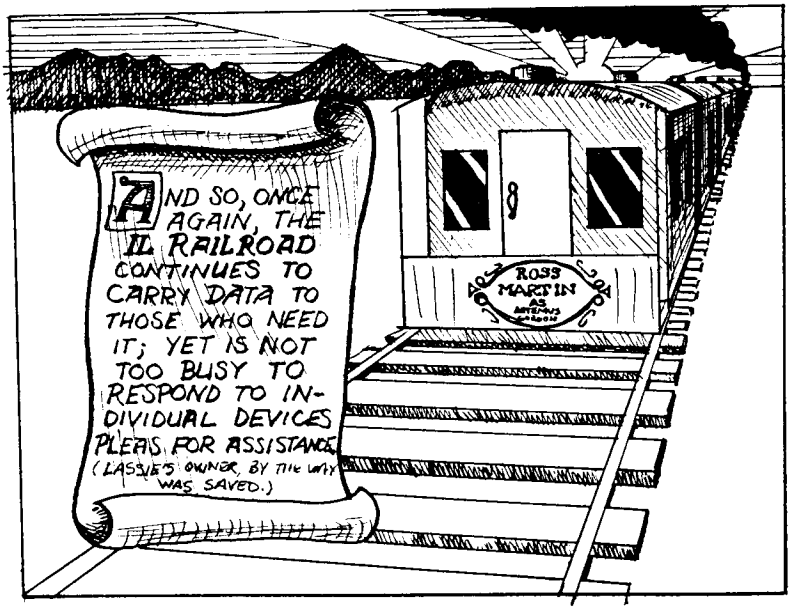
Outgoing Call Monitor

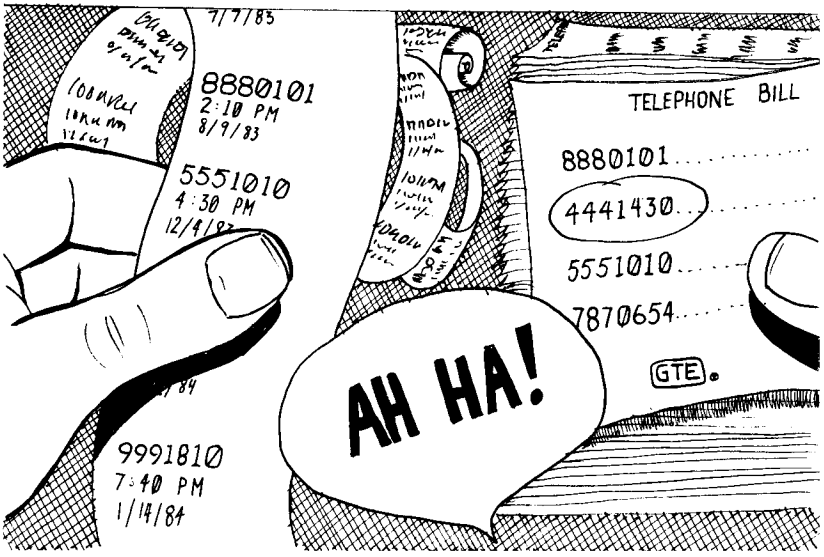
With this fringe benefit, we harness something the system does anyway--constantly monitors the phone line for manually made outgoing calls so it can mimic them in the "Last Number Dialed" function. Since we know what numbers are being dialed and what time and date they were made, why not keep track of them automatically?

Glad you asked. The difficult part of it, though, is determining which calls went through and were answered after several rings. (All the necessary hooks were in the existing algorithms, so the subroutine "log" was added at the end.) This feature creates and then appends a data file called PHLOG (Phone Log), which can be printed out once a month with this simple program:

```
10 ENDLINE
20 ASSIGN #2 TO * @ ASSIGN #2 TO PHLOG
30 READ #2;N$,T$,D$ @ PRINT N$;" ";T$;" ";D$ @ GOTO 30
```


AND UPON ITS RETURN...
STATION #5 SENT THIS SIR!
OH, NO! HE'S CAUGHT IN QUICKSAND! NOW THAT WE KNOW WHAT THE MESSAGE IS AND WHERE IT CAME FROM I CAN NOW ACT ACCORDINGLY!
IF THESE IDIOTS COULD'VE READ MY THOUGHT BALOONS THAT GUY WOULD STILL BE ALIVE!
CAPTAIN KIRKPATRICK


AND SO, ONCE AGAIN, THE IL RAILROAD CONTINUES TO CARRY DATA TO THOSE WHO NEED IT; YET IS NOT TOO BUSY TO RESPOND TO INDIVIDUAL DEVICES PLEAS FOR ASSISTANCE
(LASSIE'S OWNER, BY THE WAY WAS SAVED.)
ROSS MARTIN



Some Americans might prefer to rearrange the PRINT statement so as to display the date in the backward way to which they are accustomed.

Complete Instructions

0) (Where else but in computer literature do you see anything numbered "zero"?) Make sure the program (listed below) is in memory, along with the phone book file PHBOOK and optionally the program that prints PHLOG out once a month, which I confusingly call PRLOG (Print Log). Check the status of Flag 1, which tells the PREPROC subroutine whether you're calling from your home or office. (Refer to PREPROC subroutine for details.)

1) This system is invisible until you need its special functions. While your phone isn't being used, the 71 displays a running clock and checks for keyboard input and touch tone activity on the phone line.

2) Make an outgoing call on your phone in the usual manner. The number you dialed is now available as the LAST NUMBER DIALED (LND).

3) If the LND was busy, and you want the 71 to keep trying until it gets through, press "R". The 71 will then keep trying (you can monitor the call progress yourself using the optional amplifier, which is only on when the 71 has picked up the phone) until it determines that the other end is ringing, at which time it tells you by playing 5 long BEEPs. Pick up the phone, and then press any key on the 71 so it will "hang up".

4) To blindly redial the LND without the automatic redial-if-busy feature, press "L". When your call is completed (you'll know when that occurs if the amplifier's attached), pick up the phone and press any key on the 71 to have it hang up.

5) To use the Rolodex function, use the up-arrow and down-arrow keys to scroll through the names in your PHBOOK file. When the desired name is visible, press "D" to determine the number and dial it. If flag 1 is set, the proper office prefix will be dialed according to the instructions in PREPROC. When the call has gone through, pick up the phone and hit any key. This number is now the new LAST NUMBER DIALED.

6) Once a month (or sooner!) print out and/or delete the PHLOG file that the PHONE program appends phone usage data to whenever an outgoing call is made.

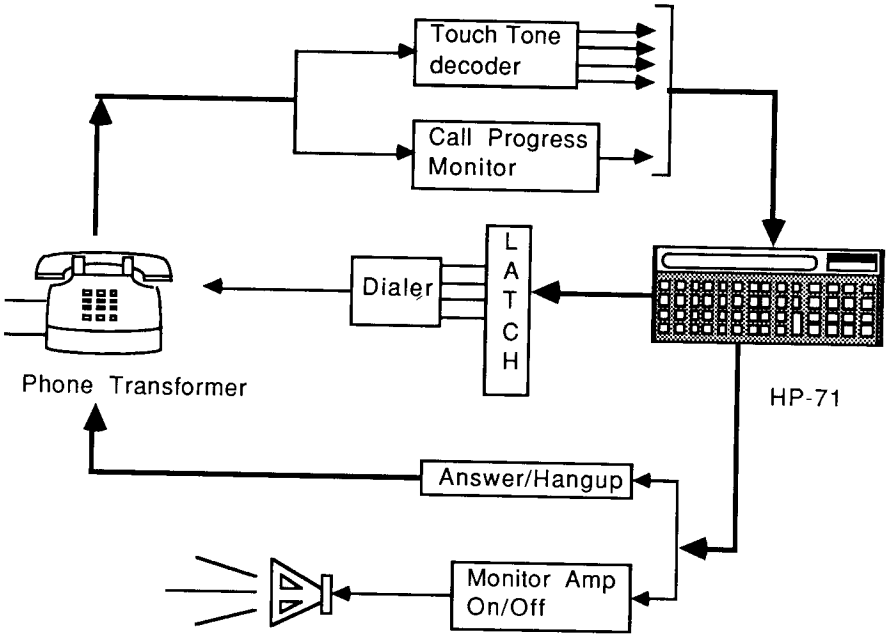
Here's the program:

```
10 ! Program PHONE controls just about everything.
20 ! Two data files are used:  PHBOOK, a name/phone #
     text file, and PHLOG, which logs all calls.
30 ! Keys used while active:  L redials last number
     dialed.
40 ! R means take the LND and redial until not busy.
50 ! Up and down arrows access names in your phone
     book.
60 ! D dials the name called up by up- and
     down-arrows.
70 ! Variables used: T$ is the last Touch Toned (r)
     number.
80 ! Z is the size of the Phonebook file.
```

```
90 ! N is the Phonebook file pointer; N$ is the last
    name retreived.
100 ! Flag 6 means hold display till x>30.
110 ! ***********************************************
120 GOSUB 'INIT'
130 'BEGIN': K$=KEY$
140 IF K$="L" THEN BEEP 2000,.07 @ CALL LND(T$,A) !
145 ! Last # dialed; bypass preprocessing.
150 IF K$="R" THEN BEEP 2000,.07 @ CALL REDIAL(T$,A) !
155 ! Redial till not busy.
160 IF MOD(SPOLL(A),64)>1 THEN CALL WHATISIT(T$,A) !
165 ! Investigate cause if service request.
170 IF K$="#50" THEN 'UPAR' ! Uparrow's been hit.
180 IF K$="#51" THEN 'DNAR' ! Down arrow has been
    pressed.
190 IF K$="#162" THEN 'MAXUP' ! g-uparrow's been hit.
200 IF K$="#163" THEN 'MAXDN' ! g-downarrow's been
    hit.
210 IF K$="D" THEN BEEP 2000,.07 @ GOTO 'THIS' !
215 ! 'D' means "Dial this name!".
220 IF FLAG(6)=0 THEN DISP FNT$(TIME$)&FND$(DATE$) @
    GOTO 'BEGIN'
230 X=X+1 @ IF X>30 THEN X=0 @ CALL LOG(#2,T$)
240 GOTO 'BEGIN'
250 ! ***********************************************
260 'UPAR': N=MAX(N-2,0) @ GOTO 'READ'
270 'DNAR': N=MIN(N+2,Z) @ GOTO 'READ'
280 'MAXUP': N=0 @ GOTO 'READ'
290 'MAXDN': N=Z @ GOTO 'READ'
300 'THIS': READ #1,N+1;T$ @ DISP "   "&T$ @ CALL
    DIAL(T$,A) @ GOTO 'READ'
310 'READ': READ #1,N;N$ ! Read a name from the phone
    book file.
320 DISP N$ @ SFLAG 6 @ X=0 @ GOTO 'BEGIN'
330 ! ***********************************************
340 'INIT':
350 RESTORE IO @ RESET HPIL
360 A=DEVADDR("GPIO")
370 SEND UNT UNL LISTEN A MTA DDL 0 DATA
    194,16,218,0,0 UNL UNT
380 SEND MTA LISTEN A DATA 0 UNT UNL ! Shuts touch
```

```
    tone chip up.
390 DIM T$[20] @ DIM Q$[12]
400 ASSIGN #1 TO * @ ASSIGN #1 TO PHBOOK ! Opens the
    PHone BOOK file.
410 Z=FILESZR("PHBOOK")-2 420 ASSIGN #2 TO * @ ASSIGN
    #2 TO PHLOG
430 ON ERROR GOTO 440 @ FOR X=1 TO 9999 @ READ #2;N$ @
    NEXT X
440 OFF ERROR ! Ha! Fooled you!
450 DEF FNT$(T$) ! Displays the current time in a
    user-friendly format.
460 C=VAL(T$[1,2])
470 IF C=0 THEN 'MID' ! Branch here to handle 12 AM.
480 IF C>12 THEN 'PM' ! Displays PM format.
485 IF C=12 THEN FNT$="  "&T$[1,5]&" PM   " @ END
490 FNT$="  "&T$[1,5]&" AM   " @ END
500 'PM': FNT$="  "&STR$(C-12)&T$[3,5]&" PM   " @ END
510 'MID': FNT$="  12"&T$[3,5]&" AM   " @ END
520 END DEF
525 ! FND$ converts a DATE$ into a more familiar
      format.
530 DEF FND$(D$)=D$[4,8]&"/"&D$[1,2]
540 ENDLINE "" @ DELAY 0,0 @ CFLAG 0 @ SFLAG -23 !
545 ! Needed to have ENTER terminate with an EOT.
550 RETURN
560 ! ***********************************************
570 SUB DIAL(T$,A)
580 CALL PICKUP(A) ! Pick up the phone.
590 CALL PREPROC(T$,N$,A) ! Dial any needed prefixes.
600 CALL DIALIT(T$,A) ! Dial the number.
610 X=0 @ T$=N$&T$ ! The number + any prefixes is
    saved for LND.
620 IF KEY$#"" OR X>5000 THEN BEEP 2000,.07 @ CALL
    HANGUP(A) @ END ! Any key exits
630 X=X+1 @ GOTO 620
640 ! ***********************************************
650 SUB DIALIT(T$,A) ! Dials # without preprocessing.
660 SEND MTA LISTEN A
670 FOR X=1 TO LEN(T$)
680 OUTPUT :LOOP ;T$[X,X]
690 WAIT .02
```

```
700 SEND DATA 32 ! Keeps the phone off hook between
    digits.
710 ! WAIT .005
720 NEXT X
730 SEND UNT UNL
740 SFLAG 6 @ END SUB ! Tell main routine that a #'s
    just been dialed.
750 ! ***********************************************
760 SUB PICKUP(A) ! Picks up the phone.
770 OUTPUT :A ;CHR$(32) @ WAIT .75 @ END
780 SUB HANGUP(A)
790 OUTPUT :A ;CHR$(0) @ END
800 ! ***********************************************
```



Intelligent Autodialer
Block Diagram

```
810 SUB PREPROC(T$,N$,A) ! Preprocessor adds the
      proper digits to get outside lines, FTS, etc.
820 ! If flag 1 set, then office prefixes are used.
825 ! If flag 1 = clear, then it dials "1" before any
830 ! string > 7 digits.
840 ! N$ returns the preprocessor's additions.
850 IF FLAG(1)=0 THEN 'HOME'
860 IF LEN(T$)<=7 THEN N$="9" @ CALL DIALIT(N$,A) @
    END
870 IF T$[1,3]="213" OR T$[1,3]="714" THEN 'NOTLOCAL'
880 N$="8" @ CALL DIALIT(N$,A) @ WAIT .5 @ END
890 'HOME': IF LEN(T$)<=7 THEN END ELSE N$="1" @ CALL
     DIALIT(N$,A) @ END
895 'NOTLOCAL': N$="9" @ CALL DIALIT("9",A) @ WAIT .5
     @ T$="1"&T$ @ END
900 ! ************************************************
910 SUB WHATISIT(T$,A) ! Determins if noise is TT
    activity or a dial tone.
920 S=MOD(SPOLL(A),64)
930 IF S=2 THEN ENTER :A ;Q$
940 IF S=2 AND FLAG(0)=0 THEN T$=Q$ @ SFLAG 0 ELSE
    T$=T$&Q$
950 IF S=2 THEN DISP T$ @ GOTO 'CHECK'
960 IF S=32 THEN S=MOD(SPOLL(A),64)
970 IF S=2 THEN 930
980 IF S=1 AND FLAG(0) THEN DISP T$ @ CFLAG 0 @ END
    ELSE END
990 ! If flag 0 set, this indicates that T$ is still
      under construction.
1000 'CHECK': X=POS(T$,":") ! Correct for any incoming
      zeros.
1010 IF X=0 THEN SFLAG 6 @ END ELSE T$[X,X]="0" @ GOTO
     'CHECK'
1020 ! ***********************************************
1030 SUB REDIAL(T$,A) ! Redials last T$ until ring or
     any key hit.
1040 DISP "   "&T$
1050 'START': CALL PICKUP(A)
1060 IF LEN(T$)>7 THEN CALL DIALIT(T$[1,1],A) @ WAIT
     .5 @ CALL DIALIT(T$[2],A) @ GOTO 1090
1080 CALL DIALIT(T$,A)
```

```
1090 CALL BUSY(Y,A)
1100 ! DISP "   "&T$ ! Keep this commented line in.
1110 IF KEY$#"" THEN BEEP 2000,.07 @ CALL HANGUP(A) @
     END ! Any key exits.
1120 ! IF Y=1 THEN CALL HANGUP(A) @ WAIT 1 @ GOTO
     'START'
1121 IF Y=1 THEN CALL HANGUP(A) @ WAIT 1 @ GOTO
     'START'
1130 FOR X=1 TO 5 @ BEEP 880,.25 @ NEXT X
1140 IF KEY$#"" THEN BEEP 2000,.07 @ CALL HANGUP(A) @
     END ELSE GOTO 1140 ! Any key hangs up.
1150 ! ********************************************
1160 SUB BUSY(Y,A) ! Y returns yes/no, A=GPIO address
     from calling proram.
1170 ! Ring: Y=0; Busy: Y=1.
1180 Y=0
1190 IF Y>=6 THEN Y=1 @ END ! If we have too many
     clicks, assume it's a reorder and exit.
1200 ON INTR GOTO 1230
1210 X=0 @ ENABLE INTR 8
1220 SEND IDY @ X=X+1 @ IF X>300 THEN Y=1 @ END ELSE
     GOTO 1220 ! Wait for noise
1230 ON INTR GOTO 1240 @ X=0
1240 X=X+1 @ ENABLE INTR 8
1250 IF X>=13 THEN Y=0 @ END ! If we hear a long
     noise, we have a ring.
1260 ! A noise will loop back to 1110; silence will
       drop to 1150.
1270 SEND IDY
1280 IF X<3 THEN Y=Y+1 @ GOTO 1190 ! Ignore  short
     bursts but recognize reorders.
1290 X=0 @ ON INTR GOTO 1320
1300 ENABLE INTR 8
1310 SEND IDY @ X=X+1 @ IF X>20 THEN Y=0 @ END ELSE
1310 ! exit on a long pause.
1320 IF X<9 THEN Y=0 @ END ELSE Y=1 @ END
1330 ! If X<9 then other party must have picked it up
       after 1st ring.
1340 ! ********************************************
1350 SUB LND(T$,A) ! Performs Last Number Dialed.
1355 DISP "   ";T$ @ X=0
```

```
1360 CALL PICKUP(A)
1370 IF LEN(T$)>7 THEN CALL DIALIT(T$[1,1],A) @ WAIT
     .5 @ CALL DIALIT(T$[2],A) @ GOTO 1400
1390 CALL DIALIT(T$,A)
1400 DISP "    ";T$ @ X=0
1410 IF KEY$#"" OR X>5000 THEN BEEP 2000,.07 @ CALL
     HANGUP(A) @ END
1420 X=X+1 @ GOTO 1410
1430 ! ********************************************
1440 SUB LOG(#2,T$)
1450 ! Appends numbers to the PHLOG (Phone Log) file.
1460 CFLAG 6
1465 END
1470 PRINT #2;T$,TIME$,DATE$
1480 END SUB
```

Well, that's all there is to it! Of course, this is just a demonstration of all the things possible with this hardware. A few more program refinements will allow the PHLOG file to record the call's duration as well as time and date; and a fancier PRLOG program could pull only the calls made between "this date" and "that date". This is where your needs and your imagination take over.

Chapter Seven

HP-41 BASED TELEPHONE ANSWERING MACHINE

Utilizing Speech Synthesis and Touch Tone Decoding

I use my Hewlett Packard for the answers found in Calculus
And problems most encountered in numerical analysis.
It calculates proportions used in heart and lung dialysis.
Eventually I'll work for them and move to where Corvallis is...

-- from I am the Very Model of an Engineering Graduate
by G. Friedman

This is the most outrageous 41 project I was able to dream up. It incorporates the speech synthesis discussed in Chapter 5, and in addition also encompasses Touch Tone recognition, telephone line interface (see previous chapter regarding FCC rules), and an Extended Memory database function. It allows you and your friends to interact with the machine from a pushbutton phone, and even lets you control things in your house from across the country!

This system performs the same functions as standard telephone answering machines: detecting a ring, "picking up" the phone, synthesizing a "Hello, I'm not home" type message, turning on a tape recorder to record the caller's verbal message, and hanging up. The fun starts when you start to add the following capabilities:

1) Automatic (verbal) Date and Time Stamping--At the end of each caller's message, the machine announces the current time and date, so you'll know upon playback when the calls came in. (Nobody ever leaves the time they called when you ask them to.)

2) Priority Message Taking--Priority callers can Touch Tone in a preassigned 4-digit code in lieu of a verbal message; the caller then gets a personalized "Thank you" (i.e. "Thank you, Nancy"), and a phone message containing the first and last name, phone number, time and date is instantly printed out, resulting in a printed list of priority clients whose calls should be returned first. (Non-priority callers will just have to wait until you get around to listening to their taped messages before their calls are returned.)

3) Priority Message Transmission--The list generated in item #2 can be read back to you over the phone in case you can't get to your equipment. After the user's 4-digit Touch Tone code is entered, the system will announce the number of messages, first and last names of each client, phone number, time and day they called.

4) Extremely Remote Control--This system also allows you to turn up to five A.C. appliances on or off from anywhere in the world just by entering the proper 4-digit Touch Tone code. Verbal confirmation and the status of the device (i.e. "Device number 1 is off.") is announced.

5) Outgoing Call Monitor--When the user is at home, the system doubles as an outgoing call monitor, where the time, date, and number dialed from any Touch Tone extension phone is automatically logged, providing a hardcopy record of all outgoing calls so they can be checked against the Phone Co.'s bills.

There is very little new information introduced in this chapter. It exists because 1) it is the most versatile answering machine on the planet, and 2) it gives yet another example of the diverse things that can be done with just a few basic circuits. It also exists to show that the 41 is no less a capable controller than the 71.

New Hardware Aspects

The circuit needed to implement the above function doesn't differ greatly from those described in previous chapters. (See schematic of Fig. 7-1.) The speech synthesizer and accompanying latch are the same as described in Chapter 5. The Touch Tone decoder interfaces to the phone line in the same way as in the previous chapter. So, to avoid repeating myself only the new hardware aspects will be described.

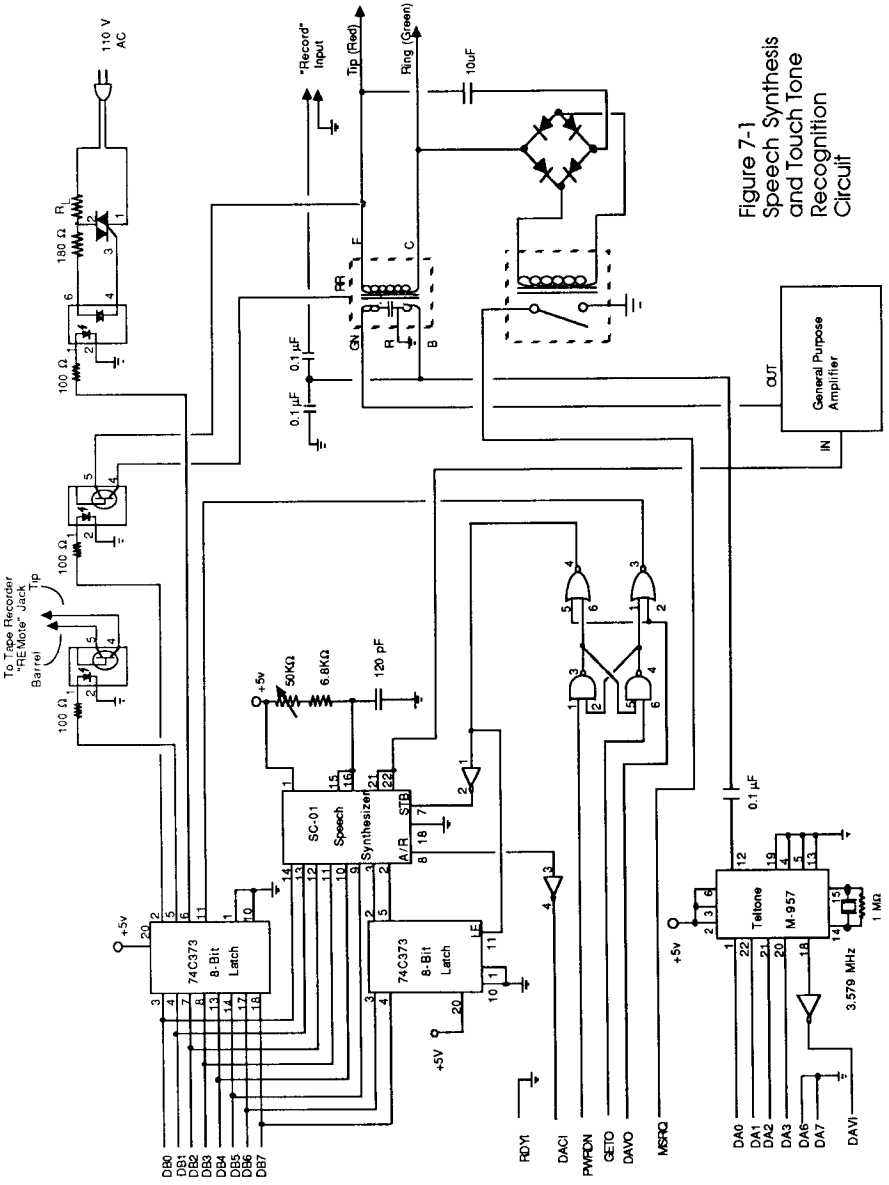


Figure 7-1
Speech Synthesis and Touch Tone Recognition Circuit

The first things you'll recognize immediately in Fig. 7-1 are located along the top: an 8-bit latch controlling three opto-isolators. Sharing the same input lines, directly below the latch, is the speech circuitry as described in Chapter 5. The opto-isolators are used to control three things in exactly the same manner as was demonstrated in the first two chapters: picking up/hanging up the phone, turning a tape recorder on and off via its "Remote" input, and switching an AC appliance on and off via a triac driver. This leaves five outputs unused, which means there is still room to expand on this system's already awesome capability.

The other part you'll recognize is the M-957 Touch Tone decoder chip, wired slightly differently, in the lower left-hand corner. The wiring differences put the chip into a more sensitive receiving mode, to allow it to hear codes dialed from, say, France.

Outgoing Call Monitor

Just to whet your appetite, here's a program for the 41 which keeps a record of all your outgoing calls, just like the 71 did in Chapter 5. (Barcode for this program begins on page 285.) With the Touch Tone decoder chip attached to the phone line as per the schematic, this program "listens in" on the line, displays all Touch Tone activity in blocks, and makes the necessary character corrections for *, 0, #, A, B, C, and D, as described in the previous chapter. If a printer is attached, a record of all activity, including the date and time, is automatically printed out! (To increase the speed, A, B, C, and D will be decoded only if flag 1 is set.) Now, next time you scream "I never made those calls!!" upon receipt of your phone bill, you can prove it (at least to yourself).

```
01*LBL "INTT2"      10 .035          19 RCLPT
02 XEQ "GPIO"       11 STO 01        20 X>0?
03 "TTONE"          12 CF 21         21 DELREC
04 3                13*LBL 01        22 .035
05 SF 25            14 INSTAT        23 STO 01
06 CRFLAS           15 FS? 01        24 FC?C 08
07 RCLPT            16 GTO 02        25 GTO 11
08 X>0?             17 ISG 01        26 32
09 DELREC           18 GTO 01        27 FINDAID
```

```
28 X=0?
29 GTO 11
30 SF 21
31 SF 12
32 ADV
33 PRA
34 CLA
35 CF 12
36 TIME
37 FIX 2
38 ATIME
39 PRA
40 CLA
41 DATE
42 FIX 4
43 ADATE
44 PRA
45 CF 21
46*LBL 11
47 CLA
48 GTO 01
49*LBL 02
50 FC? 08
51 TONE 8
52 RCL 02
53 SELECT
54 INA
55 FC? 04
56 GTO 07
57*LBL 03
58 61
59 POSA
60 X<0?
61 GTO 04
62 65
63 X<>Y
64 YTOAX
65 GTO 03
66*LBL 04
67 62
68 POSA
69 X<0?
70 GTO 05
71 66
72 X<>Y
73 YTOAX
74 GTO 04
75*LBL 05
76 63
77 POSA
78 X<0?
79 GTO 06
80 67
81 X<>Y
82 YTOAX
83 GTO 05
84*LBL 06
85 48
86 POSA
87 X<0?
88 GTO 07
89 68
90 X<>Y
91 YTOAX
92 GTO 06
93*LBL 07
94 58
95 POSA
 96 X<0?
 97 GTO 08
 98 48
 99 X<>Y
100 YTOAX
101 GTO 07
102*LBL 08
103 59
104 POSA
105 X<0?
106 GTO 09
107 42
108 X<>Y
109 YTOAX
110 GTO 08
111*LBL 09
112 60
113 POSA
114 X<0?
115 GTO 10
116 35
117 X<>Y
118 YTOAX
119 GTO 09
120*LBL 10
121 APPCHR
122 0
123 SEEKPT
124 GETREC
125 SF 08
126 AVIEW
127 .035
128 STO 01
129 GTO 01
130 END
```

(You can also find out the sequence required to play "Mary had a Little Lamb" the next time you're on the phone with an 8-year-old.)

And now, on with the circuit explanations.

Telephone Line Interface

Half of this critical function is identical to that of the intelligent autodialer from the previous chapter. The other half, consisting of a 10 microfarad capacitor, a bridge rectifier, and a relay (yes, a relay) as shown in Figure 7-1 detects the characteristic AC signal of a ringing telephone. When the relay is actuated it pulls the MSRQ line to ground, and the controlling computer must check the IL Converter's STATUS word periodically to detect the ring. (This is the identical method used to the tell the 71 there was noise on the line in the previous chapter.) Yes, there are more modern ways to detect a ring, but there still is no better way to convert a 90V AC signal into a safe, pull-down ground.

The clump of components in the lower right-hand corner comprise an amplifier module, which was added between the speech chip's output and the "GN" input of the hybrid transformer to insure high voice quality over the phone line. This can be anything from a tape recorder set to "monitor" mode to a small 1-transistor type, available as an off-the-shelf item in most electronics stores.

Chip Select

When you wish to communicate with only one of many chips connected to a data bus, the common method to do so is by utilizing the 'chip select' inputs which many ICs possess and controlling them with a ninth address line, as shown in Fig. 7-2a.

Normally, adding a decimal 256 to outgoing words will route those words to the lower chip, and words less than (or equal to) 255 automatically go to the top. However, this method cannot be employed here because 1) the SC-01 has no chip select input, and 2) in order to obtain a 9th data line, the IL Converter must be configured to be 16 bidirectional lines. This results in nearly doubling the time it takes to output an alpha string, as well as doubling the number of characters in the string needed to say a phrase. On the 41, these are quite significant!

The solution (shown in Fig. 7-2b) makes use of two NAND gates configured as a homebrew R/S flip-flop, and two unused lines from

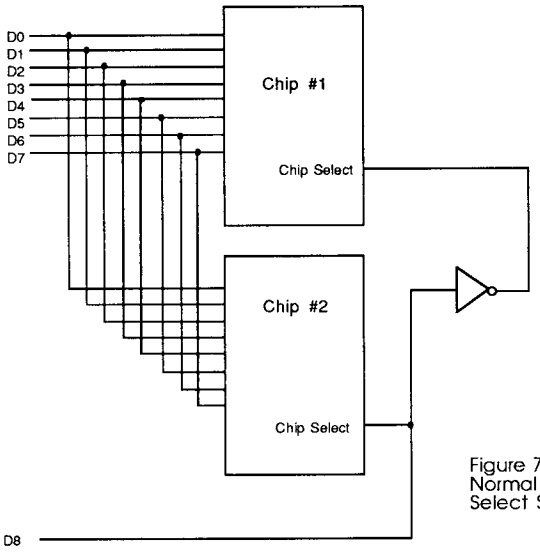


Figure 7-2a
Normal Chip
Select Scheme

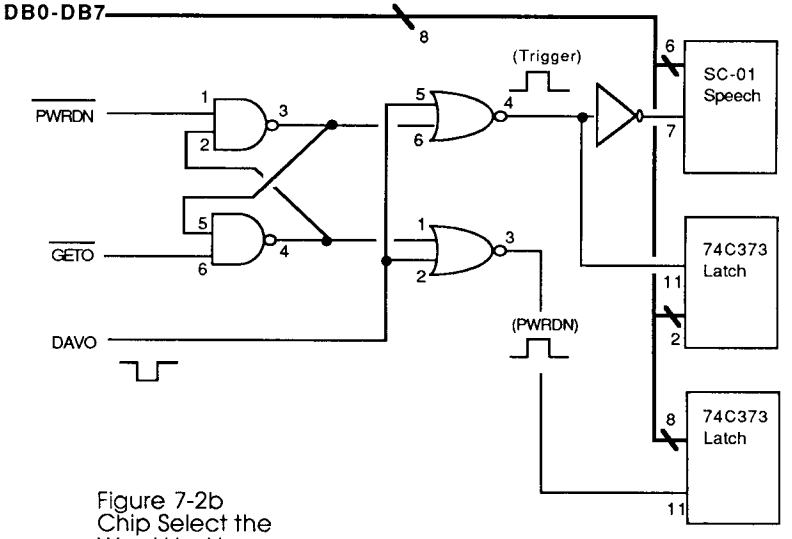
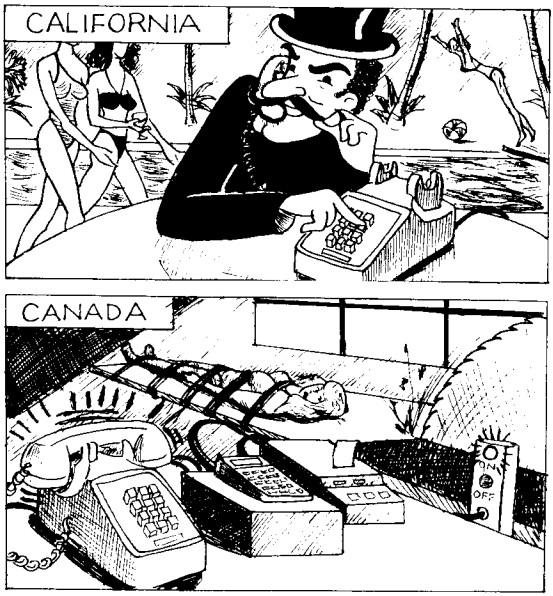


Figure 7-2b
Chip Select the
Way I Had to
Do It.

the 82166A IL Converter: PWRDN (Power Down, used to command a low-power sleep state) and GETO (Group Execute Trigger Out, normally used to synchronize events around the loop). Both can be pulsed under program control, making them ideal inputs to the flip-flop which "memorizes" which line was pulsed last.

The flip-flop's output then enables one of two NOR gates which route the DAVO (Data Valid Out) signal to the proper chip. This way the data reaches both chips, but only one chip is being told that the data is valid. (Clever, huh?) Using this method, a 41 would route its output to the currently SELECTed device (assume it's the GPIO) as follows. To send data to the speech chip:

```
TRIGGER   (Pulses the GETO output high, and routes
          all subsequent info to the speech chip)
SF 17     (Suppresses automatic CR/LF.)
MANIO     (Routes information to selected device
          instead of a printer.)
*!&^#@(   (It's actually a synthetic text line
          containing a speech pattern.)
OUTA      (Send it out to the SELECTed device.)
```



and to send data to the control latch:

```
PWRDN     (Pulses the PWRDN output high, and
          routes all subsequent info to the
          control latch.)
MANIO     (Routes information to selected device
          instead of a printer.)
2 ACCHR   (Turns the tape recorder on.)
```

Because the PWRDN (Power Down) line plays such a crucial role in this system, only the 82166 IL Converter can be used. The 82165A HP-IL/GPIO interface, as described in Chapter 1, lacks this output and therefore cannot easily accommodate our needs.

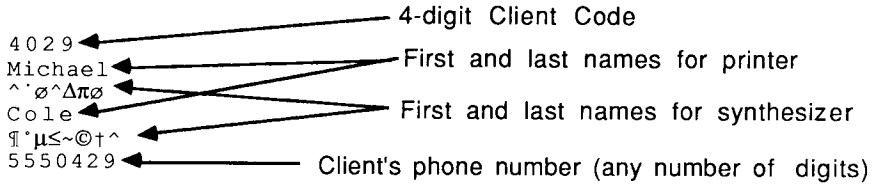
Software Description

The software detailed below performs as follows: after detecting a predetermined number of rings, the phone is answered and it says "Hello. This is Gary's Hewlett Packard. At the tone please leave your name, message, and telephone number. Beep!" (I'm too lazy to build an oscillator, so I just programmed the synthesizer to say "Beep!"). A tape recorder turns on, and the caller has 20 seconds to leave a verbal message. After the time is up, the machine announces the current time and day, (so you'll know upon playback when the message came in), says "Good-bye", turns off the tape recorder and hangs up.

You also have the option of assigning priority client status to any of your friends (or enemies) by giving them a personalized 4-digit code. If this client code is Touch Toned in during the time normally allotted for a verbal message, the tape recorder shuts off, and the machine says "Thank you, Isaac" (or whatever that client's first name happens to be). The current time and day are announced, and finally, a "Good-bye" and hang up occur. A message on the IL printer is then generated containing the caller's first name, last name, phone number, time and date of the call.

By the way, all this personal information has been stored away

in an ASCII file named "CLIENT", which has the data on all your friends organized as follows:




```
4029
Michael
^·ø^Δπø
Cole
¶°µ≤~©†^
5550429

7416
Seth
å¡£≈√·
Vallas
µ~...πø°••π∫√∂·
2125557416
```

The ASCII file is searched until a match is found to the code just entered. I usually assign the client's number to be the same as the last four digits of their phone number, making it easy for them to remember.

The system also features two user codes. One (which I assigned as 4111) will tell me, over the phone, how many priority clients have called, and then will read to me all the information contained in the printed list so far. The number of messages the calculator can remember is = (SIZE-12). The other user code, 4132, will toggle an alternating current device on or off, with verbal confirmation, each time the code is entered. This gives you remote control capability from anywhere in the world.

At all times, the display gives a running count of the number of calls that have been received.

The Program

All the software needed for this system is provided in the following pages. The program "TIMED" from Chapter 5 is also called by many of these routines. (Barcode for these programs, by the way, is provided beginning on page 287.)

"GPIO"
This is an initializing program, designed to configure the IL Converter as follows:

1) 8-bit unidirectional data transfer (8 lines going in, 8 lines going out.
2) Positive data logic.
3) Full negative handshake logic.
4) No CR/LF on input.
5) Status word shows MSRQ and that there's data waiting.
6) End-Of-Transmission occurs when the buffer is empty.

The program is listed below:

```
01*LBL "GPIO"      09 0               17 PWRDN
02 "D+++++"        10 DDL             18 0
03 ADROFF          11 5               19 STO 06
04 64              12 OUTAN           20 XEQ "ACCHAR"
05 FINDAID         13 UNL             21 TRIGGER
06 SELECT          14 ADRON           22 3
07 STO 02          15 SF 17           23 XEQ "ACCHAR"
08 LAD             16 AUTOIO          24 END
```

```
SYNTHETIC TEXT LINES:
 02: 246,68,194,16,218,0,0
```

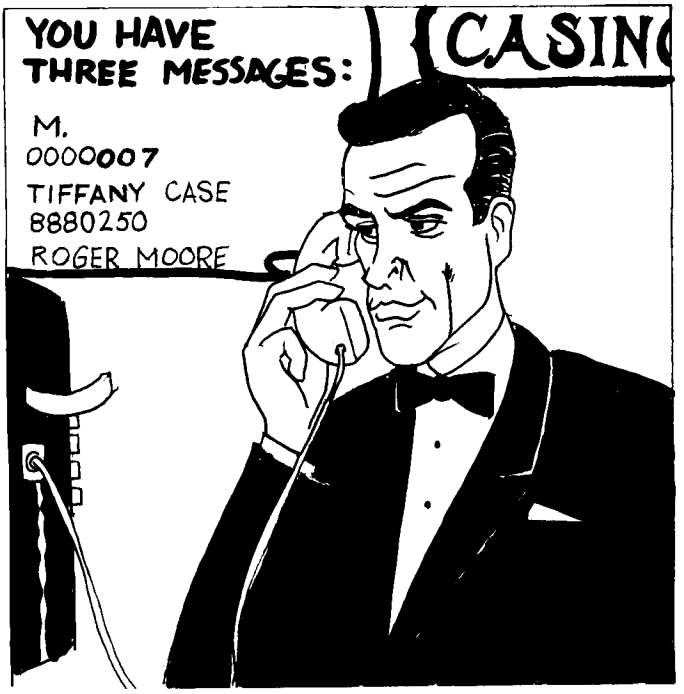
"ANSWER" (Main driver routine)

```
01*LBL "ANSWER"    12 "TIME"          23 1
02 XEQ "GPIO"      13 RCLPTA          24 X/Y?
03 CF 29           14*LBL 01          25 "|S"
04 0               15 0               26 AVIEW
05 STO 01          16 STO 01          27 CLA
06 STO 04          17 FIX 0           28*LBL 30
07 .999            18 "   "           29 FS? 49
08 STO 05          19 RCL 05          30 OFF
09 11              20 INT             31 INSTAT
10 +               21 ARCL X          32 FS? 01
11 STO 08          22 "| CALL"        33 CLRDEV
```

```
34 FC? 05
35 GTO 30
36*LBL 05
37 .999
38 STO 00
39*LBL 02
40 INSTAT
41 FC? 05
42 GTO 03
43 ISG 00
44 .
45 GTO 02
46*LBL 03
47 ISG 01
48 .
49 RCL 00
50 4
51 X>Y?
52 GTO 01
53 RCL 01
54 1
55 X=Y?
56 GTO 06
57 .03
58 STO 00
59*LBL 04
60 INSTAT
61 FS? 05
62 GTO 05
63 ISG 00
64 GTO 04
65 "   ASSHOLE"
66 AVIEW
67 PSE
68 GTO 01
69*LBL 06
70 E^X
71 LN
72 0
73 XEQ "FLIP"
74 ISG 05
75 5
76 LN
77 E^X
78 LN
79 E^X
80 INSTAT
81 FC? 01
82 XEQ "MSG2"
83 1
84 XEQ "FLIP"
85 "++++++"
86 OUTA
87 XEQ "INTT4"
88 FC? 09
89 GTO 12
90 ASTO 07
91 "CLIENT"
92 0
93 SEEKPTA
94 CLA
95 ARCL 07
96 POSFL
97 SF 25
98 X<0?
99 GTO IND 07
100 FC? 25
101 CF 09
102 FC?C 25
103 GTO 13
104 "++++++)(-+"
105 INT
106 1
107 +
108 SEEKPT
109 ARCLREC
110 SF 17
111 OUTA
112*LBL 12
113 XEQ "TIMED"
114 5
115 E^X
116 LN
117*LBL "RETURN"
118*LBL 13
119 XEQ "BYE"
120 RCL 06
121 X<>F
122 CF 01
123 X<>F
124 STO 06
125 PWRDN
126 XEQ "ACCHAR"
127 TRIGGER
128 0
129 XEQ "FLIP"
130 FC? 09
131 GTO 01
132 DATE
133 DOW
134 1 E5
135 /
136 TIME
137 FIX 2
138 RND
139 100
140 /
141 +
142 "CLIENT"
143 RCLPTA
144 INT
145 +
146 STO IND 08
147 ISG 08
148 32
149 FINDAID
150 X=0?
151 GTO 01
152 "CLIENT"
153 RCLPTA
154 1
155 +
156 INT
```

```
157 SEEKPT          168 SF 17           179 TIME
158 GETREC          169 PRA             180 FIX 2
159 SF 12           170 CLA             181 ATIME
160 SF 21           171 CF 12           182 "|      "
161 SF 17           172 1               183 DATE
162 ADV             173 +               184 FIX 4
163 PRA             174 SEEKPT          185 ADATE
164 2               175 GETREC          186 PRA
165 +               176 SF 17           187 CF 21
166 SEEKPT          177 PRA             188 GTO 01
167 GETREC          178 CLA             189 END
```

```
SYNTHETIC TEXT LINES
 104:     250, 185, 174, 169, 141, 153, 131, 41, 40,
         45, 3.  "Thank You"
 85:  246, 142, 172, 172, 172, 165, 131.  "Beep!"
```



LINES 1-13

General initialization. Configures IL Converter (via "GPIO" subroutine below) and shuts off all devices and stores constants in registers. (See also register usage table.)

LINES 14-27

ALSO CALLED LBL 01. Resets display with "(X) CALLS", X being the integer portion of R05. If only one call occurred, the "S" at the end is dropped.

LINES 28-35

This is the infinite loop performed while waiting for a call to come in. After an INSTAT, Flag 5 will indicate if the MSRQ is grounded, indicating the phone ringing. If not, it checks for low batteries (Flag 49), and clears the IL converter's buffer in case someone still at home is making an outgoing call on a Touch Tone phone (Flag 1 would indicate this condition.)

LINES 36-45

Waits for the ring to stop. Loop count of ring duration is stored in R00.

LINES 46-68

Was the detected ring too short? (R00<4?) If so, it probably was transient line current; go back to waiting for a ring. If not, wait for next ring. If we loop more than 30 times waiting for the next ring and it doesn't occur, the impatient calling party has hung up, and an appropriate expletive is displayed. PSE, and continue back to LBL 01, wait mode. If not, we have a valid call on our hands.....

LINES 69-82

The best way to damage circuitry is to answer the phone in the middle of a ring. So after the ring has stopped, we wait (LN, e/X) before answering. 0 XEQ "FLIP" means toggle bit 0 in the control latch, which will short out F and RR of the hybrid transformer and answer the phone. We now have a bona fide call, so increment R05, the call count register. Again we wait a moment (LINES 75-79), and if during that time any Touch Tone key is pressed, skip the outgoing message and go directly to the "BEEP!" (If you've ever grown impatient during extensive debugging, you'll know why I've included this feature.) Otherwise, "MSG2" (Message #2, part of a whole library) will speak a message to the caller.

LINES 83-86

Immediately before the beep tone, the tape recorder is turned on (1, XEQ "FLIP") and then, since building an oscillator was too much trouble, I have the speech chip say "BEEP!" (LINE 85,86). Really cute, huh?

LINE 87

Control is then passed to "INTT4" (Input Touch Tone, version 4) which, while the tape recorder is recording the caller's verbal message, listens in case the verbal message happens to be comprised of four Touch Tone digits. If it is, the tape recorder shuts off, Flag 9 is set, the four-digit number is AVIEW'd, and we return. If not, allow 20 seconds for the caller's message, and return with Flag 9 clear. (See documentation of INTT4 for details.)

LINES 88-96

If there was no Touch Tone activity (Flag 9 clear), GTO 12. Otherwise, begin a search of the ASCII file "CLIENT" for a match to the 4-digit code. (Register 7 is a temporary location for the code while "CLIENT" is made the working file.)

LINES 97-103

If no match is found, then maybe one of the many user's codes was entered. A GTO IND 07 will search for a 4-digit global label to execute. If the label is not found, (Flag 25 clear), we GTO 13 which will say "Good-bye" and hang up.

LINES 104-111

At this point a client's 4-digit code has been found in the "CLIENT" file. The system first says "Thank you" and then GETREC's the phonetic coding of the client's first name and says it. Your clients are rightfully blown away.

LINES 112-116

Here, "TIMED" (Time + Day, a subroutine covered in Chapter 5) is executed. If we've determined that a priority client is calling, the current time and day is spoken to them and eventually printed out. If this is just an ordinary person (INTT4 comes back with Flag 9 clear), the time and day is still spoken and will be picked up by the tape recorder, which would still be on. Then upon playback you'll know when the calls came in.

LINES 117-129

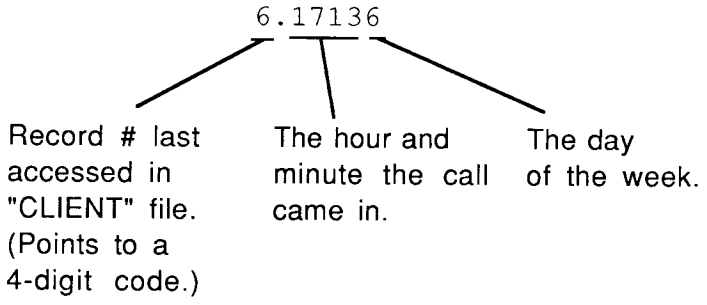
"RETURN". The program will always return to this point,

as it performs the vital function of saying "Good-bye" and hanging up.

LINES 130-147

If the caller wasn't a priority client, (Flag 9=clear), we go back and wait for another call. If it was, we will store the vital statistics of the message into memory (so the program can read the messages back to you later) and print out a phone message.

The vital statistics of each message are stored into a data register as follows:



The "4111" program will use this information later to read the client's first and last name, phone number, time and day they called back to the user. These "compact messages" are stored in data register R11 and higher.

LINES 148-188

If a printer exists, we print out a phone message. Notice I rely on the FINDAID (Find Accessory ID) command to verify printer existence because I don't trust Flag 55 for the IL printer. The remaining lines simply GETREC and PRA records from the file, and print the current time and date. Flag 17 is constantly set to nullify the effects of GETREC and suppress CR/LF on output.

"INTT4" (Input Touch Tone, version #4)
NOTE: This subroutine is not to be confused with the Outgoing call monitor program INTT2, which is a self-contained monitoring program documented elsewhere.

```
01*LBL "INTT4"     13 FC? 09          25 48
02 CF 09           14 RTN             26 X<>Y
03 .09             15*LBL 02          27 YTOAX
04 STO 01          16 SF 25           28 GTO 07
05 CF 21           17 4               29*LBL 08
06 CLA             18 INAN            30 AVIEW
07*LBL 01          19 ATOX            31 SF 09
08 INSTAT          20*LBL 07          32 1
09 FS? 01          21 58              33 XEQ "FLIP"
10 GTO 02          22 POSA            34 END
11 ISG 01          23 X<0?
12 GTO 01          24 GTO 08
```

LINES 1-6

General initializing.

LINES 7-14

Start listening for Touch Tone activity. If flag 1 is cleared after an INSTAT (=buffer empty), increment a counter (R01) and INSTAT again. If we have looped 90 times (= about 20 seconds on a 1.5X machine) exit the subroutine.

LINES 15-19

If there is data (Flag 1=set), input four digits using the INAN command. Flag 25 is set in case your caller enters less than four digits; the error will be handled by the main routine later. The ATOX on line 19 gets rid of the extra dummy character in ALPHA put there by the Extended I/O ROM command.

LINES 20-28

If a 0 (Operator) button is pressed by the caller, it shows up in the ALPHA register as a colon (:). These lines replace every found colon with a zero.

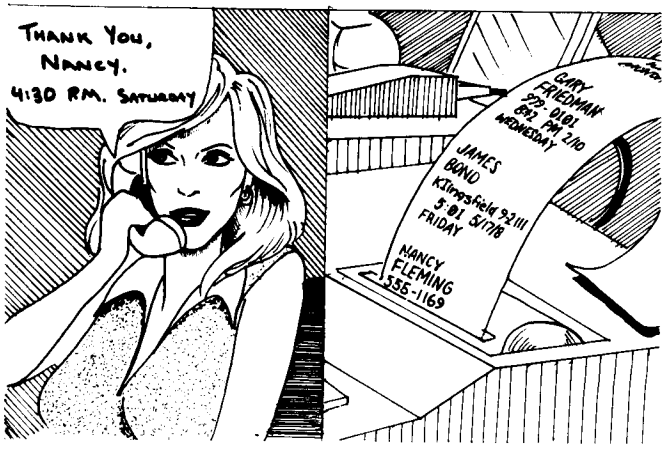
LINES 29-34

The number is AVIEW'd for local feedback, Flag 9 is set indicating valid data in the ALPHA register, the tape recorder is shut off early, and control is returned to the calling program.

"4132"
(The 4-digit code to turn on or off an A.C. device)

```
01*LBL "4132"       09 SF 17          17 X<>F
02 CF 09            10 OUTA           18 STO 06
03 2                11 "msMMC'++"     19 PWRDN
04 RCL 06           12 FS? IND Y      20 XEQ "ACCHAR"
05 X<>F             13 "|dMM+"        21 TRIGGER
06 FC?C IND Y       14 FC? IND Y      22 GTO "RETURN"
07 SF IND Y         15 "|S]]+"        23 END
08 "+,Ui_~MrL+:+" 16 OUTA
```

```
SYNTHETIC TEXT LINES
 08:      254, 30, 44, 15, 85, 105, 95, 126, 77, 114,
         76, 76, 14, 58, 3.                "Device number"
 11:      248, 109, 115, 77, 77, 67, 39, 18, 3. "one is"
 13:      245, 127, 100, 77, 77, 3.              "on"
 15:      245, 127, 83, 93, 93, 3.               "off"
```



LINE 2
Clear Flag 9 so "ANSWER" won't try to print a phone message later on.

LINES 3-16
Flag status from R06 is recalled and Flag 2 is examined and flipped. It then says "Device number one is", followed by "on" or "off" depending on the status of Flag 2.

LINES 17-21
New flag status is stored back in R06 and output to control latch, thereby fulfilling the prophecy just spoken.

LINE 22
Going back to "RETURN" in the driver program terminates calling session.

"4111"
(This is the user code to read back your priority messages to you while you are away from your home or office.)

```
01*LBL "4111"
02 CF 09
03 "TIME"
04 RCLPTA
05 "iwmC+.++"
06 RCL 08
07 INT
08 ENTER^
09 ENTER^
10 11
11 -
12 SEEKPT
13 ARCLREC
14 SF 17
15 OUTA
16 "LB_K^+'++"
17 1
18 X=Y?
19 "LB_++++"
20 OUTA
21 RDN
22 X=0?
23 GTO "RETURN"
24 X<>Y
25 1
26 -
27 1 E3
28 /
29 11
30 +
31 STO 08
32*LBL 01
33 RCL IND 08
34 INT
35 "CLIENT"
36 SEEKPTA
37 GETREC
38 2
39 +
40 SEEKPT
41 ARCLREC
42 SF 17
43 OUTA
44 2
45 +
46 SEEKPT
47 GETREC
48 XEQ "READ1"
49 RCL IND 08
50 FRC
51 100
52 *
53 STO 01
54 SF 08
55 XEQ "TIMED"
56 ISG 08
57 GTO 01
58 11.999
59 STO 08
60 GTO "RETURN"
61 END
```

```
SYNTHETIC TEXT LINES
 05:      248, 105, 119, 109, 67, 27, 46, 15, 3.    "You have"
 16:      249, 76, 66, 95, 75, 94, 26, 39, 31, 3.   "messages."
 19:      247, 76, 66, 95, 11, 30, 26, 3.           "message."
```

LINE 2

CF 09 so the main routine won't try to print a message.

LINES 3-23

The system will say "You have (# of messages) messages". The number is based on the value of the loop counter stored in R08. The "TIME" ASCII file is needed so it knows how to say a number, and lines 17-19 are used to replace "messages" with "message" if only 1 message has been received. If the number of messages =0, subroutine exits.

LINES 24-31

Constructs an ISG loop control to know when to stop reading messages. Stored in R08.

LINES 32-43

Makes "CLIENT" the working file, takes the integer portion of the "compact message" (see "ANSWER" lines 130-147) and uses it as the record pointer. The phonetic form of the first and last names are put into ALPHA and spoken.

LINES 44-48

The phone number of the client is read from the client file into ALPHA, and READ1 is called which takes the ALPHA string of digits and reads them one at a time.

LINES 49-55

The time and day of week are then decoded from the compact message and spoken. Flag 8 at line 54 tells "TIMED" to recite the time and day found in the X register rather than the current time and day.

LINES 56-59

Loop back and read the rest of the messages. If they've all been read, reset the message index register (R08) to indicate no priority messages. This way if you call up a second time to see if more messages have arrived, you won't have to hear the original ones again.

LINE 60

Jump to "RETURN", the exit point of the main routine.

Program READ1 "reads" an alphanumeric string and pronounces everything. It is called from "4111" when reading the list of priority clients back to the user over the phone.

```
01*LBL "READ1"     13 X<>Y          25 FRC
02 ALENG           14 /             26 STO 03
03 ENTER^          15 STO 03        27 RDN
04 ENTER^          16 FS?C 10       28 INT
05 1000            17 XEQ "THANK"   29 SEEKPT
06 /               18 "TIME"        30 GETREC
07 1               19 RCLPTA        31 SF 17
08 +               20*LBL 07        32 OUTA
09 STO 04          21 RCL 03        33 ISG 04
10 X<>Y            22 10            34 GTO 07
11 10^X            23 *             35 END
12 ANUMDEL         24 ENTER^
```

"ACCHAR" is a handy program for "outputting" a decimal word without using the ALPHA register. It undoes the default status, outputs the word to the control latch (not the synthesizer), and then resets the previous status. The default status is:

AUTOIO	So both the printer and IL Conv. can be addressed without SELECT'ing each device.
Flag 21 Clear	So AVIEW's won't be printed.
TRIGGER last	Data gets routed to the speech chip by default.

Recommended usage: PWRDN, ACCHAR, TRIGGER.

```
01*LBL "ACCHAR"    04 ACCHR         07 END
02 SF 21           05 AUTOIO
03 MANIO           06 CF 21
```

"FLIP" toggles a bit in a control word without disturbing the other control bits.

```
01*LBL "FLIP"      05 SF IND Y      09 XEQ "ACCHAR"
02 RCL 06          06 X<>F          10 TRIGGER
03 X<>F            07 STO 06        11 END
04 FC?C IND Y      08 PWRDN
```

Finally, MSG2 is a subroutine which says, "Hello. This is Gary's Hewlett Packard. At the tone please leave your name, message, and telephone number." and returns. Rather than give a line-by-line analysis, I'll show the phonetic symbols of the text lines since I'm sure no one else will want to use this message without modification. The two routines at the end, "BYE" and "THANK", are called upon at other times to appease your callers.

```
01*LBL "MSG2"
02 SF 17
03 "[BX+m~~~~~yg_C"
04 OUTA
05 "'+~\A@k|iRC"
06 OUTA
07 "[iwXAjC+++:++"
08 OUTA
09 5
10 LN
11 E^X
12 LN
13 E^X
14 LN
15 E^X
16 "~~njCCysC++mMM~"
17 OUTA
18 "eXl++XliOC"
19 OUTA
20 ")5++M`aL+"
21 OUTA
22 5
23 LN
24 E^X
25 LN
26 E^X
27 "+++g^Z+++oMM^C"
28 OUTA
29 "++++/&|+MrLLN:+"
30 OUTA
31 RTN
32*LBL "BYE"
33 "+++NUiC"
34 SF 17
35 OUTA
36 RTN
37*LBL "THANK"
38 SF 17
39 "++++++<>+"
40 OUTA
41 END
```

```
Line 3:     H   EH1    L   O    W    -    -    -    -    -
           91   66   88 166  109  126  126  126  126  126

           TH    I    S   -
          121  103   95  67

Line 5:     I    Z    -
           39   18  126
```

```
             G   EH1   EH3   R   E1   Y   Z   -
            92   65    64   107 124  105 82  67

Line 7:     H   Y   U1   L   EH1   T   -
            91  105 119 88   65   106  67

             P   AE   K   ER   D   -
            165 174  153  58   30   3

Lines 9-15  Slight delay.

Line 16:     AE   T   -   -   TH   UH   -
            110  106  67  67  121 115  67

             T   O   W   N   N   -
            170 166 109 77  77  126

Line 18:     P   L   E   Z   -   L   E   Y   V   -
            101 88  108 146 131 88  108 105 79  67

Line 20:     Y   O1   R   -   N   A   AY   M   -
            41   53  43   3  77  96   97  76  190

Lines 22-26  Brief pause.

Line 27:     M   EH1   S   I   D   J   -   -   -
            140  130  159 103 94  90  190 190 190

            AE1   N   N   D   -
            111  77  77  94  67

Line 29:     T   EH1   L   EH2   F   O   N   -
            170  130  152  129  29  38  13   3

             N   UH1   M   M   B   ER   -
            77   114  76  76  78   58   3

Line 33:     G   OO   D   B   AH1   Y   -
            28   23  30  78   85   105  67

Line 39:     TH   AE   Y   N   K   -   Y   U   W   -
            185  174  169 141 153 131 41  40   45  3
```

Other useful information:

Register Usage:

R00 ISG register to determine how long the relay's been down.
R01 # of rings detected; timing loop for INTT4.
R02 Loop address of IL Converter.
R03 String to be read in READ1. (Temporary.)
R04 Looping control for READ1.
R05 Number of calls received.
R06 Last byte written to the control latch.
R07 Temporary location for ASTO'ing the 4-digit code.
R08 Message index register, for storing priority client's messages.
R11 and up: Priority client's messages.

Flag Usage:

0-7 Used with INSTAT.
8 Tells TIMED to read the time in X rather than the current time.
9 There exists valid INTT4 input.

Concluding Remarks

I believe the Austrian Emperor Joseph II summed it up pretty well when he said, "Well then...there it is!".

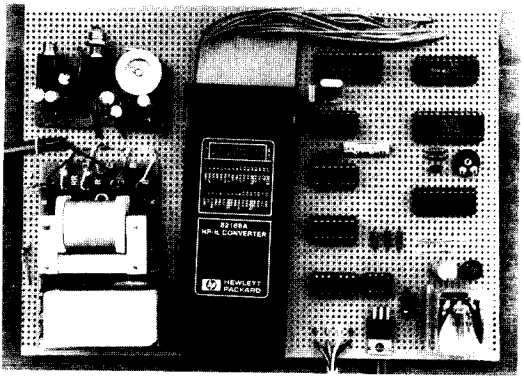


Photo of completed answering machine circuit. Hybrid transformer is shown in the lower left-hand corner.

Chapter Eight

KEYBOARDS FOR THE 71

"A Keyboard? How quaint!"

--Scotty

Described here is the first of two types of keyboards that compensate for one of the 71's drawbacks: namely, its own keyboard. The second type, which is easier to implement, tells how to convert an IBM (YECCH!) PC to a dumb terminal and is covered in Chapter 10.

But here we get more creative than just hooking up a dumb terminal. Nothing's more counterproductive than confining your 71 to a wall outlet (and consuming a billion times more power) just so you can enhance your productivity. In my quest to preserve portability, I have built some keyboards that not only run on batteries, but generate the weird escape sequences that the 71 expects to see as well. Your success at duplicating these efforts depends on your ability to scavenge at swap meets, surplus shops, and garage sales in order to find a suitably modifiable keyboard. And because every keyboard found will require its own unique solution to attach it to the 71, a thorough understanding of the principles described in this chapter will help you hook up any keyboard you happen to possess.

The first keyboard that this chapter will describe is the easiest type to hook up: the parallel-encoded kind. It is good for very basic work, but for serious development work further improvements are necessary, and these are described in the ADD EPROMS FOR ENHANCEMENTS section. Finally, the Nth degree of complexity (well, it may seem like it, anyway) is covered as I describe my favorite keyboard, the one originally designed for the Otrona

Attache computer.

The 'KEYBOARD IS' Lexfile

Before we get to the hardware, we must first know what the 71 expects to see so we can accommodate it as much as possible. Here we make use of HP's KEYBOARD IS lexfile, which is available either on magnetic cards from HP or in the FORTH/ASSEMBLER ROM. HP designed this Lexfile to allow larger terminals or computers (in conjunction with the DISPLAY IS command already resident in the 71) to act as the 71's keyboard and display. Used in this way, a programmer might be able to overcome the 71's human interface limitations and actually produce some code in reasonable time without raising his or her blood pressure.

Like most general-purpose interfaces, HP had obstacles to overcome to make this type of input work with all possible keyboards. Most 'smart' keyboards that generate their own characters adhere strictly to the ASCII standard, which defines how the characters A-Z,a-z,0-9,!-+ and the first 15 control characters are represented via 7-bit binary numbers. Unfortunately, all of the other keys such as cursors, functions, Tab, CAPS LOCK, and even backspace are not as rigidly defined; these can vary from manufacturer to manufacturer. The 71 represents these in its own individual fashion.

How is a normal keyboard supposed to send these crucial non-standard keystrokes? HP's solution was to insist that the user precede every special key with the ESCAPE keystroke. The 71 would then cross-reference any escape-preceded keystrokes with those already defined in an "escape" buffer and, if a match is found, "press" the predefined button. A sequence is defined in the escape buffer by entering a line which looks like this:

```
ESCAPE "A",50
```

After the above assignment is typed into the 71, hitting the external keyboard's ESCAPE key followed by the "A" key will activate key #50 on the 71; the Up Arrow key. (Refer to the 71's Keyboard map in their instruction manual for how to specify other keys.) In this manner, any 71 function (such as ATTN, Command Stack, I/R,

and the cursors) could be executed from the remote keyboard. As will be seen shortly, this method has its drawbacks.

The Parallel Keyboard

There are generally 3 different types of keyboards that are sold as surplus: The "matrix only" kind, which is a board full of switches without any support components (requires too much work; avoid this type); the "serial" kind, which has between 3 and 5 wires in it's host-going cable and sends the information like an RS-232 link; and the parallel kind, which is by far the easiest to hook up since it transfers data the same way as the GPIO: 8 bits at a time with a 9th line acting as a strobe.

Consider the parallel keyboard presented below. This particular keyboard has 10 labelled function keys, special characters, a numeric keypad, and an excellent keyfeel, making blind tying a cinch.

Hooking up a parallel-encoded keyboard is exactly like attaching a printer to the GPIO, only the data goes in the other direction. Parallel keyboards are already programmed, off-the-shelf, to generate ASCII characters when the appropriate keys are pressed. Therefore, connecting it is as simple as the drawing in Figure 8-2:

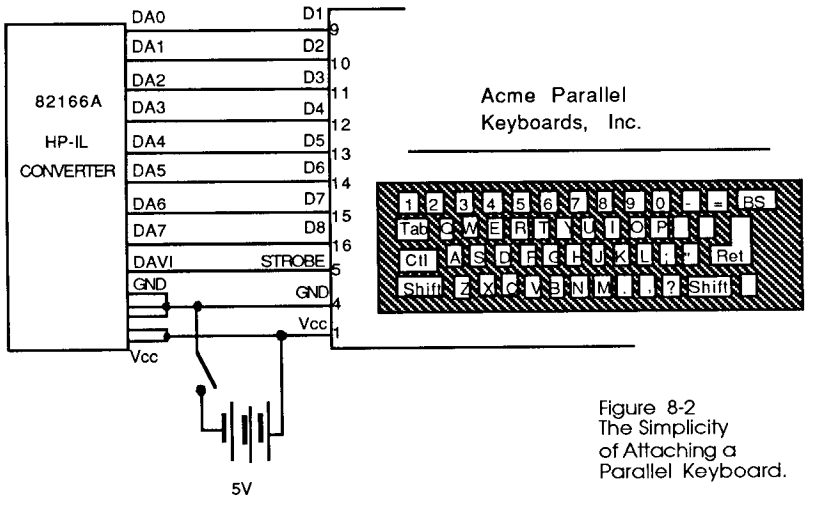


Figure 8-2
The Simplicity
of Attaching a
Parallel Keyboard.

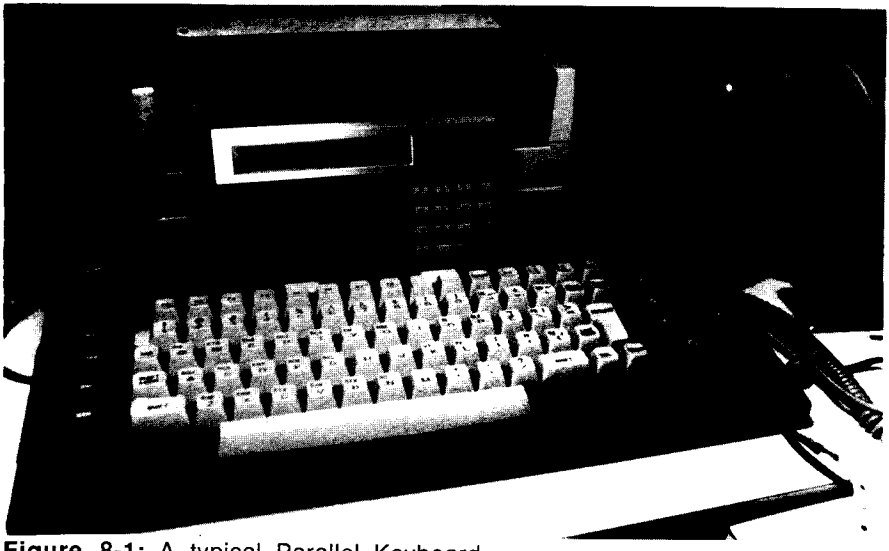


Figure 8-1: A typical Parallel Keyboard.

and the software commands are as simple as:

```
10 A=DEVADDR("GPIO")
20 SEND MTA LISTEN A DDL 0 DATA 226,16,24 UNL UNT
30 KEYBOARD IS :GPIO
```

Obviously, a few key words (again, no pun intended) of explanation are in order here. If the KEYBOARD IS routine is to function efficiently, the GPIO must be modified to flag service requests whenever there is outgoing data in its transfer buffer, and to send End-of-Data frames when the buffer is empty. Both these tasks are accomplished by the DDL 0 command of line 20 above, which sets the following attributes:

226--GPIO will send service requests when any computer-bound data is waiting in the buffer, in addition to any normal status request.

16---Send an End-Of-Data frame if the buffer's empty.

24---Strobed handshake, positive data and negative handshake logic.

Every time the GPIO is powered up, the configuration program must be run (but you already knew that!). To make things easy, I include in this program all the ESCAPE buffer definitions, although these really only have to be run once.

While defining the ESCAPE buffer definitions, another problem (for this keyboard, anyway) came up: How can I define a non-standard keystroke in the ESCAPE buffer if I don't know what decimal byte the keyboard sends out?

I answered this question by putting my 41 on the loop and, using a wonderful feature of the IL Development ROM, put it into SCOPE mode. This mode lets you look at individual HP-IL frames as they pass through the loop. (It also allows you to save these frames in a buffer for future observation, generate custom packets of your own, watch for interrupts, and a host of other tasks.) Anyway, every time I hit a key of unknown keycode, I'd watch the 41's display for the decimal byte, embedded in each DAB frame. These numbers were then used in the CHR$()'s needed for the ESCAPE buffer definitions in the program below, which configures the GPIO as well as defining the escape sequences:

```
5 ! Program KEYBD implements KEYBOARD IS on the
    Microswitch keyboard.
10 RESTORE IO @ RESET HPIL
20 KEYBOARD IS *
30 A=DEVADDR("GPIO")
40 SEND MTA LISTEN A DDL 0 DATA 226,16,24 UNL UNT
50 SFLAG -15
60 KEYBOARD IS :GPIO
70 ESCAPE CHR$(144),50      ! assigns up arrow key
80 ESCAPE CHR$(145),51      ! assigns down arrow
90 ESCAPE CHR$(147),103     ! assigns backarrow to
                            ! destructive backspace.
100 ESCAPE CHR$(146),48     ! assigns right arrow.
110 ESCAPE CHR$(9),43       ! assigns tab key to ATTN.
120 ESCAPE CHR$(238),43     ! assigns SHIFT TAB to
                            ! ATTN.
130 ESCAPE CHR$(99),104     ! assigns esc 'c' to -CHAR.
```

```
140 ESCAPE CHR$(67),104      ! assigns 'C' to -CHAR.
150 ESCAPE CHR$(169),162     ! assigns left blank key
                             ! to all the way up.
160 ESCAPE CHR$(185),162     ! assigns shift left blank
                             ! key to all the way up.
170 ESCAPE CHR$(163),163     ! assigns right blank key
                             ! to all the way down.
180 ESCAPE CHR$(179),163     ! assigns shift right
                             ! blank key to all the way
                             ! down.
190 ESCAPE CHR$(171),150     ! assigns home to command
                             ! stack.
200 ESCAPE CHR$(187),150     ! assigns shift home to
                             ! command stack.
210 ESCAPE CHR$(8),159       ! assigns back space to
                             ! far left.
220 ESCAPE CHR$(237),159     ! assigns shift backspace
                             ! to far left.
230 ESCAPE CHR$(164),160     ! assigns back tab to far
                             ! right.
240 ESCAPE CHR$(180),160     ! assigns shift back tab
                             ! to far right.
250 ESCAPE CHR$(108),107     ! assigns esc 'l' to
                             ! -LINE.
260 ESCAPE CHR$(76),107      ! assigns esc 'L' to
                             ! -LINE.
270 ESCAPE CHR$(215),106     ! assigns F7 to LC.
280 ESCAPE CHR$(231),106     ! assigns shift F7 to LC.
290 ESCAPE CHR$(216),105     ! assigns F8 to I/R.
300 ESCAPE CHR$(232),105     ! assigns shift F8 to I/R.
310 CONTRAST 15
320 END
330 SUB KEYOFF
340 KEYBOARD IS *
350 CFLAG -15
360 CONTRAST 9
370 BYE @ END SUB
```

You might notice some other additions, too: FLAG -15 (lines 50 & 350) switches to lowercase mode, and the KEYOFF subroutine at line 330 (activated by command CALL KEYOFF) restores the

changed parameters when finished.

Other Potential Problems

In theory, parallel keyboards are the easiest to hook up and use. In reality though, as we have just seen, differences in individual keyboards can cause problems in implementation. Just like the non-ASCII keys producing arbitrarily defined output, there also exists the problem of inadequate pulse widths coming from the keyboard's STROBE line.

In the past, we dealt with inadequate handshake pulse widths with something called a pulse expander, which will takes any length pulse as input and outputs a nice, long 100ms pulse width so the GPIO will accept it.

The specs of my particular keyboard state that the pulse width it generates lasts anywhere from 10 to 90 milliseconds, which means there exists about a 63 to 37 chance of NOT meeting the GPIO's requirements and losing your data. This necessitates the additional circuitry shown in Fig. 8-3:

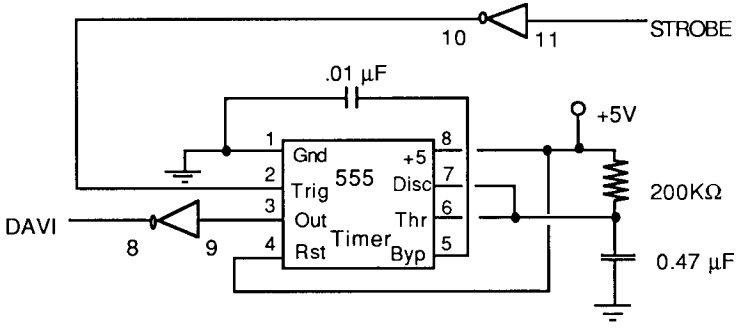


Figure 8-3
Pulse
Expander

This circuit is identical to the one introduced in Chapter 4, and is necessary because if the pulse width is less than 60 ms, the

GPIO thinks no keys have been pressed and the whole system will quietly ignore you.

Now that the parallel keyboard has been hooked up and the pulse expander (if it was needed) has been added and tested, all one has to do is just power up the keyboard (& the GPIO, too!), run the setup program, and type away! Notice how much faster you can tell the 71 what to do! Notice the 32-character "type-ahead" feature, thanks to the GPIO's 32-character transfer buffer.

There are, regretfully, drawbacks. The keyboard's ESCAPE key, unlike the more-famous CONTROL key, must be pushed and then released before the next key depression. This means that each Backarrow (non-destructive backspace) requires 2 keystrokes: ESC and <--. When editing text files, this can be a very frustrating procedure. Specially defined function keys and those important cursor keys share the same frustrating qualities. These seemingly trivial drawbacks are so constraining that the rest of this chapter will be spent explaining a method of overriding them.

Add EPROMs For Enhancements (Just What The 71 Wants!)

If the 71 expects to see all useful non-alphanumeric functions presented as 2-character ESCAPE sequences, wouldn't life be wonderful if we had a keyboard that generated 2 such characters for every keystroke? (It was a rhetorical question; the answer is

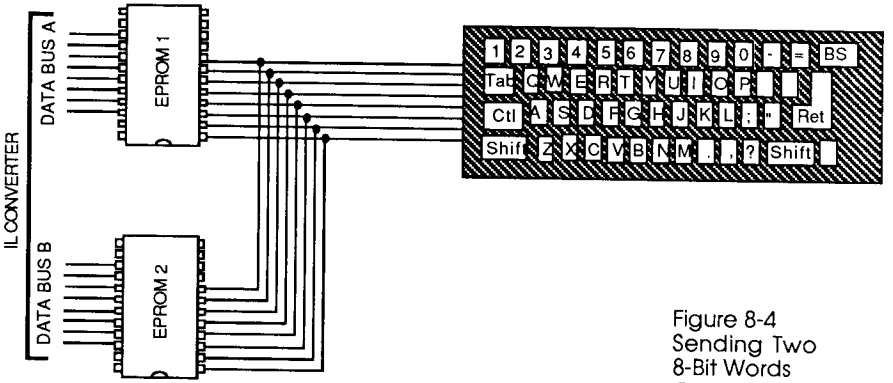


Figure 8-4
Sending Two
8-Bit Words
On Every
Keystroke.

"Yes".)

Through the use of a few extra parts, including some custom 2716 EPROMS, we can take any parallel keyboard and turn it into a "custom made" job, where all key definitions (especially the cursors) can all take place with a single keystroke!

This method uses a little trick to generate two 8-bit words for every single keystroke. Examine the diagram in Figure 8-4. Each time a key is pressed, a unique ASCII code appears on the 8 data lines. Normally, this code is fed directly to the GPIO; but this time we're treating it as an address to the two EPROMs. The EPROMs, functioning as an electronic lookup table, supply a 16-bit word to the GPIO, which has previously been configured to have a 16-bit word size. This means that for every 8 bits that come in, we get 16 user-specified bits coming out.

You can probably guess what happens next. The GPIO sends one 16-bit word to the 71, but the 71, expecting input in the form of 8-bit words, takes this double-sized word and treats it as 2 CONSECUTIVE bytes.

Let's take an example. If the right-arrow key is pressed on my particular keyboard, the on-board logic generates a decimal byte of 146 on the 8 data lines, and then pulses the STROBE line momentarily. The EPROMS immediately assume this is an address (since, after all, these data lines <u>are</u> being fed to their address lines), and generate a pre-programmed 16 bit word, Hex 1B 2A, to the GPIO. Because the KEYBOARD IS function in the 71 only expects to see 8-bit words, it treats these 16 bit words as 2 words: 1B (CHR$(27)) which is Escape, and 2A (CHR$(42)) which is an arbitrary ASCII character. The KEYBOARD IS escape key buffer has been told beforehand using the ESCAPE command that ESCAPE 2A should be interpreted as Right-arrow, so then it performs that function. If the Right-arrow key is held down, the repeat function takes effect and multiple escape sequences are generated, all with one key!

In the case of sending out the letter 'A', no escape sequence is needed. The two hexadecimal bytes sent are FF 41, where FF is a dummy byte which is (luckily) completely ignored by the 71, and 41 is the ASCII code for 'A'. Although twice as many characters per keystroke are being sent with this method as compared to a normal

keyboard, there is no noticeable decrease in speed.

Because every keyboard will generate different sequences for cursors, and because you may wish to define unique key sequences to activate the 71's functions, a custom set of EPROMs must be developed for each individual application. It is not a difficult thing to do; I would imagine the hard part would be for the average user to get hold of an EPROM burner. Deciding how to map out the EPROMs is an easy task. Just make a list (and check it twice):

WHAT I WANT TO PRESS	WHAT I WANT IT TO DO
'DEL'	ATTN
'LINE FEED'	COMMAND STACK
'TAB'	I/R
CNTRL-C	-CHAR
CNTRL-L	-LINE
'A'	A
'a'	g-A (shifted)

Then determine what decimal bytes the keyboard normally generates when the keys in the left column are pressed. These will become the EPROM addresses. Then go to the right column, and specify what you want the 71 to see when the key on the left is pressed. For example, when the 'A' key is pressed, we actually want the 71 to receive an 'A', so we feed it FF 41. (FF is a dummy byte used when no escape sequence is desired. 41 is the hexadecimal ASCII code for 'A'.) Another example: Hitting the 'DEL' key produces a decimal byte of 127; so that goes into the left column. We want to generate an escape sequence for this one: 1B (ESCAPE), CB (arbitrary character). This goes into the right column. We're not done yet! Any escape sequence must also be defined in the ESCAPE buffer in order to be reassigned by the 71. So we must now also add 1 line to the configuration routine:

```
ESCAPE CHR$(127),43
```

which tells the 71 to press its ATTN key (key #43) every time it encounters the sequence ESCAPE CHR$(127) (in hex, it looks like

1B 7F).

When filling out your map, be sure to cover every possible key sequence; both shifted and non-shifted letters, control- , shift- , and control-shift- . Every hex address (every possible keyboard output) should generate some sort of code, even if you think you'll never use it. (A sample of a complete EPROM map is shown later in this chapter.)

Matrix-Only and Serial Keyboards

Earlier I said not to bother with these, and now I'm going to go back on my own advice and discuss them at length. You see, my favorite keyboard in the whole world had characteristics of both the above boldfaced adjectives. This keyboard (when properly modified) works with the 71 so nicely, and it has just the right size and power consumption, that not to discuss it would be a disservice!

Most keyboards you're likely to find are of the matrix-only type, as shown in Fig. 8-5.

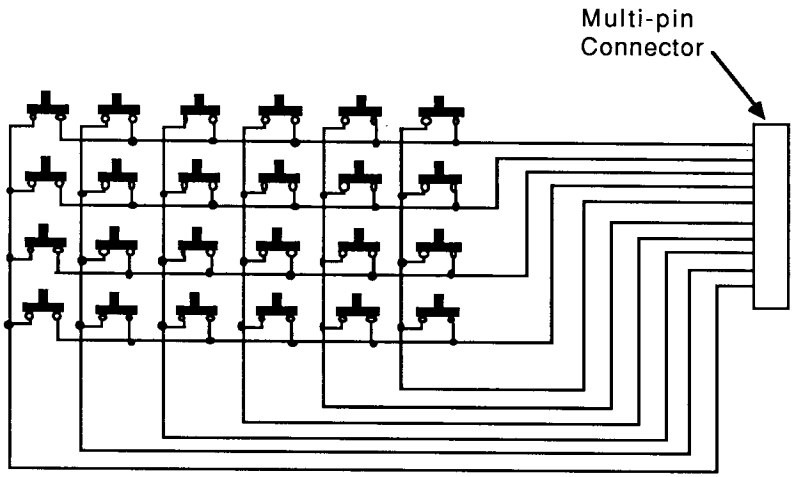
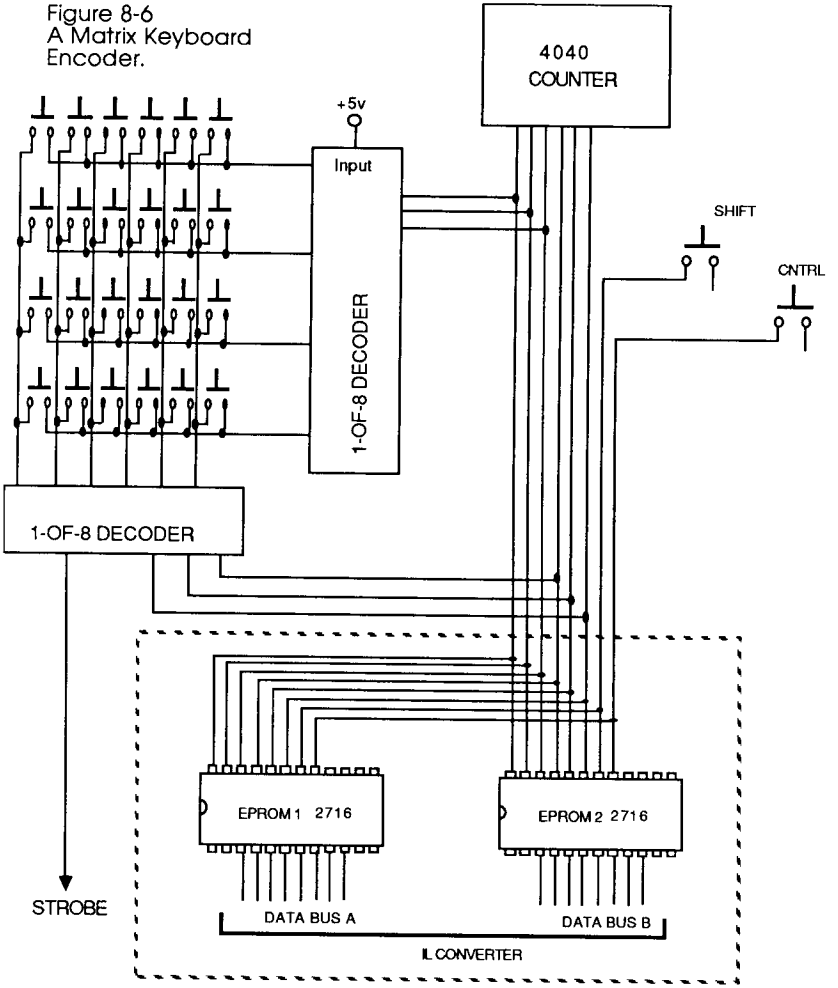


Figure 8-5
A matrix keyboard.

As can be seen, pressing a key connects one of the row lines to one of the column lines, and it requires additional circuitry to detect which key was pressed and to translate it into an ASCII character. The previously described parallel keyboard had such circuitry already on board; and therefore was easier than π to hook up.

If you do find such a keyboard (you'll know when you have it; there are no electronic components at all), Fig. 8-6 shows an example of what is needed to decode it:

Figure 8-6
A Matrix Keyboard
Encoder.



This simplified setup works as follows: the keyboard matrix has two 1-of-8 decoders driving its rows and columns. These decoders are designed to route a signal to any of eight different places, depending on the status of its three address lines. In the case of the vertical decoder, it takes the +5v from the input marked "x" and, depending on the state of its three address lines, redirects it to one of the four wires connected to the rows. The bottom decoder sends its information the other way: it scans the six columns of the keyboard matrix and sequentially routes any signals it finds to the "x" pin emanating from the bottom. So in normal operation, the highest bits from the eternally-running counter cause the 5v at "x" to sequentially appear at each of the rows, and the lowest three bits allow the status of each column to sequentially appear at the "x" of the column decoder.

If a key has been depressed, eventually the counter will generate the proper code to let the 5v from the row decoder reach the "x" output of the column decoder, which we rename as STROBE. This tells the world, "Hey! Somebody's pressed a key, and if you look at the counter's bits right now, you'll see a bit pattern which uniquely defines the pressed key!". The two EPROMS at the bottom are, in fact, doing just this. They are constantly translating this unique keycode, as well as the status of the SHIFT and CONTROL keys, into ASCII code. If a GPIO were to look at this system from the bottom, it would see 16 data lines and a strobe. And when the strobe goes to +5v (= "1"), a 2-byte escape sequence appears at the output!

The World's Best Keyboard

If you thought that was complicated, take a look at Fig. 8-7 (next page), which shows the detailed schematic implementing this keyboard scanning method without the EPROMS. It is easy to see that, even though a method exists to transform a matrix-only keyboard into something that can work with the 71, it is really a great deal of trouble.

Imagine my joy when I stumbled upon a tiny keyboard which already had the necessary circuitry built in! These schematics actually describe the keyboard from an old Otrona Attache, a

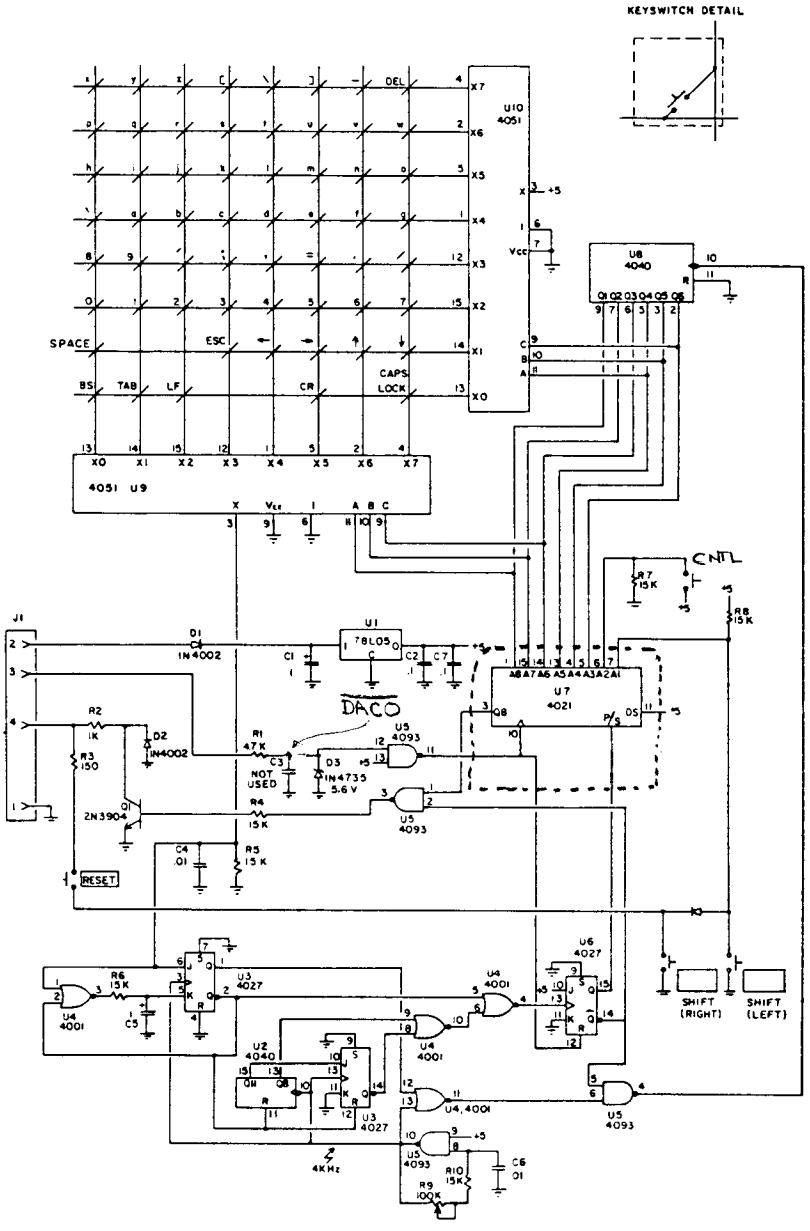


Figure 8-7: Schematic for the Otrona Keyboard

superb portable computer whose key feel and size I had always admired. (It's too bad the firm folded; it wasn't for lack of a superior product.) The keys feel wonderful, and all the circuitry is constructed in power-saving CMOS. The only problem was that this was a serial keyboard; the unique keycode goes through a parallel-to-serial converter before being shipped to the host computer. (In this instance, this setback is easily remedied by removing said serial converter, as described later.)

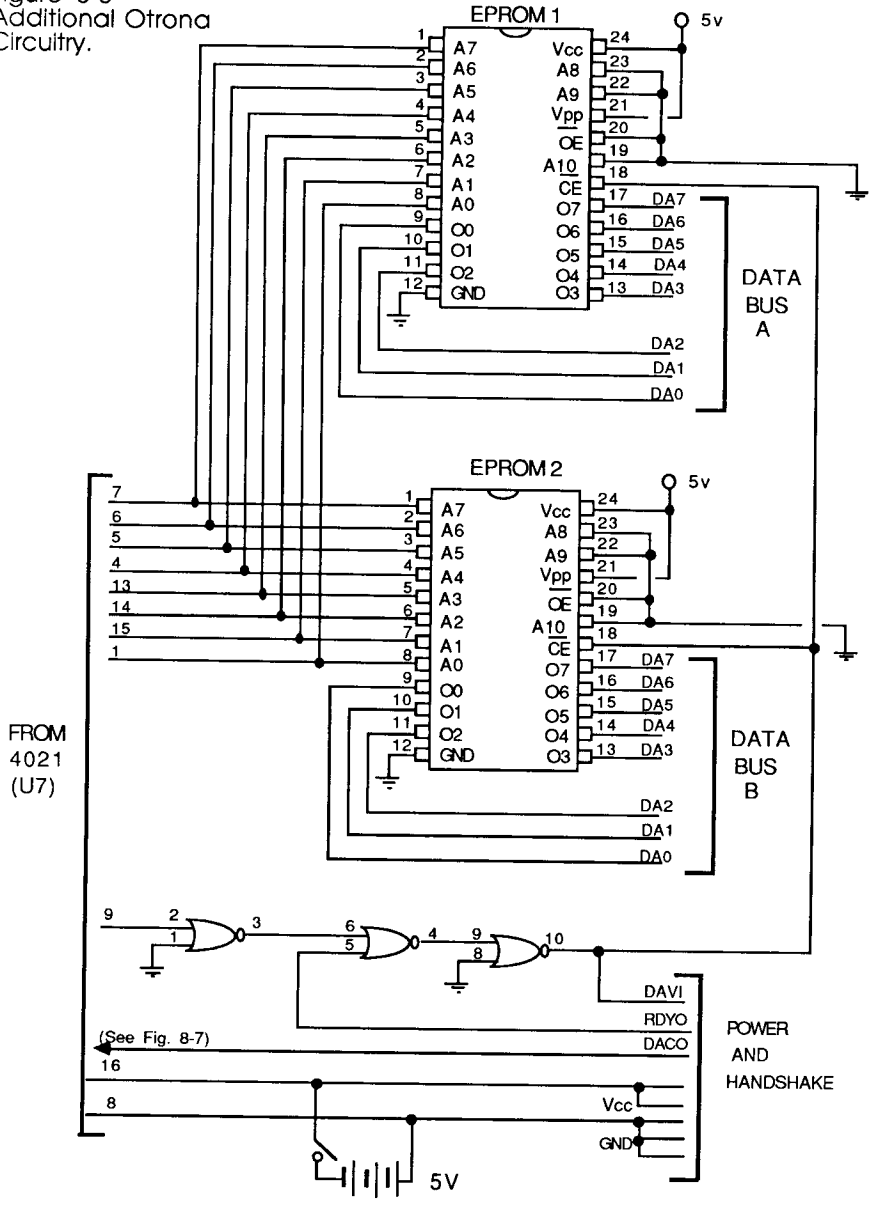
Without going into great detail about how the gates and flip-flops work, the components function as follows: All the components on the bottom of the diagram form a clock, whose output goes directly to pin 10 (the input) of the 4040 CMOS counter chip. The keyboard scanning behaves exactly as described above, except the counter's output is fed not to two EPROMs, but to a 4021 parallel-to-serial converter!

Here, then, is what to do in order to interface with a serial keyboard: simply remove the parallel-to-serial converter from the circuit board. We are now back to parallel again. (Wasn't that easy?)

The dashed box around the 4021 IC represents the component to be removed and all the wires normally attached to it. In this application the box is removed and replaced with the two EPROMs and three gates as shown if Fig. 8-8. These parts take the keycode as input and output two parallel 8-bit ASCII characters, complete with the required handshake.

Implementing this scheme requires slight modification of the Otrona keyboard, whose components are well labeled. First, transistors Q1 and U1, the only two on the circuit board, should be removed to cut current consumption. Next, U7, the parallel-to-serial converter described above, must be de-soldered and removed from the board. Jumper wires replace this IC and carry signals over to the EPROM's address inputs as in Fig. 8-6 and detailed in Fig. 8-8. Pin 9 from U7 is used as a handshake signal. It is ANDed with the RDYO signal from the GPIO. (Notice the use of NOR gates for this function to reduce the chip count.) The third handshake line, DACO, is connected to resistor R1 as shown in the schematic. This way we make use of the otherwise unused NAND gate (configured as an inverter) for positive

Figure 8-8
Additional Otrona
Circuitry.

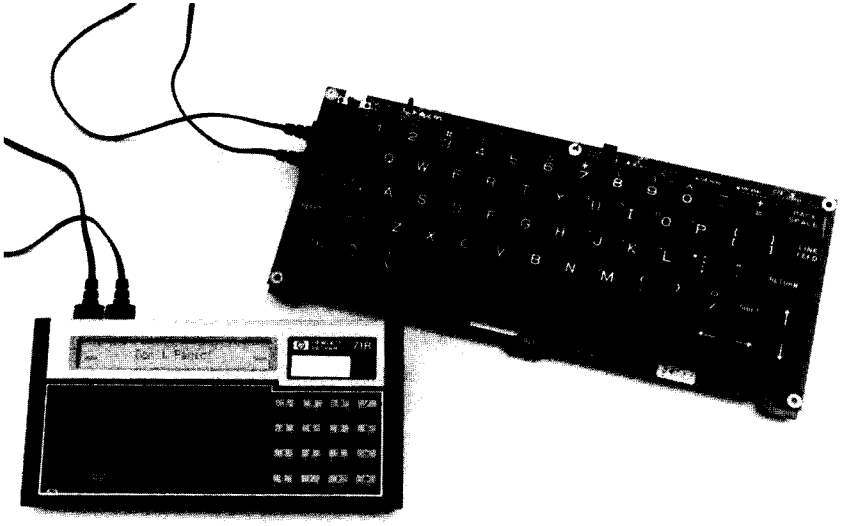


handshake logic.

There's also a potentiometer, R9, which is used to control the speed of the repeat function. I recommend slowing it down to match the receiving speed of the 71, so the cursor doesn't move forever when the key is released.

The configuration program below performs exactly the same functions as the one presented earlier, except it is customized for the Otrona keyboard. (See the end of this chapter for a list of the Otrona's EPROMs.)

```
10 ! "OTRONA" setup program for Otrona Keyboard.
20 RESET HPIL @ RESTORE IO
30 KEYBOARD IS *
40 A=DEVADDR("GPIO")
50 SEND MTA LISTEN A DDL 0 DATA 226,16,30 UNL UNT
60 SFLAG -15
70 KEYBOARD IS :GPIO
80 ESCAPE CHR$(41),50  ! Assigns up arrow key
90 ESCAPE CHR$(40),51  ! Assigns down arrow key
100 ESCAPE CHR$(42),48 ! Assigns right arrow key
```



The world's best keyboard.

```
110 ESCAPE CHR$(43),47    ! Assigns left arrow key
120 ESCAPE CHR$(203),43   ! Assigns DEL key to ATTN
130 ESCAPE CHR$(194),162  ! Assigns shift up key to all
                          ! the way up
140 ESCAPE CHR$(226),162  ! Assigns LC shift up key to
                          ! all the way up
150 ESCAPE CHR$(193),163  ! Assigns shift down key to
                          ! all the way down
160 ESCAPE CHR$(225),163  ! Assigns LC shift down key
                          ! to all the way down.
170 ESCAPE CHR$(10),150   ! Assigns line feed to
                          ! Command stack.
180 ESCAPE CHR$(196),159  ! Assigns shift left to far
                          ! left.
190 ESCAPE CHR$(228),159  ! Assigns LC shift left to
                          ! far left.
200 ESCAPE CHR$(195),160  ! Assigns shift right to far
                          ! right.
210 ESCAPE CHR$(227),160  ! Assigns LC shift right to
                          ! far right.
220 ESCAPE CHR$(76),107   ! Assigns CNTL L to -LINE.
230 ESCAPE CHR$(108),107  ! Assigns LC CNTL l to -LINE.
240 ESCAPE CHR$(67),104   ! Assigns CNTL C to -CHAR.
250 ESCAPE CHR$(99),104   ! Assigns LC CNTL c to -CHAR.
260 ESCAPE CHR$(202),106  ! Assigns CAPS LOCK to LC.
270 ESCAPE CHR$(234),106  ! Assigns LC CAPS LOCK to LC.
280 ESCAPE CHR$(9),105    ! Assigns TAB to I/R.
290 ESCAPE CHR$(33),103   ! Assigns backspace to
                          ! destructive backspace.
300 ESCAPE CHR$(83),102   ! Assigns CNTRL S to SST.
310 ESCAPE CHR$(96),78    ! Assigns 'Back Apostrophe'
                          ! to that character using
                          ! text editor's keys.
320 ESCAPE CHR$(126),80   ! Assigns Shift Tilda to same
                          ! character using text
                          ! editor's keys.
330 ESCAPE CHR$(92),92    ! Assigns backslash to same
                          ! character using text
                          ! editor's keys.
340 ESCAPE CHR$(124),93   ! Assigns Shift "two vertical
                          ! segments" to same chars
```

```
                                    ! using text editor's keys.
350 DELAY .5,.05
360 END
370 SUB KEYOFF
380 KEYBOARD IS *
390 CFLAG -15
400 BYE @ END SUB
```

Mapping Out the EPROMS

Probably the most complex and tedious task about converting an arbitrary keyboard into a 71-compatible one is figuring out what the EPROMs should contain. To that end, I have written a small program which generates a form you can fill out. It was originally written for the Otrona keyboard, which the SHIFT and CONTROL prefixes being inserted into predictable places. Just attach a ThinkJet (or other 80-column printer) to the 71, enter a "PRINTER IS PRINTER", and run the following program:

```
10 ! PROGRAM FORM
20 ! PRINTS TABLE FOR MAPPING OUT OTRONA KEYBOARD.
30 STD
40 CALL HEADER
50 FOR Y=1 TO 255
60 PRINT TAB(6); ":"; TAB(14); ":"; TAB(29); ":";
   TAB(52); ":";      TAB(66); ":"; TAB(73); ":";
   TAB(80); ":"
70 ENDLINE ""
80 PRINT Y; TAB(6); ": "; DTH$(Y); TAB(2); ": ";
90 ENDLINE
100 LET B=Y
110 CALL BINARY(B)
120 PRINT TAB(17); ": ";
130 IF FLAG(2,0)=1 THEN PRINT "CNTRL ";
140 IF FLAG(1,0)=0 THEN PRINT "SHIFT ";
150 PRINT TAB(40); ":   "; TAB(54); ":"; TAB(61); ":";
    TAB(68);       ":";
160 PRINT
170 PRINT TAB(6); ":"; TAB(14); ":"; TAB(29); ":";
     TAB(52); ":";        TAB(66); ":"; TAB(73); ":";
```

```
TAB(80); ":"
180 PRINT
"-------------------------------------------------------------
---------------------------"
190 IF MOD(Y,15)=0 THEN PRINT CHR$(12) @ CALL HEADER
200 NEXT Y
210 END
220 SUB BINARY(B)
230 A$="#,D"
240 CFLAG 1 @ CFLAG 2
250 FOR X=7 TO 0 STEP -1
260 PRINT USING A$; INT(B/2^X);
270 IF X=7 AND RES=1 THEN SFLAG 1
280 IF X=6 AND RES=1 THEN SFLAG 2
290 B=MOD(B,2^X)
300 IF X>5 THEN PRINT USING "#,X";
310 NEXT X
320 END SUB
330 SUB HEADER
340 PRINT "DEC      HEX      BINARY            KEY
    SEQUENCE                CHARS SENT    ROM 1   ROM 2"
350 END SUB
```

A sample of the program's output appears on the opposite page.

The first three columns represent the eight bits generated by the keyboard in decimal, hexadecimal, and binary. Notice the two most significant bits in the binary column have been separated; this was to help me visually see the effect of the SHIFT and CONTROL keys of the Otrona keyboard, which are directly responsible for those two bits. Capitalizing on this, I had the program automatically fill in SHIFT if bit 8 was zero (that's the effect when the SHIFT key is pressed), and CNTRL when bit 7 was one (what happens when the CONTROL key is pressed).

To fill in the table, first press any key on the keyboard and notice its output. (One easy way to do this is to put a 41 in SCOPE mode into the loop and watch the data go around; another way is to program the 71 to read in a byte and display it.) Match this output with an entry in the left columns; use whichever of the decimal, hex, or binary columns are appropriate, and mark in the "Key

Sequence" column the key you pressed to generate this table entry. Next, ask yourself "What character(s) do I want to generate whenever I press this key?". If you pressed the letter "a", you probably want the letter "a" to be sent. Mark this in the "Characters Sent" column. A quick check of the ASCII table in Appendix F reveals that the character "a" is represented by hex value 61, and this value goes in the rightmost box, labeled "ROM 2". The "ROM 1" column is filled in with hex FF since the "a" is all we want. (If we want to send an ESCAPE sequence, it is filled with a hex 1B.)

Another example: Let's say we want the DEL key to act as the ATTN key on the 71. First, press the DEL key on the keyboard and find out what key code it generates. Find this in one of the first three columns of the EPROM map, and enter "DEL" as the Key Sequence. This time, we want an arbitrary ESCAPE sequence to be sent, so in the Chars Sent column, we enter two characters:

DEC	HEX	BINARY	KEY SEQUENCE	CHARS SENT	ROM 1	ROM 2
1	00001	0 0 000001	SHIFT			
2	00002	0 0 000010	SHIFT			
3	00003	0 0 000011	SHIFT			
4	00004	0 0 000100	SHIFT			
5	00005	0 0 000101	SHIFT			

ESCape, and decimal 203 (hex CB), an arbitrary character. This translates to the hex byte 1B in the "ROM 1" column, and the hex byte CB in the "ROM 2" column. Since this is an ESCAPE sequence, we must also work this into the configuration program so the 71 will know to translate this specific code into the ATTN key. So we add this line to the configuration program:

```
ESCAPE CHR$(203),43 ! Assigns DEL key to ATTN
```

It is useful to fill in every possible key sequence. Even if you have no intention of using, say, the CNTL-SHIFT-TAB combination, program in an escape sequence there so you may make use of it later in the configuration program.

If you are trying to map out a keyboard other than the Otrona, the automatically-included SHIFT and CNTRL keys will no doubt get in your way, since your keyboard's SHIFT and CNTL key combinations probably produce very different output. To remove them from the printed list, delete lines 130 and 140 from the FORM program listed above.

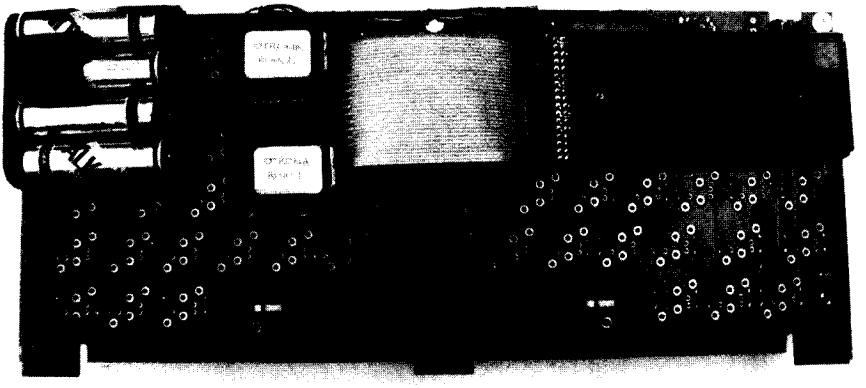
My Own EPROM Map

Well, here it is finally. This is the EPROM map I produced to go with the Otrona keyboard. This, in conjunction with the "Otrona" program listed above, turns this innocent looking keyboard into the world's best for the 71. (What makes it so good? The size and keyfeel are wonderful! It's CMOS type integrated circuits consume so little power that 4 AA batteries will power the thing for 7 months!)

HEX KEYCODE	KEY SEQUENCE	ROM 1	ROM 2
00	SHIFT BS	FF	08
01	SHIFT TAB	1B	C5
02	SHIFT LF	1B	C6
05	SHIFT RETURN	FF	0D
07	SHIFT CAPS LOCK	1B	C7
08	SHIFT SPACE	FF	20
0B	SHIFT ESC	FF	1B
0C	SHIFT Left Arrow	1B	C4
0D	SHIFT Right Arrow	1B	C3
0E	SHIFT Up Arrow	1B	C2
0F	SHIFT Down Arrow	1B	C1
10	SHIFT 0	FF	5E
11	SHIFT 1	FF	21
12	SHIFT 2	FF	40
13	SHIFT 3	FF	23
14	SHIFT 4	FF	24
15	SHIFT 5	FF	25
16	SHIFT 6	FF	26
17	SHIFT 7	FF	2A
18	SHIFT 8	FF	28
19	SHIFT 9	FF	29
1A	SHIFT '	FF	22
1B	SHIFT ;	FF	3A
1C	SHIFT ,	FF	3C
1D	SHIFT =	FF	2B
1E	SHIFT .	FF	3E
1F	SHIFT /	FF	3F
20	SHIFT `	1B	7E
21	SHIFT A	FF	41
22	SHIFT B	FF	42
23	SHIFT C	FF	43
24	SHIFT D	FF	44
25	SHIFT E	FF	45
26	SHIFT F	FF	46
27	SHIFT G	FF	47
28	SHIFT H	FF	48
29	SHIFT I	FF	49
2A	SHIFT J	FF	4A
2B	SHIFT K	FF	4B
2C	SHIFT L	FF	4C
2D	SHIFT M	FF	4D
2E	SHIFT N	FF	4E
2F	SHIFT O	FF	4F
30	SHIFT P	FF	50
31	SHIFT Q	FF	51

32	SHIFT R	FF	52
33	SHIFT S	FF	53
34	SHIFT T	FF	54
35	SHIFT U	FF	55
36	SHIFT V	FF	56
37	SHIFT W	FF	57
38	SHIFT X	FF	58
39	SHIFT Y	FF	59
3A	SHIFT Z	FF	5A
3B	SHIFT [	FF	7B
3C	SHIFT \	1B	7C
3D	SHIFT]	FF	7D
3E	SHIFT -	FF	5F
3F	SHIFT DEL	1B	C8
40	CNTRL SHIFT BS	1B	88
41	CNTRL SHIFT TAB	1B	89
42	CNTRL SHIFT LF	1B	8A
45	CNTRL SHIFT RETURN	1B	0D
47	CNTRL SHIFT CAPS LOCK	1B	CC
48	CNTRL SHIFT SPACE	1B	20
4B	CNTRL SHIFT ESC	1B	9B
4C	CNTRL SHIFT Left Arrow	1B	CD
4D	CNTRL SHIFT Right Arrow	1B	CE
4E	CNTRL SHIFT Up Arrow	1B	CF
4F	CNTRL SHIFT Down Arrow	1B	D0
50	CNTRL SHIFT 0	1B	B0
51	CNTRL SHIFT 1	1B	B1
52	CNTRL SHIFT 2	1B	B2
53	CNTRL SHIFT 3	1B	B3
54	CNTRL SHIFT 4	1B	B4
55	CNTRL SHIFT 5	1B	B5
56	CNTRL SHIFT 6	1B	B6
57	CNTRL SHIFT 7	1B	B7
58	CNTRL SHIFT 8	1B	B8
59	CNTRL SHIFT 9	1B	B9
5A	CNTRL SHIFT '	1B	A7
5B	CNTRL SHIFT ;	1B	BB
5C	CNTRL SHIFT ,	1B	AC
5D	CNTRL SHIFT =	1B	BD
5E	CNTRL SHIFT .	1B	AE
5F	CNTRL SHIFT /	1B	AF
60	CNTRL SHIFT `	1B	E0
61	CNTRL SHIFT A	FF	01
62	CNTRL SHIFT B	FF	02
63	CNTRL SHIFT C	FF	03
64	CNTRL SHIFT D	FF	04
65	CNTRL SHIFT E	FF	05

66	CNTRL SHIFT F	FF	06
67	CNTRL SHIFT G	FF	07
68	CNTRL SHIFT H	FF	08
69	CNTRL SHIFT I	FF	09
6A	CNTRL SHIFT J	FF	0A
6B	CNTRL SHIFT K	FF	0B
6C	CNTRL SHIFT L	FF	0C
6D	CNTRL SHIFT M	FF	0D
6E	CNTRL SHIFT N	FF	0E
6F	CNTRL SHIFT O	FF	0F
70	CNTRL SHIFT P	FF	10
71	CNTRL SHIFT Q	FF	11
72	CNTRL SHIFT R	FF	12
73	CNTRL SHIFT S	FF	13
74	CNTRL SHIFT T	FF	14
75	CNTRL SHIFT U	FF	15
76	CNTRL SHIFT V	FF	16
77	CNTRL SHIFT W	FF	17
78	CNTRL SHIFT X	FF	18
79	CNTRL SHIFT Y	FF	19
7A	CNTRL SHIFT Z	FF	1A
7B	CNTRL SHIFT [	1B	DB
7C	CNTRL SHIFT \	1B	DC
7D	CNTRL SHIFT]	1B	DD
7E	CNTRL SHIFT -	1B	AD
7F	CNTRL SHIFT DEL	1B	C9



Mounting the components onto the Otrona keyboard.

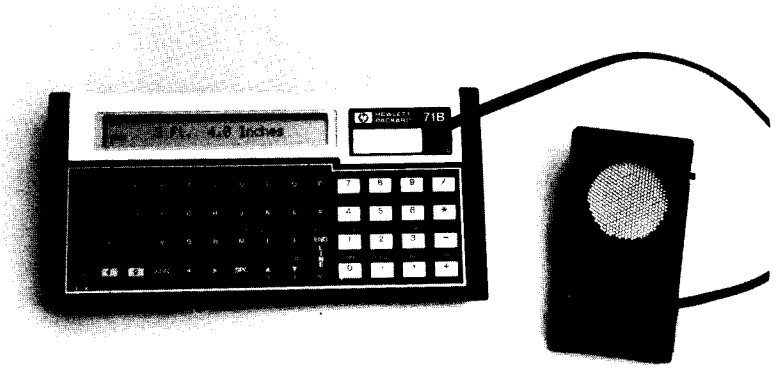
80	BS	1B	21
81	TAB	1B	09
82	LF	1B	0A
85	RETURN	FF	0D
87	CAPS LOCK	1B	CA
88	SPACE	FF	20
8B	ESC	FF	1B
8C	Left Arrow	1B	2B
8D	Right Arrow	1B	2A
8E	Up Arrow	1B	29
8F	Down Arrow	1B	28
90	0	FF	30
91	1	FF	31
92	2	FF	32
93	3	FF	33
94	4	FF	34
95	5	FF	35
96	6	FF	36
97	7	FF	37
98	8	FF	38
99	9	FF	39
9A	'	FF	27
9B	;	FF	3B
9C	,	FF	2C
9D	=	FF	3D
9E	.	FF	2E
9F	/	FF	2F
A0	`	1B	60
A1	A	FF	61
A2	B	FF	62
A3	C	FF	63
A4	D	FF	64
A5	E	FF	65
A6	F	FF	66
A7	G	FF	67
A8	H	FF	68
A9	I	FF	69
AA	J	FF	6A
AB	K	FF	6B
AC	L	FF	6C
AD	M	FF	6D
AE	N	FF	6E
AF	O	FF	6F
B0	P	FF	70
B1	Q	FF	71
B2	R	FF	72
B3	S	FF	73

B4	T	FF	74
B5	U	FF	75
B6	V	FF	76
B7	W	FF	77
B8	X	FF	78
B9	Y	FF	79
BA	Z	FF	7A
BB	[	FF	5B
BC	\	1B	5C
BD	]	FF	5D
BE	-	FF	2D
BF	DEL	1B	CB
C0	CNTRL BS	1B	08
C1	CNTRL TAB	1B	D1
C2	CNTRL LF	1B	D2
C5	CNTRL RETURN	1B	8D
C7	CNTRL CAPS LOCK	1B	D3
C8	CNTRL SPACE	FF	0D
CB	CNTRL ESC	1B	D4
CC	CNTRL Left Arrow	1B	AB
CD	CNTRL Right Arrow	1B	AA
CE	CNTRL Up Arrow	1B	A9
CF	CNTRL Down Arrow	1B	A8
D0	CNTRL 0	1B	30
D1	CNTRL 1	1B	31
D2	CNTRL 2	1B	32
D3	CNTRL 3	1B	33
D4	CNTRL 4	1B	34
D5	CNTRL 5	1B	35
D6	CNTRL 6	1B	36
D7	CNTRL 7	1B	37
D8	CNTRL 8	1B	38
D9	CNTRL 9	1B	39
DA	CNTRL '	1B	27
DB	CNTRL ;	1B	3B
DC	CNTRL ,	1B	2C
DD	CNTRL =	1B	3D
DE	CNTRL .	1B	2E
DF	CNTRL /	1B	2F
E0	CNTRL `	1B	FE
E1	CNTRL A	1B	41
E2	CNTRL B	1B	42
E3	CNTRL C	1B	43
E4	CNTRL D	1B	44
E5	CNTRL E	1B	45
E6	CNTRL F	1B	46
E7	CNTRL G	1B	47

E8	CNTRL H	1B	48
E9	CNTRL I	1B	49
EA	CNTRL J	1B	4A
EB	CNTRL K	1B	4B
EC	CNTRL L	1B	4C
ED	CNTRL M	1B	4D
EE	CNTRL N	1B	4E
EF	CNTRL O	1B	4F
F0	CNTRL P	1B	50
F1	CNTRL Q	1B	51
F2	CNTRL R	1B	52
F3	CNTRL S	1B	53
F4	CNTRL T	1B	54
F5	CNTRL U	1B	55
F6	CNTRL V	1B	56
F7	CNTRL W	1B	57
F8	CNTRL X	1B	58
F9	CNTRL Y	1B	59
FA	CNTRL Z	1B	5A
FB	CNTRL [	1B	5B
FC	CNTRL \	1B	FC
FD	CNTRL]	1B	5D
FE	CNTRL -	1B	2D
FF	CNTRL DEL	1B	D5

Closing Statement

Hooking up a simple keyboard isn't nearly as overwhelming a task as this chapter may have seemed. It is certainly a worthwhile endeavor, as it allows easy programming of the 71 while in the lab without giving up its size advantage in the field.


71B

Chapter Nine

AN ELECTRONIC TAPE MEASURE

If you can't measure it,
you can't control it.
--MacNarama's Management Philosophy

The next two chapters will address a topic that is usually reserved for larger, faster, and more power-hungry computers: Real-time I/O (Input/Output). This term means that the computer can detect and react to signals that are only milliseconds apart, something the HP-IL interface wasn't really designed to do.

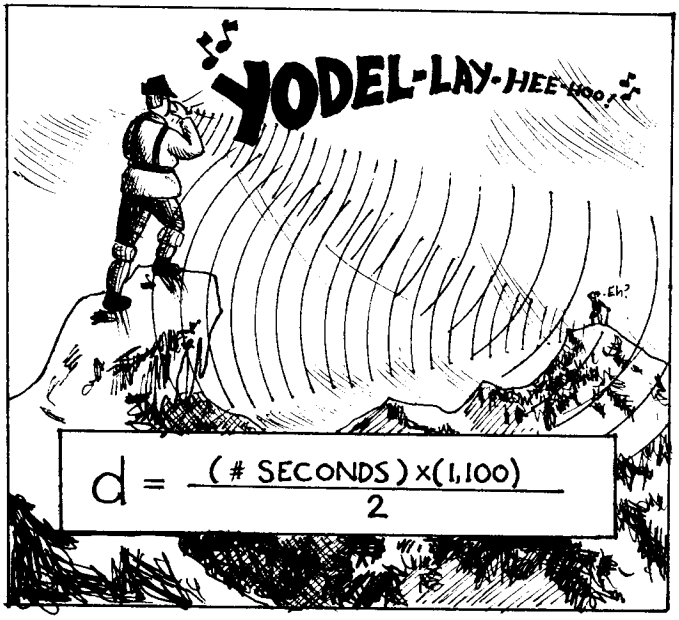
As an example of some of the possibilities, this chapter explains how to harness Polaroid's ultrasonic transducer (that little gold disk seen atop the SX-70 autofocus cameras) and allow the 71 to measure distances of objects by bouncing sound off of them. Using the same I/O technique, Chapter 10 describes a slide projector dissolve unit, which is a system that will continuously control the brightness of 2 high-wattage lamps independently using pulse-width modulation techniques.

Because these applications far exceed the standard I/O capabilities of even the 71, two things must be employed to attain this boost in system performance: 1) heavy use of assembly language for its inherent speed, and 2) connection of the CPU *directly* to the outside world. We'll cover that second topic a little later on.

The Polaroid Transducer

Back in 1972, Polaroid Co. developed a method of autofocusing

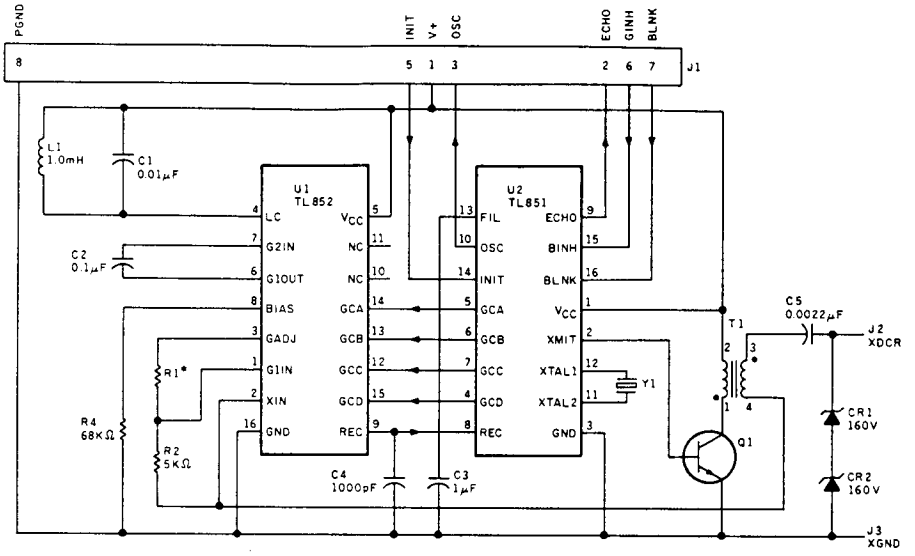
for their new SX-70 cameras which determined the distance between the camera and the subject by using sound waves. Using a specially developed transducer which could act both as a transmitter and receiver, 16 ultrasonic "chirps" would be sent out and the time it took for the sound waves to return would be measured. The system worked exactly the same way a mountaineer might measure the distance of an adjacent mountain: yell something and measure the time it takes for the echo to come back.



If an object is only five feet away, the echo returns in 9 milliseconds. A 71 using HP-IL, or even a 41 using the Time Module pads described in Chapter 3, isn't nearly fast enough to detect this echo. Assembly language routines using the CPU lines, however, have absolutely no problem.

Polaroid has made their technological development available to other product designers, and even put together a "prototyping kit", so people like me could buy just one and play around with it. That was about 10 years ago; since then TI has produced custom ICs that make interfacing to the special transducer a much easier

task, and today everything needed for playing around costs roughly 1/3 of what it used to. (Refer to Appendix B for cost and availability.)



A schematic of the circuit is shown in Fig. 9-1 above. What makes this module so wonderful is that it will automatically increase its sensitivity to the returning echo the longer it waits; extending its measurement range. Even better is it only takes four wires to interface the unit to a computer. (And two of them are power!)

Operation is simple: The controlling computer raises the INIT line to "1" (=5V). This sets the module about its business of sending the pulses and detecting their return. As soon as an echo is detected, the module pulses the ECHO line momentarily, which the controlling computer will detect. The time lapse between the INIT and ECHO pulses determine the time for the sound to travel, and therefore the distance. Piece of cake!

Assembly Language

The results from a simple speed test should explain why assembly language is essential for this application. Using HP-IL and one of the 8-bit ports, I tried to program the 71 to generate a high-frequency symmetrical square wave by turning one bit on and off as rapidly as possible. The results, measured on an oscilloscope, were disappointing: A BASIC program could only produce a frequency of 22 Hz; the equivalent FORTH program could only reach 14 Hz. (Although FORTH does indeed run 10 times faster than BASIC, FORTH's ENTER and OUTPUT routines call similar routines in the BASIC environment, thus making FORTH slower for I/O and nullifying one of its traditional advantages.) The same task in assembly language bypasses HP-IL altogether and gives an output of 18 KHZ, roughly 800 times faster than BASIC!!! Where did this square wave appear? It just so happens that the 71 possesses some internal CPU pins that are unused and readily accessible! (But more on this later.)

The 71 was selected for these demanding applications for several reasons: it has unused CPU lines available for input and output, its clock speed is faster than that of the 41, and the machine supports assembly language without a lot of excess bulk (such as a Machine Language Development Lab). This last attribute can be a mixed blessing; as at this time there is very little literature (with the exception of Richard Harvey's excellent book, "The Basic HP-71") to introduce beginners to assembly language on the 71's custom CPU. Learning from HP's documentation requires some previous knowledge, a small investment to purchase their operating system source code listings, and considerable study time.

(NOTE: Don't let this scare you away from exploring the immense possibilities of assembly language and direct I/O lines! After all, most of the work has already been done for you; and it is easier to modify someone's example than to re-invent the wheel.)

Figure 9-2 shows the layout of the 71's CPU registers. As can be seen, it is a 64-bit machine with nine general-purpose registers and an 8-level subroutine return stack. Only two of these, the A and C registers, are powerful enough to be called accumulators

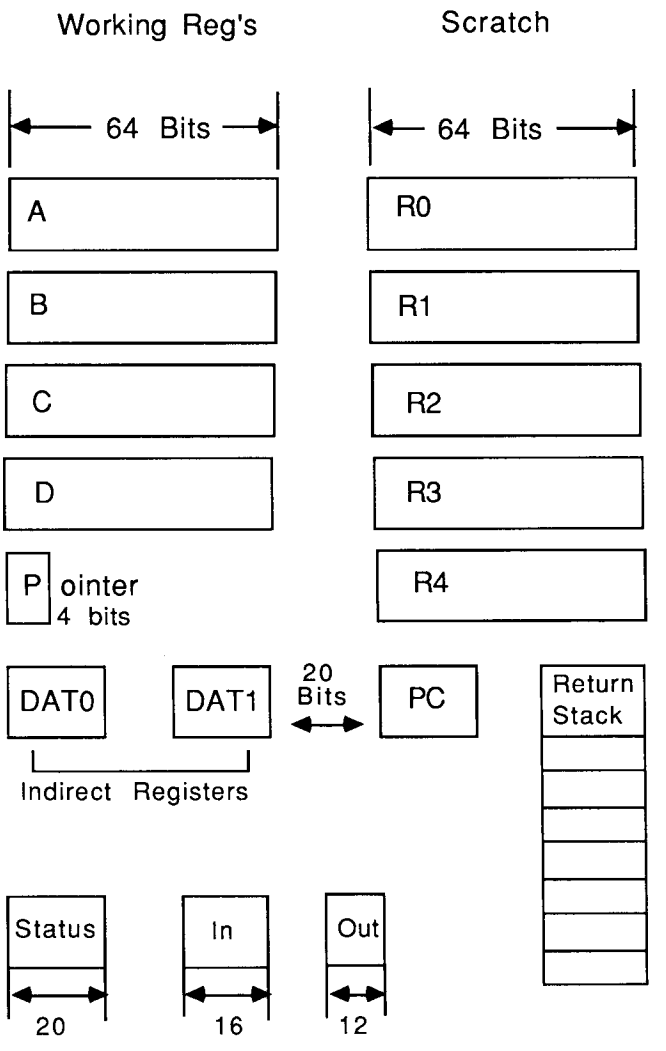


Figure 9-2
71's CPU
Registers

due to the fact that the instruction set performs almost all operations on these two registers. The others are used for temporary storage, loop control, data manipulation, etc. The FORTH/Assembler ROM's documentation covers the entire instruction set and explains their usage much better than I could do here, so I refer you to that as a companion to reading this chapter. They do not adequately explain the IN and OUT registers' function, for which I shall now compensate by using them in an example.

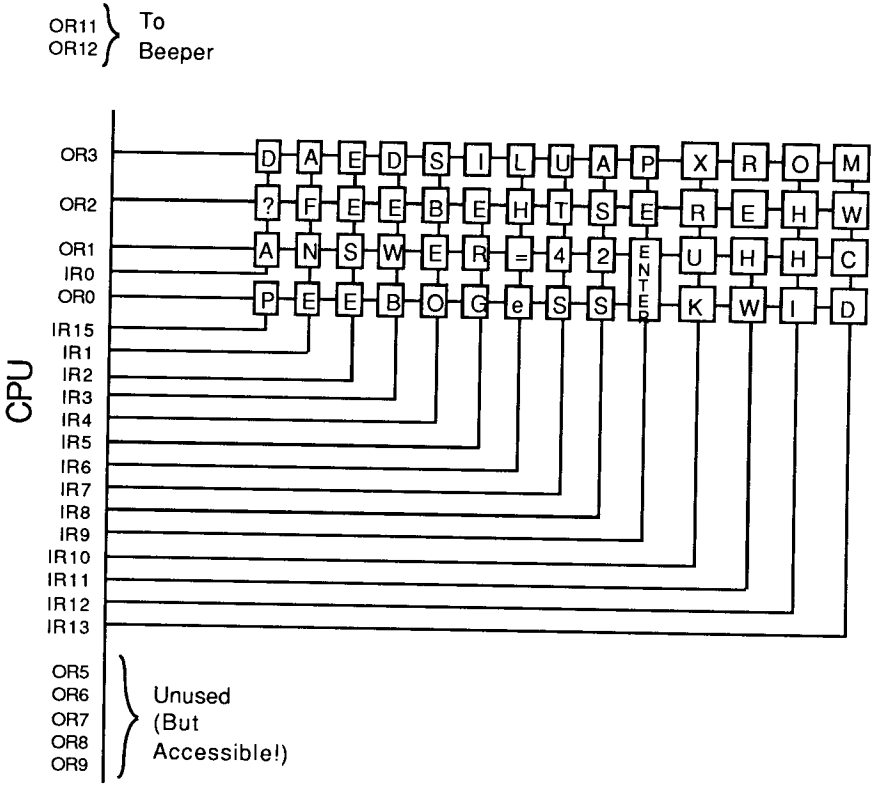


Figure 9-3
Keyboard Map
and Word Search

An Example: A Morse Code Keyer Program

If this will be your first assembly language program, it'll be fun and it'll give you a feel for it. If it isn't, do it anyway because it will show how you can directly access some CPU pins otherwise unreachable using BASIC or FORTH.

The following program turns the 71's keyboard into a Morse Code keyer, meaning it beeps as long as you hold any key down and stops when you release it. This deceivingly simple task, which cannot be accomplished using the 71's high-level languages, requires some knowledge as to how the keyboard scanning is performed.

There are twelve dedicated output lines and 16 dedicated input lines emanating from the CPU for the purpose of keyboard scanning (among other things). These input and output "words" are hooked up to the keyboard as shown in Fig. 9-3. When the 71 scans the keyboard to determine if a key has been pressed, the following sequence of events occur:

1) Line 0 of the output register (called OR0) is set to "1" (which =5 Volts) via the CPU's OUT=C command. All other output lines ="0".

2) If a key has been pressed (the space key for example), that key connects its row to its column, therefore the "1" coming from OR0 will appear at IR7 (input register 7).

3) The CPU looks at input lines 0-13 using the CPU`s C=IN command. If all lines are set to "0", then no key has been pressed in that row. If that word is not ="0", (as in our example), then the scanning algorithm concludes that the 7th key (IR7="1") in the zeroth row (OR0="1") has been pressed. It then jumps to a lookup table to figure out what that key is supposed to do.

4) If no keystrokes were detected, OR0 goes to "0" and then OR1 is set to "1", and step #3 is repeated again to check for keys depressed in the second row.

5) The whole process repeats until all 4 rows have been scanned.

The output register (see Fig. 9-4) also contains two additional bits, OR11 and OR12, which connect directly to the piezo-electric

buzzer. If the CPU alternately outputs a "1" and then a "0" to either of these lines, a loud or a soft beep will be heard. (OR12 has a lesser resistor connected to it, resulting in a louder tone).

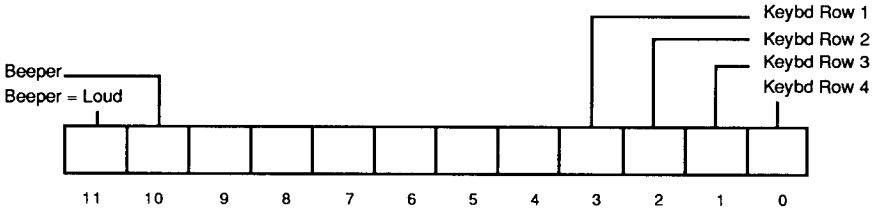


Figure 9-4
The OUT Register

As you can see, writing a Morse code keying program means you must have access to the dedicated input and output registers of the CPU; something not offered by the powerful BASIC and FORTH environments. This necessitates programming in assembly language, which is OK since we also could use the inherent increase in speed.

The algorithm used for the Morse code keyer is a little different from the way the keyboard is scanned. It goes something like this: the program constantly searches for a key being pressed (input register is not =0) and if it is it sets one of the beeper lines high, waits a finite period, sets the beeper line low, waits the same finite period, and goes back to see if the key is still being pressed. The duration of the finite period determines the frequency of the square wave generated.

The difficult part of this program was getting it to exit when the ATTN key was pressed. The method is very simple, but finding it took awhile. The ATTN key, when pressed, generates a hardware interrupt which will IMMEDIATELY jump to an interrupt routine at absolute address Hex 0000F. This routine does different things depending on certain conditions, but in this case it just sets the 12th bit in the status register and returns to the program that

was originally running. My program simply checks to see if this bit is set every so often, and if it is jumps to the ENDBIN exit routine.

The program listing for the Morse code BINary file appears below:

```
1:    BIN 'MORSE'
```

Line 1 identifies this as a BINary program and states the name by which it shall be called: MORSE.

```
2:    CHAIN -1
```

Line 2, CHAIN -1, lists the number of subroutines called by this program (which is zero, but the assembler wants to see -1).

```
3:ENDBIN EQU #0764B
```

Line 3 is a label which equates the word ENDBIN with the absolute operating system address of Hex 0764B. This is the address of a routine through which all BIN programs should exit.

```
4:    ST=0 #C
```

Line 4 sets the 12th bit in the status register to zero. When the ATTN key is eventually hit, the operating system will set it to 1.

```
5:    LCHEX 0000F
6:    OUT=C
```

Lines 5&6 perform two functions. LCHEX 0000F loads the nibble F into the C register and clears all remaining nibbles in the C register's A (address) field. OUT=C takes the F and puts it into the output register, driving all the keyboard scanning outputs high. The program will then respond to ANY key being hit.

```
7:    INTOFF
```

Line 7, INTOFF, disables the interrupt routine usually jumped to when any key (except ATTN) is pressed.

```
8:LOOP3 C=IN
9:    ?C#0 A
10:   GOYES START
```

Lines 8, 9, & 10 read the input register and if it's not zero, then a key has been pressed and will jump to the label START.

```
11:   ?ST=1 #C
12:   GOYES EXIT
```

Lines 11-12: If not, it then checks the 12th bit of the status register and if it's =1, then ATTN has been pressed and it should jump to EXIT.

```
13:   GOTO LOOP3
```

Line 13: If not, go back to LOOP3 and check for everything again.

```
14:START LCHEX F00
15:   OUT=C
```

Lines 14 & 15: LCHEX F00 and OUT=C set bits 11 and 12 of the OUTput register (beeper = LOUD) to 1. It also turns bits 8 and 9 on so we can watch this On-Off action on an Oscilloscope.

```
16:   LCHEX 015
17:LOOP1 C=C-1 X
18:   ?C#0 X
19:   GOYES LOOP1
```

Lines 16-19 form a delay loop =Hex 15. Modifying this constant and the one in Line 22 determines the beep frequency.

```
20:   LCHEX 01F
21:   OUT=C
```

Lines 20 & 21 turn the piezoelectric beeper off and turn the keyboard scanning lines on again.

```
22:   LCHEX 015
23:LOOP2 C=C-1 X
24:   ?C#0 X
25:   GOYES LOOP2
```

Lines 22-25 form another delay loop to produce a perfect square wave.

26: GOTO LOOP3

Go back to LOOP3 and start scanning the keys again.

27:EXIT INTON

28: GOVLNG ENDBIN

Lines 27 & 28 form the EXIT routine. It re-enables keyboard interrupts and jumps to the program that returns the user to the BASIC operating system.

29: END

End.

How to Enter Assembly Language Files

Like most assemblers, the FORTH/Assembler ROM expects to see the above program (without the comments on the right) in a text file, which can be created using the EDTEXT program contained in the same ROM. Just type in

```
EDTEXT (filename) <ENDLINE>
```

and when the 71 comes back with the 'Eof, Cmd:' prompt, just type

```
T <ENDLINE>
```

and start entering the code, one line at a time. WATCH FOR THE LEADING THREE SPACES ON EACH LINE!!! The assembler treats anything within the first three spaces as a label rather than an instruction. Also, you don't need to type in the line numbers; the text editor will put them in for you. When you are finished, hit the ATTN key, and at the Cmd: prompt, type 'E' (for Exit) and this returns control to the 71 you know and love.

When you go into the FORTH environment and type

```
" (textfile name)" ASSEMBLE <ENDLINE>
```

the assembler converts your text file to machine code. If the assembler detects an error in your code, it will quietly display an error message with no beeps. Watch it carefully!

After going into FORTH and assembling the program, all you have to do is type BYE (which returns you to BASIC), type "RUN MORSE" <ENDLINE> and you're in business.

Other notes while we're on the subject of producing assembly language files:

Be sure the name of your text file is not the same as the assembly language file it will be producing! (i.e., make sure the name following 'BIN' is unique!) This is necessary because, unlike many other computers, the 71 doesn't support different file types with identical names. If the file you want to access is first in the filechain, great! All others after that with identical filenames cannot be accessed.

It is highly recommended that a backup of your latest program version be kept in independent RAM, so the inevitable MEMORY LOSTs won't set you back too much timewise. I've defined a few words in the FORTH environment to make this task automatic:

```
: REPLACE " PURGE MORSET:PORT(2) @ COPY MORSET TO
     :PORT(2)" BASICX ;
: ASS REPLACE " MORSET" ASSEMBLE " BEEP" BASICX BYE ;
```

All I have to do now is go into FORTH mode and type ASS (that's short for ASSemble, gang!) and it automatically generates a backup copy of MORSET (the "T" suffix distinguishes it as a Text file) in PORT(2) of independent RAM, assembles it, BEEPs when completed and returns me to BASIC.

I know what you're thinking: "Gee, this program sure was fun to produce and play with, but what does this have to do with interfacing to the outside world?". Well, it just so happens that the IN and OUT registers, with which we have just now become

intimately familiar, have extra bits that directly control CPU pins that aren't used for anything, and can be turned on and off using the OUT=C command!

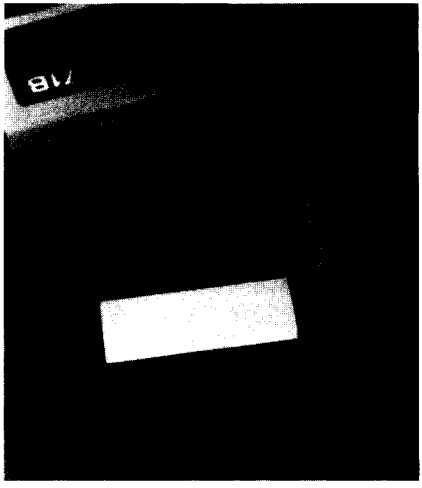
Referring back to Fig. 9-3 (the OUT register map), we see bits 5 through 9 unused. Furthermore, just as all the other bits (which control the keyboard rows and the beeper) are accessible at the CPU pins, so too are these unused bits, just waiting to be harnessed! There are, unfortunately, no extra pins leading to the INPUT register for similar applications, but we do have one that can do double duty: IR14, the input usually used for peripherals (ROMs, RAMs, card reader, etc.) to generate an interrupt and grab the CPU's attention. Before we can use any of these, though, we must first find some way of bringing these signals to the 71's surface.

Hardware Modification

I decided to make this modification as versatile as possible by opening up the 71 and bringing a total of six signals to a 6-conductor modular telephone connector. The six signals are:

OR7, OR8, OR9
IR14
5V, GND

The fun part, of course, is opening up the 71 and soldering wires to strategic points on its circuit board. Currently, there exist two versions of the 71; the later of the two being much easier to disassemble than the earlier. Visual inspection of the 71's exterior will reveal which version you have: the earlier one has three tiny brass screw heads showing on the bottom plate; the later one doesn't. The earlier one has all the springs in the battery compartment lined up on one side; the later one doesn't. The earlier one contains a ribbon cable inside that joins the top and bottom halves and therefore requires a little more caution than the newer version. Procedures for both are provided next.



The Easy Part

The first part of the modification is relatively risk-free. First, get a small, 6-conductor modular phone jack. (Not all are small enough to fit neatly, so scavenge carefully.) To this, attach six strands of insulated wire (the kind used for wire wrapping works nicely), about 8-10 inches in length. Next, we go to work personalizing the 71.

Every 71 comes with a "dummy" block to fill in the space that the card reader normally occupies. If you remove this block and carefully pry off the decorative metal strip, you can then take a miniature grinding tool (or a drill with a tiny drill bit) and carefully cut out a square hole. It is best to do this slowly, and periodically check the hole's size against the jack to be installed. It is also necessary to remove the structural rib from inside the block, so the jack can be pushed in from the bottom and glued there.

How to Disassemble the 71

It is clear by the machine's construction that HP didn't want anyone to open the 71. (We'll show them!) Getting into it isn't such a difficult thing once you know how, but there are some basic precautions: 1) make sure the warranty has expired (some of you more daring folk may choose to ignore this; I know I did). 2) This is one case where I definitely recommend soldering and mechanical experience before attempting this feat! (If you lack such experience, you may wish to consult Appendix B for firms that will perform this modification for you.)

The next step is to take the 71 apart and solder the 6 wires from the modular jack to strategic places on the circuit board. Essential

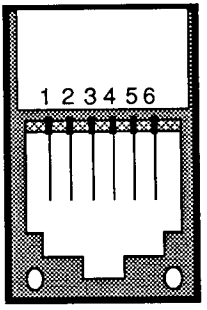
tools needed are 1) a small jeweler's flathead screwdriver, and 2) a #6 TORX wrench. WARNING: Do not attempt this disassembly if the proper tools (specifically the TORX wrench, which is probably difficult to obtain (even from HP, which sells it as part number 8710-1424)) are not available. These special screws damage easily if not handled by the correct instrument. (Appendix B also contains sources of difficult-to-find items.)

Old 71

If you own the "old" style 71, disassemble it via the following procedure:

1) Flip the 71 over so its bottom is up and the four ports are facing you. Remove the rear card reader cover, the battery cover, and the batteries.

2) With the flathead screwdriver, carefully peel off the three remaining rubber feet. Fig. 9-6 shows the five visible TORX screws, one in each corner and one along the top center.



Pin #	Color	Signal
1	White	OR8
2	Black	OR9
3	Red	+5v
4	Green	GND
5	Yellow	IR14
6	Blue	OR7

Figure 9-5
Modular Jack
Pinout.

WARNING!!!

Depending on the modular cable's construction, the receiving jack's pinout may or may not be the reverse (i.e, mirror image) of that shown above. Examine the cable carefully!

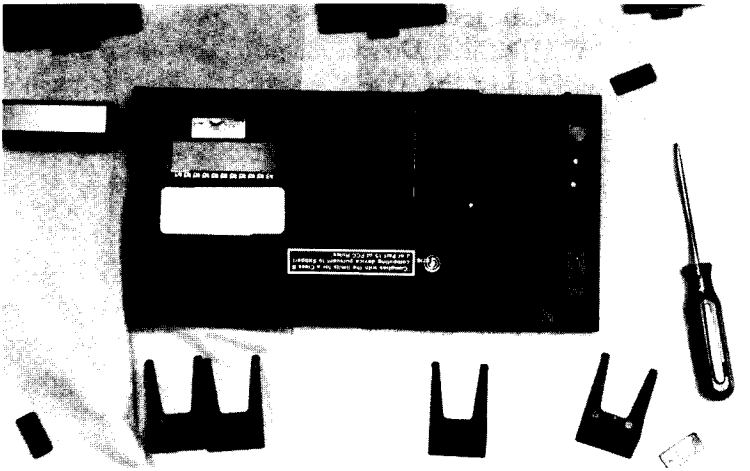


Figure 9-6. The 71's back just before opening. Knowing which brass screws to remove is the key.

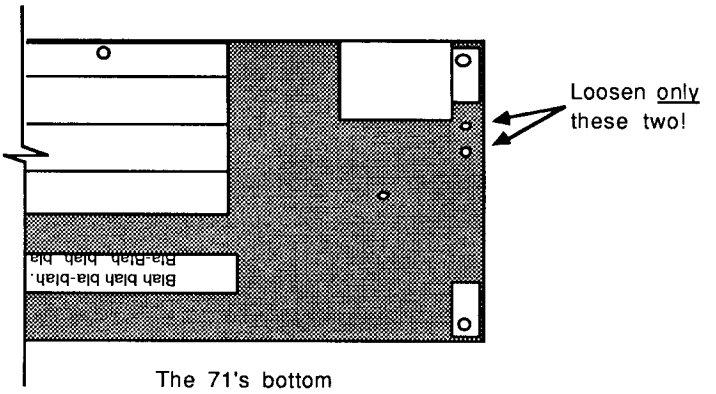


Figure 9-7
Disassembling
the old 71.

3) Carefully unscrew all five TORX screws. DO NOT ATTEMPT TO DO THIS IF YOU DON'T HAVE THE PROPER TOOL!

4) Notice the three remaining tiny brass screws on the surface facing you. Using the flathead jeweler's screwdriver, loosen ONLY the two which are clumped together. See Fig. 9-7.

5) Now comes the crucial part of separating the top and bottom half. AS YOU DO THIS STEP, BE VERY CAREFUL NOT TO BEND, TWIST, OR OTHERWISE CRACK THE RIBBON CABLE CONNECTING THE TWO HALVES TOGETHER. As you attempt to lift the bottom half away from the remainder of the computer, you will notice that the "northern" part, which normally accommodates the batteries, IL Module, and card reader, lifts off easily; while the edge containing the four plug-in modules seems hesitant to move. This is normal. Just hold the 71 vertically so the "ON" key is closest to the ceiling, and gently pull the two halves apart from the top. The only resistance you are likely to encounter will be a plastic "catch" placed in the center of the 71's four ports; "twisting" the almost-separated halves somewhat while pulling will usually free them. Be careful when separating the two halves and laying them down flat, for the ribbon cable connecting the two halves is very fragile. When done, the new unit should be laid down like a book, with the keyboard half on the right and the bottom half on the left.

Open the 71 this way, with the ON key pointing up. The old version has a fragile ribbon cable at the bottom, which should be treated as a hinge.

6) On the right half is a large sheet of copper, used to shield the unit from electromagnetic interference. It can easily be peeled off from the ICs it is attached to. Doing so reveals three adjacent ROMS and a somewhat distanced CPU.

7) Sit and stare at the incredible job HP did squeezing such a powerful computer into a tiny space.

8) We are now going to solder the six wires from the modular jack to the 71's guts. Pins #3 and 4, which supply +5v and Gnd respectively, connect to the 470 microfarad capacitor right next to the 71's AC adapter input. (Fig. 9-8) Pin 3 gets connected to the side of the capacitor labeled as "+".

9) Now attach pins 1,2,5, and 6 of the modular jack to the right side (top half). Figure 9-9 shows a subset of all the plated-through holes surrounding the one isolated 1LF2 integrated circuit that we must access. The pinout is as follows:

OR7	CPU Pin 8
OR8	CPU Pin 9
OR9	CPU Pin 10
IR14	CPU Pin 51

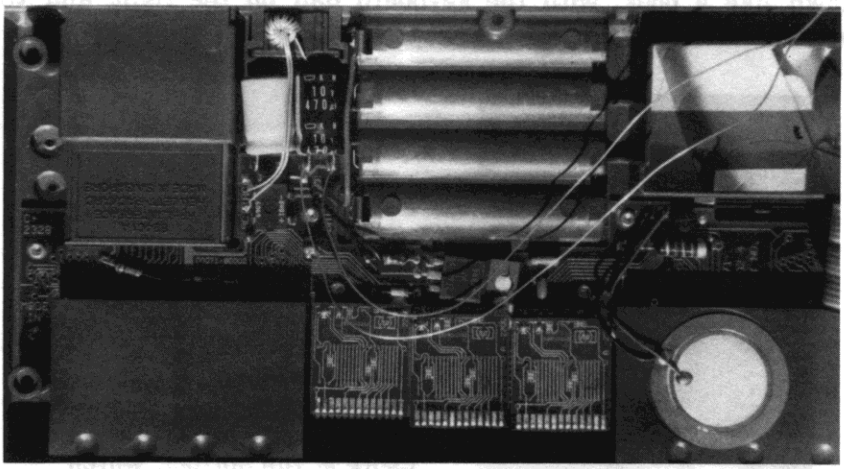
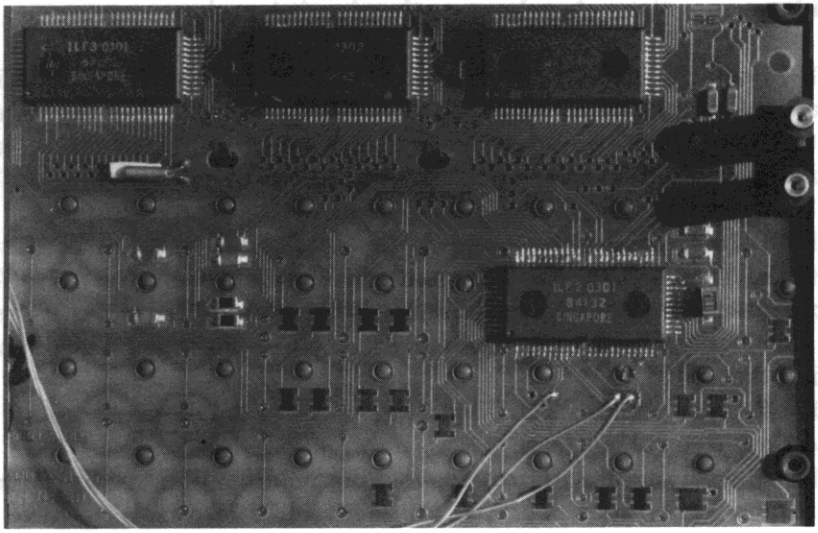


Figure 9-8. +5v and Ground are easily tapped on the 71's power supply capacitor leads.



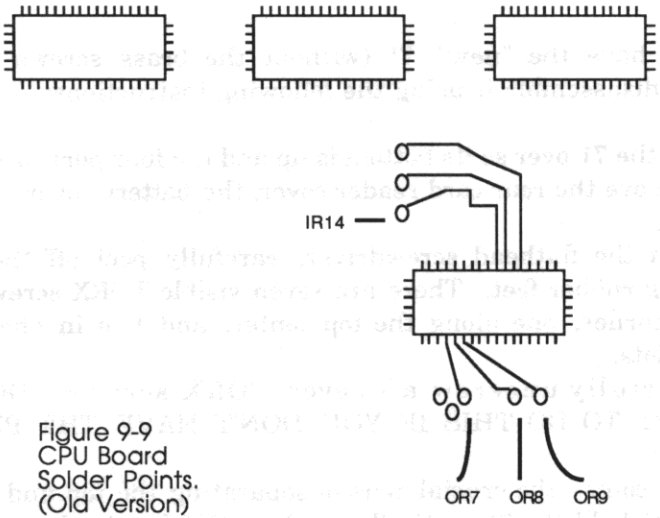


Figure 9-9
CPU Board
Solder Points.
(Old Version)

Be careful when soldering wires to these plated-through holes; excessive heat may damage the IC, and excessive solder may bridge connections elsewhere on the circuit board.

10) Push the block (now with wires attached) through the hole in the left (lower) half of the 71, and close the machine. Pull the wires out as far as you can while closing, but be careful that the wires don't get caught in places that prevent the case from closing fully or between the card reader's contact pins.

11) Before replacing the screws, replace the batteries and confirm that the 71 will turn on. (You should expect a MEMORY LOST, since there's no memory retention when changing batteries.) Also do a soft and a loud BEEP to insure the piezo- electric buzzer isn't being disturbed.

12) Replace the five TORX screws, the two small brass screws, and the three rubber feet.

13) Marvel at your impressive-looking modification, and prepare to respond to billions of stupid comments like "Wow! You have a modem? What speed is it?".

New 71

If you have the "new" 71 (without the brass screws on the bottom), disassemble it using the following instructions:

1) Flip the 71 over so its bottom is up and the four ports are facing you. Remove the rear card reader cover, the battery cover, and the batteries.

2) With the flathead screwdriver, carefully peel off the three remaining rubber feet. There are seven visible TORX screws, one in each corner, one along the top center, and two in one of the battery slots.

3) Carefully unscrew all seven TORX screws. DO NOT ATTEMPT TO DO THIS IF YOU DON'T HAVE THE PROPER TOOL!

4) Now comes the crucial part of separating the top and bottom half. Just hold the 71 vertically so the "ON" key is closest to the

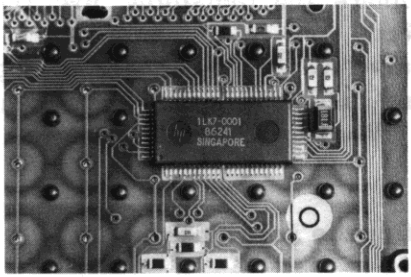
ceiling, and gently pull the two halves apart from the top. The two halves separate quite easily and there are no internal wires connecting them.

5) Sit and stare at the incredible job HP did squeezing such a powerful computer into a tiny space.

6) We are now going to solder the six wires from the modular jack to the 71's guts. Pins #3 and 4, which supply +5v and Ground respectively, connect to the 470 microfarad capacitor right next to the 71's AC adapter input. (Refer to Fig. 9-8 from page 222). Pin 3 gets connected to the side of the capacitor labeled as "+".

7) Now to hook up pins 1,2,5, and 6 of the modular jack to the right side (top half). Figure 9-10 shows a subset of all the plated-through holes surrounding the one isolated 1LF2 integrated circuit. The pinout is as follows:

OR7	Pin 8
OR8	Pin 9
OR9	Pin 10
IR14	Pin 51



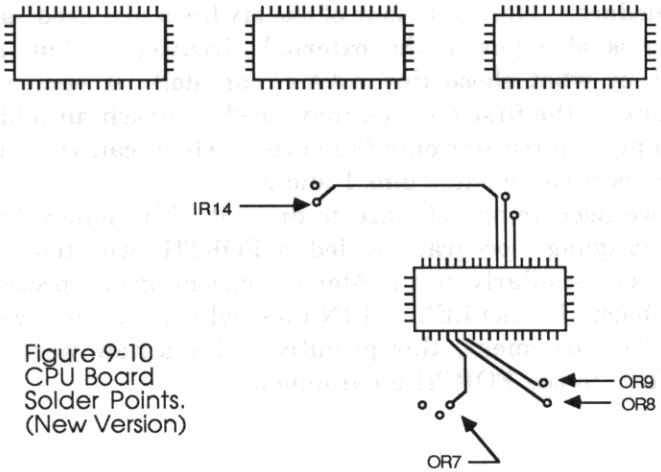


Figure 9-10
CPU Board
Solder Points.
(New Version)

Be careful when soldering wires to these plated-through holes; excessive heat may damage the IC, and excessive solder may bridge connections elsewhere on the circuit board.

8) Push the block (now with wires attached) through the hole in the left (lower) half of the 71, and close the machine. Pull the wires out as far as you can while closing, but be careful that the wires don't get caught in places that prevent the case from closing fully or between the card reader's contact pins.

9) Before complete re-assembly, we should check the 71's health. Put the two halves together and replace only the two adjacent TORX screws in the 4th battery chamber. Upon replacing the batteries, confirm that the 71 will turn on. (You should expect a MEMORY LOST, since there's no memory retention when changing batteries.) Also do a soft and a loud BEEP to insure the piezo- electric buzzer isn't being disturbed.

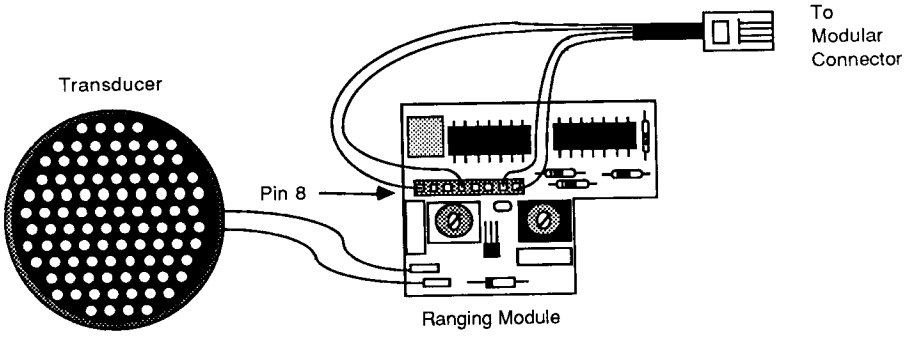
10) Replace the remaining five TORX screws and the three rubber feet.

11) Marvel at your impressive-looking modification, and prepare to respond to billions of stupid comments like "Wow! You have a modem? What speed is it?".

Connecting the Transducer

Fig. 9-11 shows how we can use the new 71 I/O lines to connect to the transducer. Here only four of the six lines are used, and the 71's batteries also power the external circuitry. (I'm always impressed by what those tiny AAA's can do!) If your module doesn't work on the first try, you may need to attach an additional 3.9K Ohm pull-up resistor onto the output, which can conveniently be soldered between module pins 1 and 2.

Next, we need some software to drive it. The following is an assembly language program called a FORTH primitive, and it functions very similarly to the Morse Code program presented a few pages back. Unlike LEX or BIN files, which add new words to the BASIC environment, this primitive adds a new word, called "MEASURE", to the FORTH environment:



Module Pin #	71 Connections
Pin 1	+5v
Pin 2	ECHO
Pin 5	INIT
Pin 8	Gnd

Figure 9-11
How the Ultrasonic Ranging Module is Attached.

Note: A 3.9KΩ resistor may be needed between pins 1 and 2 for proper operation.

```
1:      FORTH              Identifies this as a FORTH
                           word.
2:      WORD 'MEASURE'     Name of the function.
3:      INTOFF             Disable interrupts normally
                           generated when IR14 is brought
                           high.
4:      A=0 A              Set the Address field of the A
                           reg. =0.
5:      LCHEX 20F          Turn OR9 on. (Keyboard is
                           activated, too.  See text.)
6:      OUT=C
7:      C=0 A
8:WAIT  A=IN               Read the inputs.  Only IR14
                           will be recognized.
9:      C=C+1 A            C keeps track of how long we've
                           waited.
10:     GOC EXIT           Provide a timeout in case we
                           measure the sky.
11:     ?A=0 A             Has IR14 not come back?
12:     GOYES WAIT         Yes, go back to line 8 and do
                           it again.
```

```
13:EXIT D1=D1- 5          Otherwise, we received an echo.
14:     DAT1=C A          Take the value in C and push it
                          onto the FORTH stack.
15:     C=0 A             Bring the INIT line low again.
16:     OUT=C
17:     RTNCC             Return with Carry Clear, the
                          proper way to return to FORTH.
```

This program, being a FORTH word, must be run in the FORTH environment. After assembling, just type

```
MEASURE . <ENDLINE>
```

which tells it to run the program MEASURE and to display the first number on the the stack (the "." command). The display will show something like

```
163  OK { 0 }
```

The number on the left shows the number of times the C register got incremented while waiting for an echo to return, and the number between the brackets on the right shows how many numbers are on the stack. We must now calibrate these readings if the result on the left is to be meaningful.

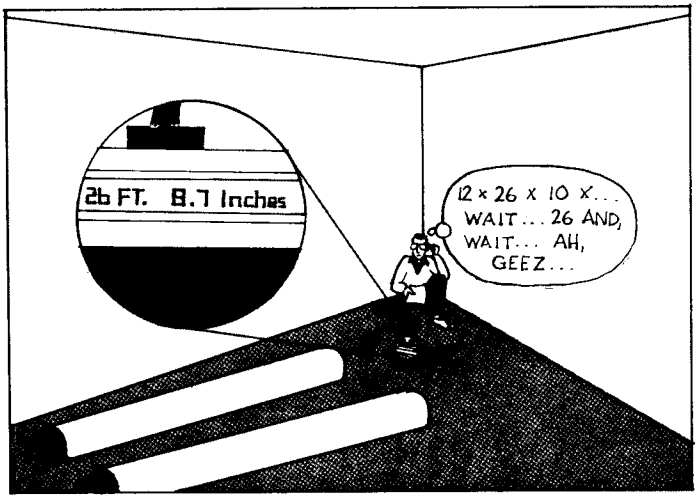
Calibration is a simple process. Place the transducer at several fixed, known distances from a flat wall and record the number returned by MEASURE. Plotting these on graph paper should yield the straightest of lines, and the slope of the line determines the number you multiply the result by to get the distance in inches. My own measurements yielded this equation:

inches = (result from MEASURE)x 0.370

I also discovered during this time that you should NOT plug an AC

adapter into the 71 during transducer operation. The noise from this badly filtered source is so great that the driving module gets confused and won't recognize echoes that travel further than 10 feet. The following program, written in BASIC, does the proper conversion:

```
5 ! Program TAPEM (Tape Measure) works with the
    Polaroid Transducer.
6 ! It takes the results from the FORTH environment
7 ! and displays them in feet and inches.
10 FORTHX " MEASURE"
20 N=FORTHI
30 N=N*.37
40 F=INT(N/12)
50 I=MOD(N,12)
60 IF F=0 THEN DISP USING 70;I ELSE DISP USING 80;F,I
70 IMAGE 2D.D," Inches"
80 IMAGE 4D," Ft. ",2D.D," Inches"
```



This program automatically goes into the FORTH environment and executes MEASURE (line 10), multiplies the result by the experimentally obtained slope, and formats the output to read in feet and inches. Standing in a corner, you could use the above program to measure the volume of a room by taking only 3 readings!

An extra measure of protection (no pun intended) is provided in line 5 of the FORTH primitive. In the event that you try to measure the distance of anything greater than 30 feet away (such as the ionosphere), no echo will be received and the program will just sit there forever counting and waiting. Because the keyboard's OR0-OR3 bits were also turned on in line 5, a returning echo OR ANY KEYSTROKE will terminate the loop and provide a reading. This makes for easy recovery from bad aiming; if the program hangs because it didn't receive an echo, just press any button on the keyboard and the program terminates instantly.

Personal Space Invasion Alarm for Valley Girls

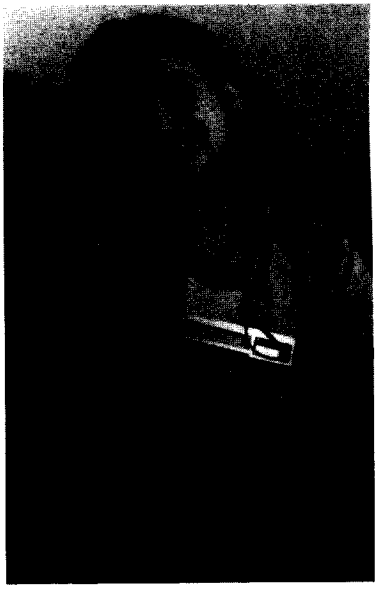
Another great application combines both high-fashion and security, and would probably sell a million if it were on the market. The Personal Space Invasion Alarm audibly warns passers-by that the user's space is being invaded, and is a must for Valley girls who travel abroad.

In practice, the transducer is worn around the neck as a piece of fashionable jewelry, while the relatively ugly driver components are hidden in the purse along with the 71. The 71 has been programmed to periodically fire the transducer, and will beep continuously as soon as anything comes within 1 1/2 feet of the user.

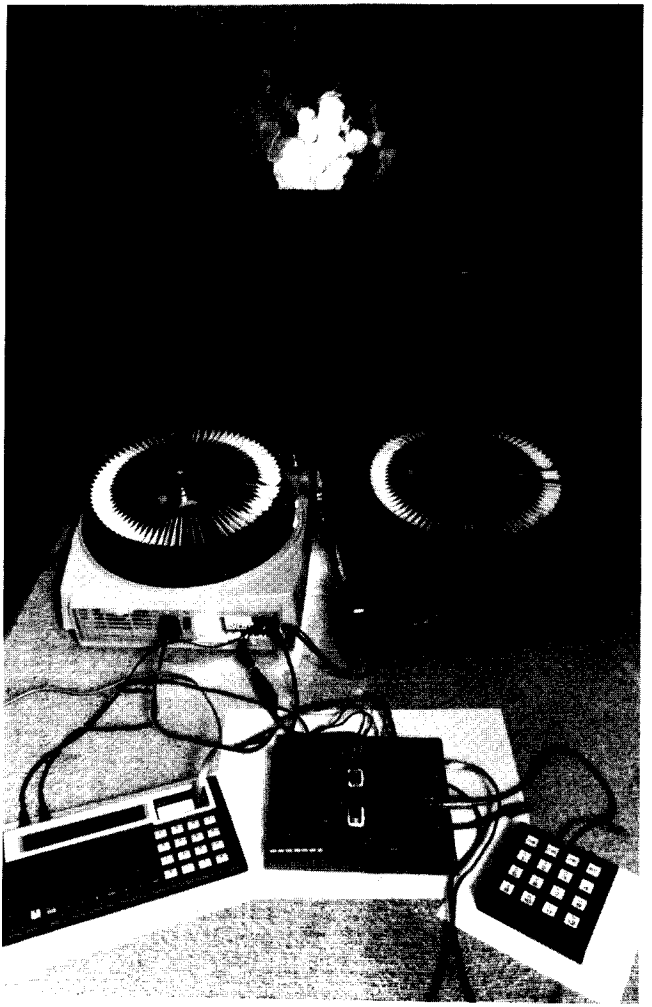
This application requires 2 additional lines to the TAPEM program above:

```
25 IF N < 50 THEN BEEP @ GOTO 10
65 GOTO 10
```

It is unfortunate that any modification must be made at all in order to harness an otherwise untapped potential of the 71, for its small size makes it the only practical option for portable measurement devices. I have found, however, that it is a most worthwhile change and does nothing but increase the performance of an already powerful machine.



The fashionable **Personal Space Invasion Alarm** consists of the transducer which is worn around the neck, and the ugly guts which are hidden in the purse.



Chapter Ten

A SLIDE PROJECTOR DISSOLVE UNIT

While we're at it, why don't we write a 71 program that decreases entropy?!

-Richard Nelson

Chapter 9 described a modification to the 71 that allowed direct access to some CPU pins. This chapter shows a very different use for this modification. Here the 71 becomes the center of a large audio/visual system, taking its cues from audio tape and controlling the intensities of 2 projector lamps.

A slide projector dissolve unit is designed to make slide shows a more artistic and pleasant experience. Rather than the harsh bright-black-bright that usually accompanies such shows, this system uses 2 projectors and slowly fades one image out while the next image slowly fades on, and the transition looks as if one image dissolves into the other. Of course, the dissolve intervals and duration are all user-controlled.

Designing this system required the merging of several disciplines, including assembly language, FORTH, Touch Tone signaling, pulse width modulation, and zero-crossing detectors. All but the last two items have been discussed in previous chapters, which means only one or two new techniques must be introduced in order to accomplish something vastly different.

How Light Dimmers Work

Contrary to popular belief, lights do not have to be dimmed by putting them in series with a potentiometer or rheostat so as to

reduce the voltage driving them. Another method best used by computers actually fools the eye and keeps it on only half the time. For example:

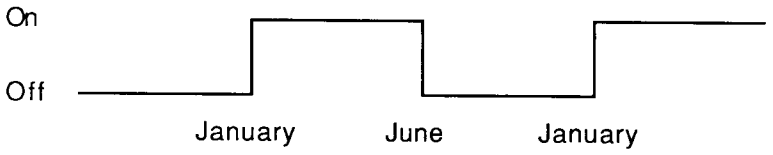


Figure 10-1
Low Resolution
Light Dimming.

The diagram in Fig. 10-1 will keep a light bulb going 50% of the time. Its obvious drawback is its 6 month period, which certainly will not fool the human eye into thinking it's on continuously.

Let`s do it again, only this time we'll use a faster time scale:

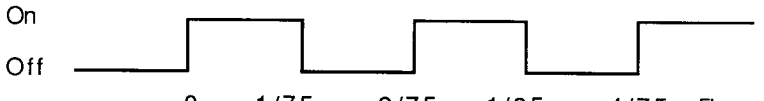


Figure 10-2
A Trivial
Improvement.

Same duty cycle (50%), but this time it's so fast that the light appears to be continuously dim when it's actually turning on and off very quickly. If you're dimming with this method, any brightness from 0 to 100% can be attained by varying the duty cycle (ratio of ontime/offtime).

Controlling an alternating current (AC) lamp is a bit more complicated. (No pun intended.) For example, let's take the previous timing diagram and apply it to an incandescent bulb, as shown in Figure 10-3.

When you try to dim an AC lamp by varying its pulse width, there's a good chance you'll develop what's known as an interference pattern, as shown in Fig. 10-3C. Interference patterns occur due to the computer not being synchronized with

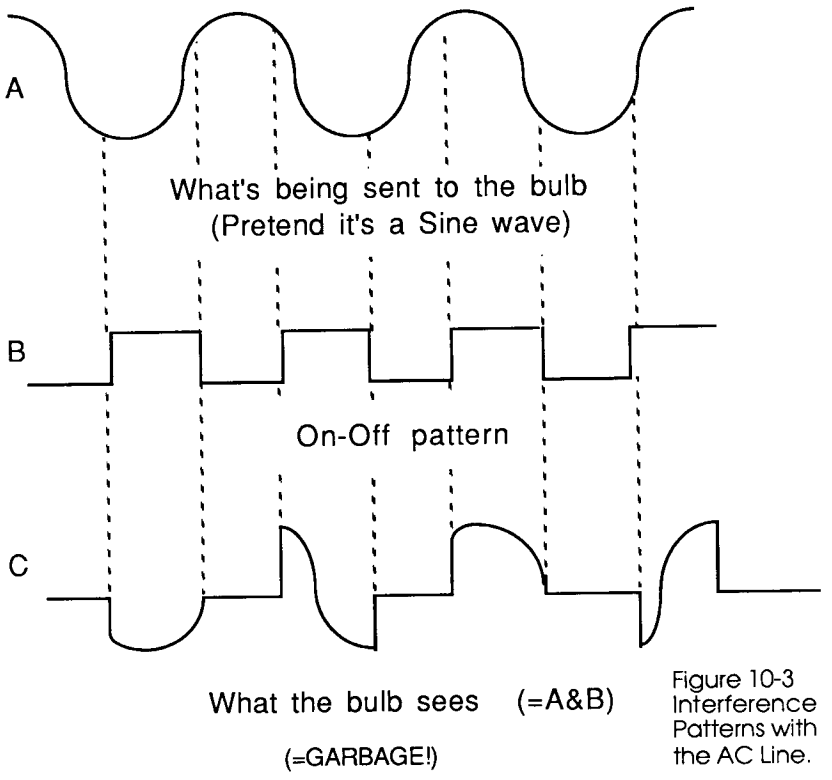


Figure 10-3 Interference Patterns with the AC Line.

the AC lines, and make the bulb's output look like there's an intermittent break in the AC cord, rather than the constant illumination we're so used to seeing.

The Zero Crossing Detector

The way around this is to synchronize the two waveforms with a zero-crossing detector (ZCD), whose function is illustrated in Fig. 10-4 (next page).

Every time the AC signal crosses zero volts, the ZCD outputs a short pulse to the computer generating the control on/off signals. And if the computer was smart, it would use the pulses as in Figure 10-5:

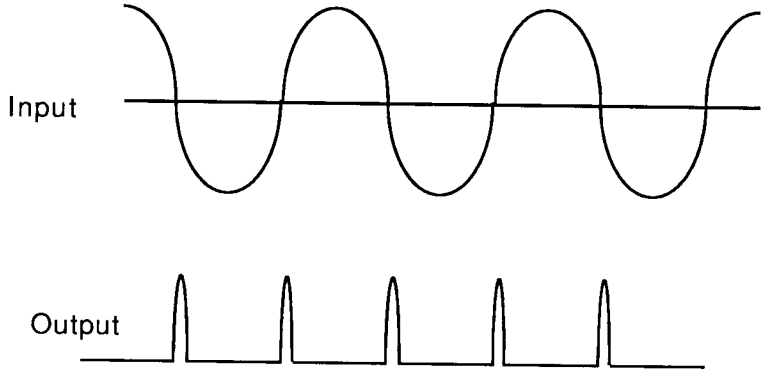


Figure 10-4
The Output of a Zero Crossing Detector.

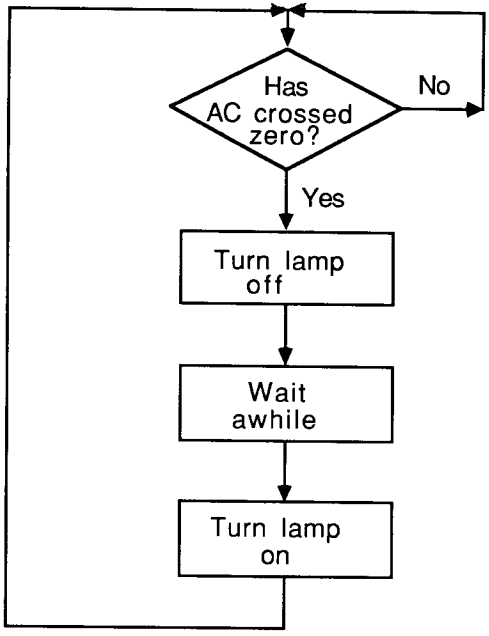


Figure 10-5
What We Must Do Within 1/120 of a Second.

and your output would look like Fig. 10-6:

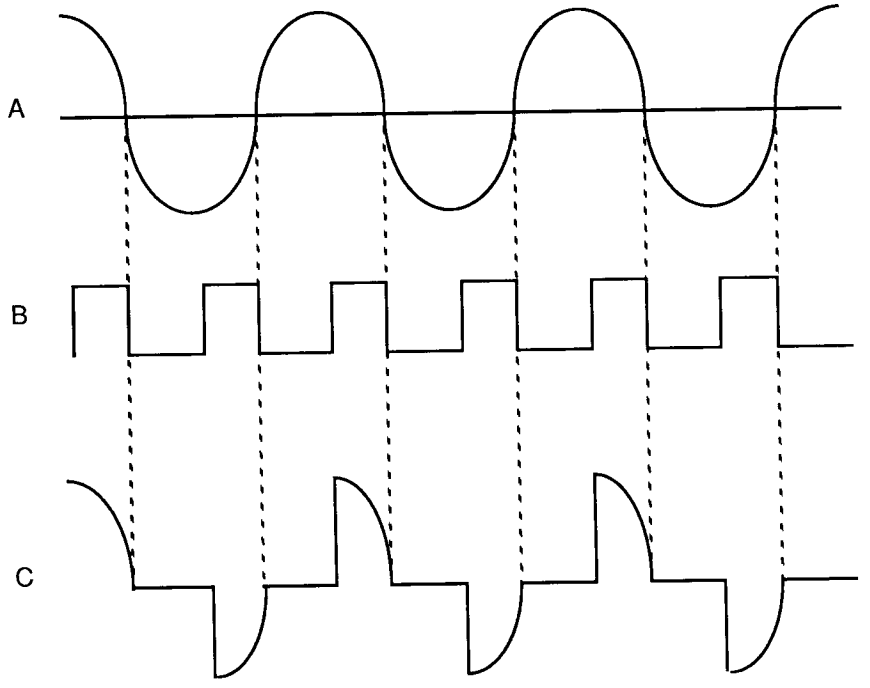


Figure 10-6
Clean Output.

The amount of time you wait, of course, determines the intensity of the lamp. If you don't wait at all the brightness is 100%; if you wait for more than 1/120 of a second, it behaves more like a lantern than a lamp. Our purpose, of course, is to wait at a continuously variable rate between 0 and 100%.

The circuitry shown on the next page not only provides the ZCD pulses to the computer, but also acts as a power supply for both the circuitry and the 71. (See Fig. 10-7.)

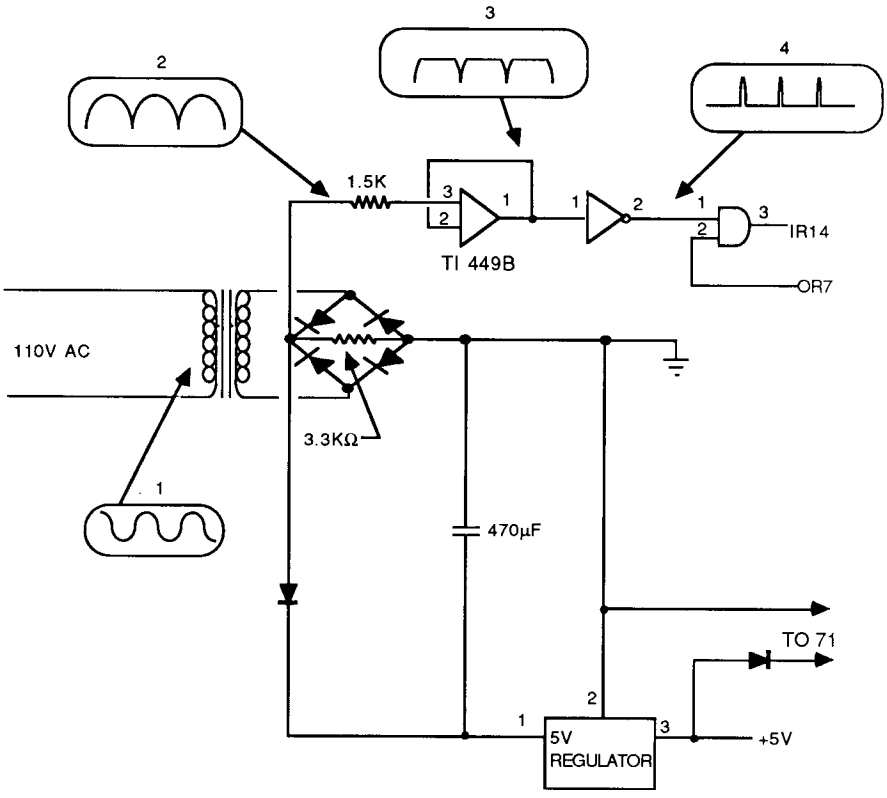


Figure 10-7
A Walkthrough
of the Zero
Crossing Detector.

First, start with HP's standard, hot-running 8v AC transformer (1). Feed this into the bridge rectifier which turns the AC into pulsating DC (2). (Notice the load resistor across the bridge rectifier!) The low power Op Amp which follows is configured for a gain of 1. Here we make use of the fact that an Op Amp's output voltage can only be a little less than its power supply, so that with an input waveform of 8V (2), only the 0-5V portion gets reproduced correctly and the rest gets "clipped" (3). This now makes the signal correct for the inverter, which converts (3) into (4), and we have our end product: a signal to tell the 71 when the AC power crosses zero volts.

The remainder of the circuitry near the bottom, which includes the diodes, the capacitor, and the 5V regulator, form the power supply which is enough to power all the circuitry as well as the 71.

How to harness it

Here we make use of the two extra wires installed in the last chapter but never used. We need to access output lines OR7, OR8, OR9, and Input line IR14. We also use the +5v and Gnd wires to feed power to the 71, just to insure that your AAA batteries don't die during a presentation.

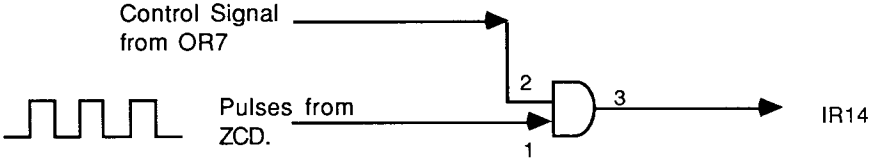


Figure 10-8
Protecting IR14
From Constant
ZCD Pulses.

Fig. 10-8 shows how one of these wires are hooked up. Since IR14 is used for all system interrupts (including the ATTN key), we must be very careful to send it signals ONLY DURING THE ASSEMBLY LANGUAGE ROUTINES, otherwise the 71 will be interrupted 120 times a second and will be bogged down with having to handle them.

OR7 is used with the AND gate to solve this problem and block out these ZCD inputs when they're not needed. When the assembly language portion of the program is ready to accept the ZCD as input, it simply sets OR7 high which "turns the AND gate on" and allows its other input to appear at its output, letting the pulses through the gate to reach IR14. Setting OR7 to "0" when it is through shuts off the pulse train. This way, the 71 can be used

when it isn't dimming lights.

OR's 8 and 9, which are used to directly control each projector lamp, are hooked up as shown in Fig 10-9 below. Because we don't wish to overload the fragile CPU's final driver transistors, an AND gate whose inputs are tied together is used as a buffer to drive the 3010 Opto Isolator/Triac Driver IC. This in turn controls the high voltage of a triac, which in turn connects in series with a light bulb and controls its states.

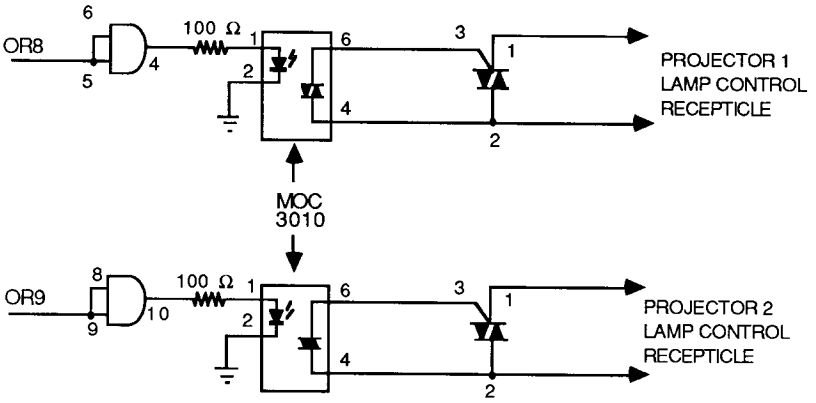


Figure 10-9
How Output Register Bits 8 and 9 Directly Control Lamps.

Sample Program

Just to show how all this is supposed to work together, let's look at a "very simple" (I suppose everything's relative) assembly language program that takes only one light bulb and fades it from off to on in about 10 seconds.

This program is a FORTH primitive; a new word added to the FORTH dictionary written in assembly language rather than other FORTH words. It works like the flowchart of Fig. 10-5, but alters the "wait awhile" variable so the intensity gradually changes. The total time to fade from off to on is controlled by the stack input which can range from 1-15: 1 will fade slowly; 15, the

highest you can go with Touch Tone input, will fade so quickly as to look instantaneous. (Refer to Fig. 10-10 for OUT register map and Fig. 10-11 for register usage.)

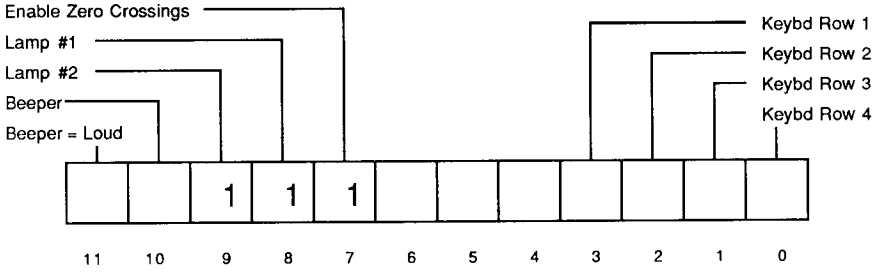


Figure 10-10
The OUT Register.

```
FORTH      *** FADES AN AC LIGHT FROM OFF TO ON ***
WORD 'FADEON'          ( DISSOLVE SPEED --   )
INTOFF                 Ignore pulses from IR14.
SETHEX
LCHEX  00120
R0=C                   Store wait time in R0.
A=DAT1 A               Pop dissolve speed
D1=D1+ 5               off stack.
R1=A                   Store dissolve speed into R1.
LCHEX  00080           Clear outputs and set OR7 high
                       to enable ZCD input thru the
                       AND gate.
OUT=C
LCHEX  00004           Constant in D is used to
D=C A                  uniformly extend the dissolve
                       time.
-------------------------------------------------------
SETUP
   A=R0                Load wait time into A.
   C=R1                Load dissolve speed into C.
   D=D-1  A            Don't go on to change
   GONC   LOOP1        parameters unless D=0.
```

```
    A=A-C   A              Decrement wait time.
    GOC     EXIT           Exit if at full brightness.
    LCHEX   00004
    D=C     A              Reset D counter.
    R0=A                   Store new wait time in R0.
LOOP1
    C=IN                  *
    ?C=0    A             * Wait for zero crossing.
    GOYES   LOOP1         *
    C=0     A
    OUT=C                  Shut off bulb.
    LCHEX   380            Load bit mask to turn either
                           lamp on and re-enable ZCD
                           inputs.
LOOP2
    A=A-1   A              Delay for pre-determined
    GONC    LOOP2          amount of time.
    OUT=C                  Turn lamp on.
    GOTO    SETUP          That was 1/120 of a second; go
    EXIT                   back and do the whole thing
    RTNCC                  again.
    END
```

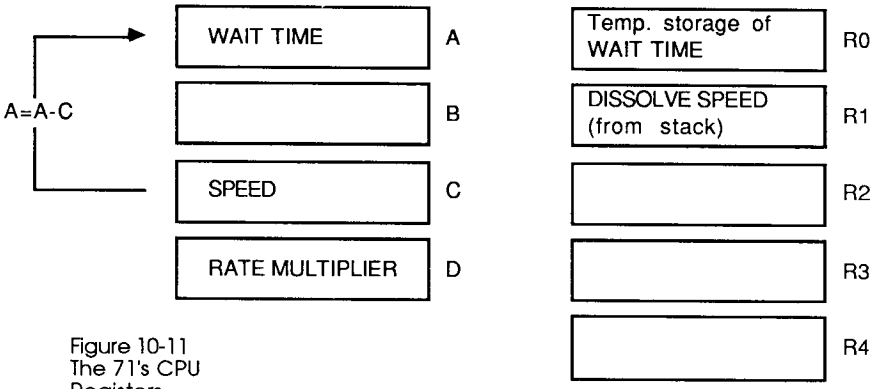


Figure 10-11
The 71's CPU Registers.

IL Interface

Despite all the hardware modifications for I/O mentioned above, an 8-bit port is still needed for two important functions in this system: Touch Tone signal decoding, and slide projector advance. (The hardware for both functions have been covered in previous chapters.) Briefly, sending a "1" or a "2" to the converter and a "0" immediately afterward will cause the circuitry in Fig. 10-12 to "press a button" and advance either projector 1 or projector 2. The Touch Tone (which, by the way, is still a registered trademark of AT+T) decoder IC as shown in Fig. 10-13 is used to tell the 71 not only when to dissolve to the next picture, but also how quickly.

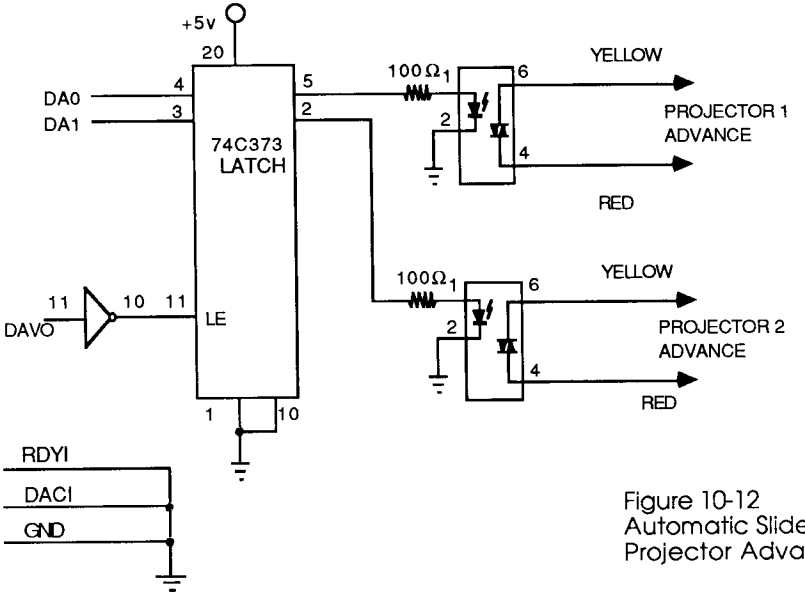


Figure 10-12
Automatic Slide
Projector Advance.

Because I wanted the flexibility to have complete control over the timing parameters and have the system automated as well, a scheme was devised where the 71 could either get its input "live" via an external keypad, or taped so the presentation could forever be synchronized with music. Once again Touch Tone Technology (or "T-cubed" for short) is employed to meet all the requirements.

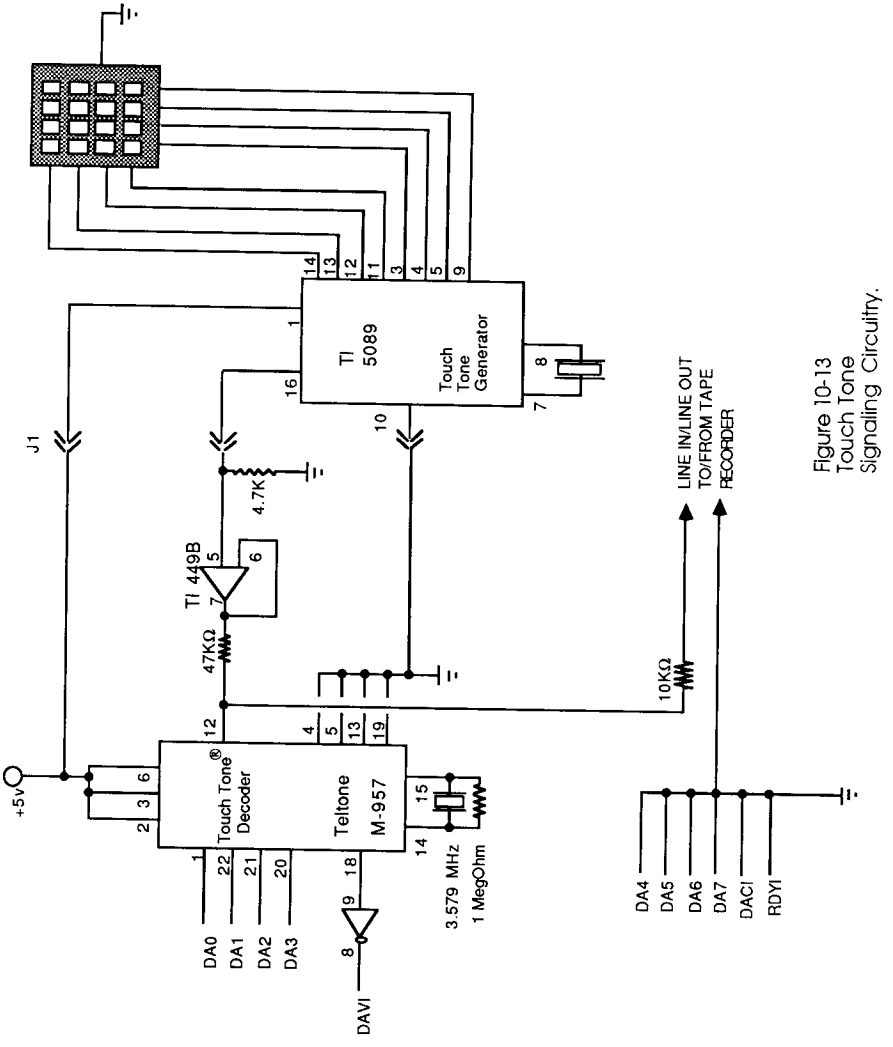


Figure 10-13
Touch Tone
Signaling Circuitry.

Fig. 10-13 shows a familiar pairing: the M-957 Touch Tone decoder chip on the left, and a Touch Tone generator (manufactured by Texas Instruments; just trying to be different) on the right. The decoder chip, which is attached to Data Bus A the same way it was in Chapters 6 and 7, is the sole source of input to the 71. This chip's input, however, can come from one of two sources: the generator chip on the right (with its accompanying keypad), or from the LEFT CHANNEL-LINE OUT output from a stereo tape recorder. (The right channel, of course, contains music or narration or whatever.)

Because of a severe impedance mis-match between the audio LINE IN/LINE OUT feed and the Touch Tone Generator chip, we must also add a simple-looking circuit called an impedance-matching network. The network is used to generate or receive signals from either of two sources: 1) Using line in/line out VU levels from standard low-impedance audio equipment and 2) the-ultra-high impedance Touch Tone generator chip, allowing both signal sources to co-exist without the low impedance source "sucking up" the signal from the high impedance one. In addition, this network must correctly attenuate the generator chip's output so a signal traveling to LINE IN will be at standard VU levels for optimum signal recording.

The solution appears in Fig. 10-13 in the form of the 10K, 47K, & 4.7K resistors and an op amp. The op amp is 1/2 of the 449 dual op amp, whose other half we used while constructing the zero-crossing detector.

Connections to the slide projector can be a bit tricky. Although rumor has it that you can call Kodak and ask them for a 7-pin connector with cable, I found that I could get it done faster by merging a spare remote control cable and one of those universal car stereo power connectors, available at your local electronics outlet (See Fig. 10-14).

All in all, the circuitry is shown complete in Fig. 10-15.

The Software

The Driver program, written in FORTH, is shown on the next page.

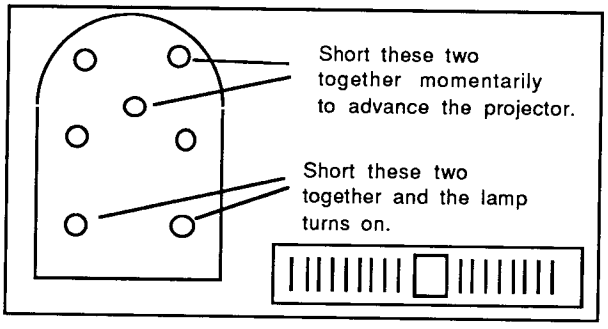


Figure 10-14
Kodak's slide
projector connector.

```
VARIABLE IO
: ADVANCE DUP 1+ IO ! IO 1 OUTPUT 100 0 DO LOOP 0 IO !
IO 1  OUTPUT ;
BASE HEX
: SLIDES " CLEARLOOP@STANDBY ON" BASICX "
DEVADDR('GPIO')" BASICI PRIMARY !
0 IO ! IO 1 OUTPUT
0 IO 1 ENTER DROP @ FADEON
BEGIN IO 1 ENTER DROP @
     DUP C = IF DROP IO 1 ENTER DROP @ FADEOFF "
CLEARLOOP" BASICX -1 ABORT"      Enjoy the show?  "
     ELSE DUP B = IF DROP FLASH
     ELSE DUP 0 = IF DROP DROP
     ELSE DISSOLVE 100 0 DO LOOP ADVANCE THEN THEN
0 UNTIL ;
```

Basically, (or in this case FORTHly), it uses the ENTER command to wait for a Touch Tone input from either the keypad or an audio track. If it's =0 it's discarded to avoid an infinite loop; if it's =B (the * key was hit) it executes the FLASH primitive which just switches projectors without changing anything else. If it's =C (the # has been hit), this means that the next signal will be the last slide and therefore should only fade the current bulb off. If none of these

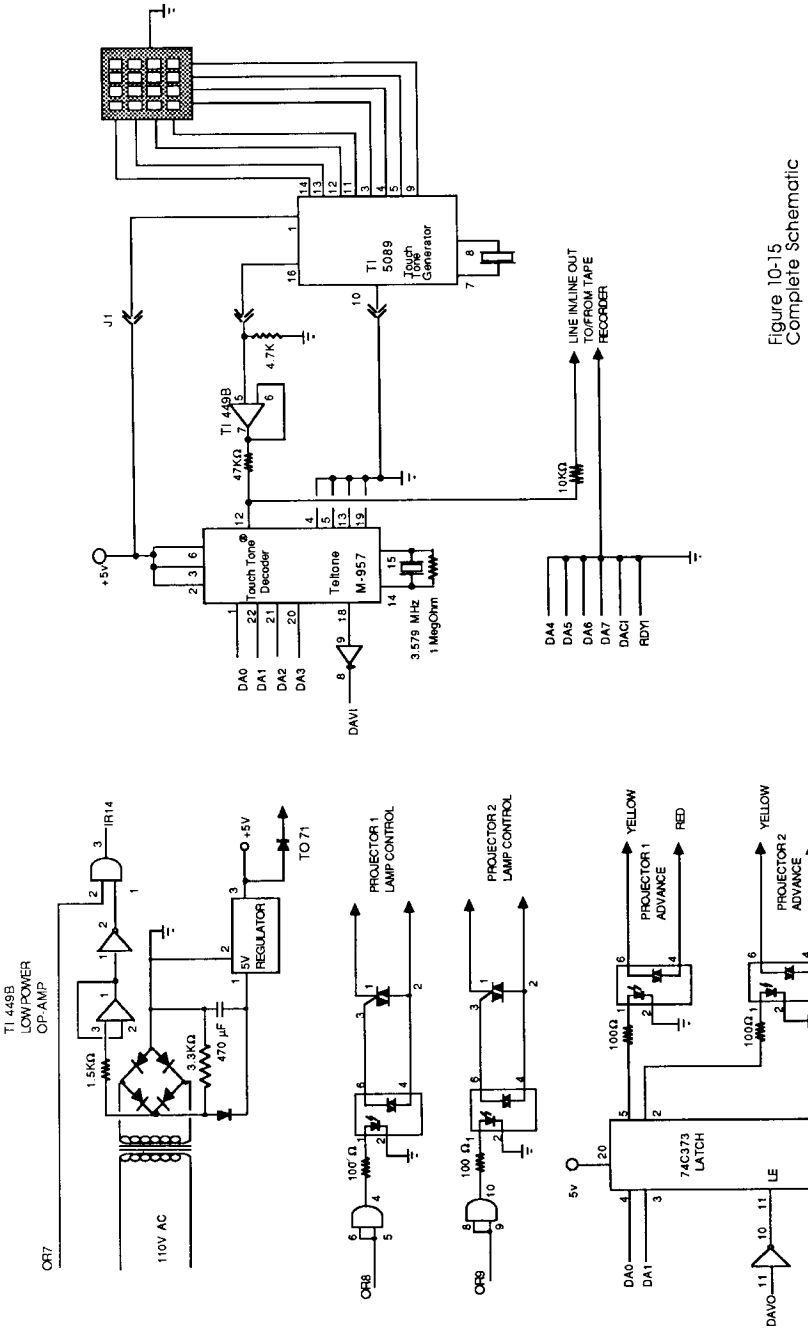


Figure 10-15
Complete Schematic

special conditions exist, then it's an ordinary dissolve, and the primitive DISSOLVE is fed the current projector status ('0'= projector 1 currently lit, '1' = projector 2 currently lit) and the dissolve speed (1-15, 1 being slowest) from the stack. When DISSOLVE returns, the new projector status is left floating on the stack. We wait from 1 to 100 to let it breath, then ADVANCE the proper projector.

That was the easy part. Now for the primitives.

The Primitives

There are five primitives in all, and they have all been merged into one big text file. They function very similarly to the single-bulb example given a few pages ago, except these must now work two bulbs, each having a different intensity at any given moment.

To load these, EDTEXT DISS4TH and enter the listings below into a text file. The comments need not be added, but the leading three spaces on most lines are critical!

Next, EDTEXT DRIVER and enter the driver program listed above. Finally, go into FORTH and enter the following command:

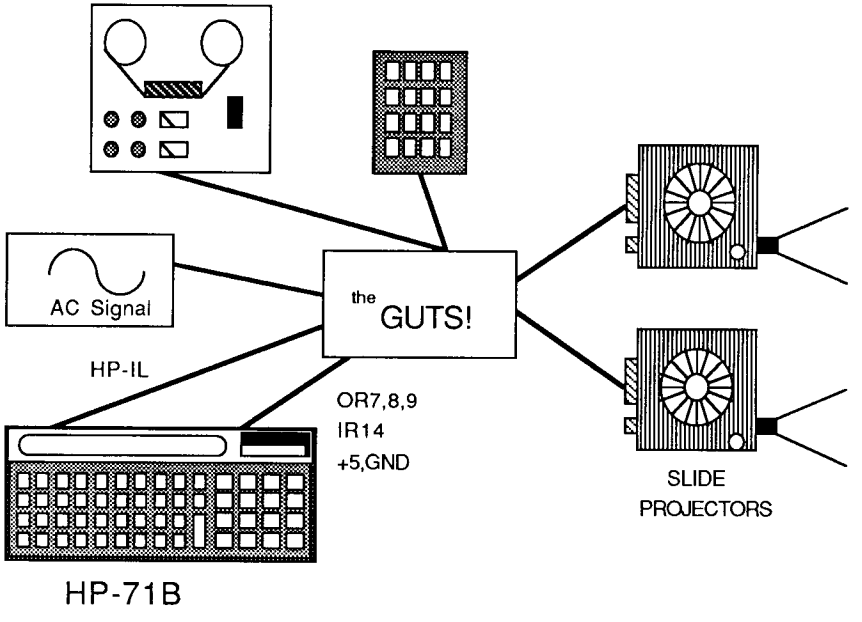
```
" DISS4TH" ASSEMBLE " DRIVER" LOADF <ENDLINE>
```

(watch the spaces!) which loads everything in the proper order into your FORTHRAM file. Save the FORTHRAM file onto disk; that way running this application in the future requires you only load the FORTHRAM file and type SLIDES <ENDLINE>.

```
WORD 'FLASH'          *** SWITCHES PROJECTORS ***
C=DAT1 A              (Old Status -- New Status)
D1=D1+ 5              Pop lamp status off stack.
?C=0   A              Is lamp #1 on?
GOYES  OTHER1         Branch here to handle it.
LCHEX  100            Set bits to turn lamp #1 on.
OUT=C                 Implement it.
R4=C                  Save OUT register status in
                      R4.
```

```
   C=0 A                Push new status (=0,indicating
   D1=D1- 5             lamp #1 is on) onto stack.
   DAT1=C A
   RTNCC                Return carry clear.
OTHER1
   LCHEX  200
   OUT=C                Enable zero crossing.
   R4=C                 Store OUT status in R4.
   LCHEX  001           Set status bit to 1,
                        indicating lamp #2 is on.
   D1=D1- 5             Push this value onto stack.
   DAT1=C X
   RTNCC                Return carry clear.
      *
   WORD 'DISSOLVE'      (Lamp Status  Time --  Lamp
                        Status)
   INTOFF
   C=DAT1 A
   D1=D1+ 5             Pop dissolve time off stack
                        and into C.
   A=DAT1 A             Pop projector status
   D1=D1+ 5             off the stack and into A.
   ?A#0   A             Is lamp #1 on? (A<>0?).
   GOYES  OTHER         Yes, branch to OTHER.
   R0=C                 Store # on stack into R0.
   C=-C   A             Take the 2's complement of C
   R3=C                 and store it in R3.
   LCHEX  00095         095 = wait time of lamp #2.
                        (0% brightness).
   R2=C                 Store it in R2.
   C=0    A             0 = wait time of lamp #1.
                        (100% brightness).
   R1=C                 Store it in R1
   ST=0   7             Set status bit to show lamp #1
                        is now on.
     *                  (Well, it's about to be,
                        anyway!)
   LCHEX  280           Turn on lamp #2 and enable
   OUT=C                interrupts.
   D=0 X                Clear wait loop.
   GOTO   START         Begin dissolve routine.
```

```
OTHER
   R3=C                    Lamp #1 must have been on.
   C=-C    A               Store number in R3, and the
   R0=C                    2's complement into R0.
   LCHEX   00095           095 = wait time of lamp #1 (0%
                           brightness).
   R1=C                    Store it in R1.
   C=0     A               0 = wait time of lamp #2 (100%
                           brightness).
   R2=C                    Store it into R2.
   ST=1    7               Set status bit to show lamp #2
                           is now on.
   LCHEX   180             Turn on lamp #1 and enable
                           interrupts.
   OUT=C
   D=0     X               Delay loop index =0
    *
    *                      *SETUP REGISTERS*
```



HP-71 Controlled Slide Projector Dissolve Unit

```
     *
START
     D=D-1   X              Decrement slowdown index.
     GONC    START1         Repeat same old values if we
                            haven't looped N times.
     LCHEX   004            If we have, reset D register.
     CDEX    X
     C=R1                   Load C with wait time for lamp
                            #1.
     A=R0                   Load A with the constant to
                            add.
     C=C+A   A              Add A and C, and store as new
     R1=C                   wait time.
     C=R2                   Load C with wait time of lamp
                            #2.
     A=R3                   Load A with the constant to
                            add.
     C=C+A   A              Add, store new lamp #2 wait
     R2=C                   time.
START1
     C=0     A              Clears debris not covered by
                            OUT.
LOOP0
     C=IN               *
     ?C=0    A          * wait for zero crossing pulse.
     GOYES   LOOP0      *
     LCHEX   00080          Shut both lamps off, but keep
     OUT=C                  zero-crossing input enabled.
     R4=C                   Store status in R4.
     A=R1                   Load A with lamp #1 wait time.
     C=R2                   Load C with lamp #2 wait time.
LOOP1
     A=A-1   A              Decrement lamp #1 wait time.
     GOC     ON1            If 0 hit, branch to turn lamp
                            #1 on.
LOOP2
     C=C-1   A              Decrement lamp #2 wait time.
     GOC     ON2            If 0 hit, branch to turn lamp
                            #2 on.
     GOTO    LOOP1          Loop again.
     *
```

```
CONTINUE
   ?ST=0   7          Is lamp #1 being turned to full
                      power?
   GOYES   CONT2      If so, branch to CONT2.
   LCHEX   00095
   CAEX    A
   C=R2               Load wait time for lamp #2.
   A=A-C   A          Subtract current wait time
                      from the maximum.
   C=R3               Compare with step size.
   ?C>A    A          Is difference less than step
                      size?
   GOYES   EXIT       Yes, exit (lamp #2 is at full
                      intensity).
   GOTO    START      Otherwise, prepare for another
                      cycle.
CONT2
   LCHEX   00095      Load maximum wait time
   CAEX    A          into A.
   C=R1               Load current wait time
   A=A-C   A          and subtract.
   C=R0               R3= current step size.
   ?C>A    A          Is difference < step size?
                      (lamp #1 = full intensity?)
   GOYES   EXIT       Yes, exit.
   GOTO    START      Otherwise, prepare for another
                      cycle.
     *
ON1
   CR4EX              Recall R4 and store C away
                      temp. in R4.
   CAEX    A          Move C into A.
   LCHEX   100
   C=C!A   A          Flip lamp #1 bit to 'ON' and
   OUT=C              implement new status.
   CAEX    A          Move new OUT contents into A
                      register.
   LCHEX   380        What OUT would be if both
                      lamps were on.
   ?A=C    X          Are both lamps on?
   GOYES   CONTINUE   Yes, break out of loop.
```

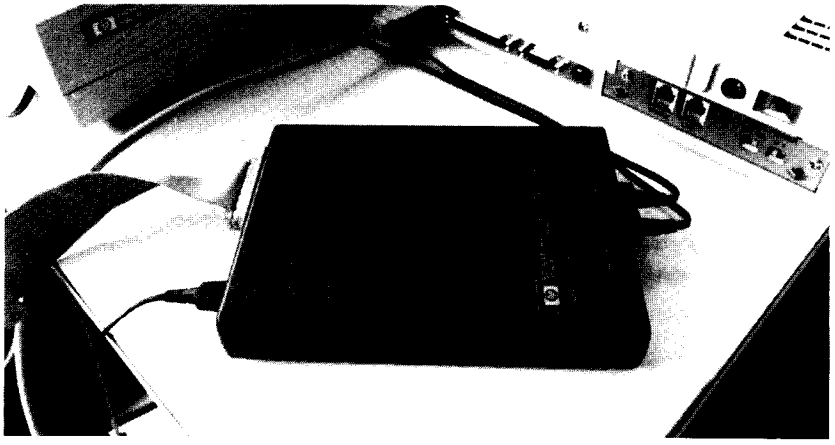
```
   CAEX A                  OUT status moved back to C,
                           and 380 (which is < 095) into
                           A.
   CR4EX                   OUT into R4, previous C status
                           into C.
   GOTO   LOOP2            Continue counting.
     *
ON2
   AR4EX                   A preserved in R4, OUT into A.
   LCHEX  200          *
   C=C!A  A            * Flip lamp #2 bit to 'on'.
   OUT=C               *
   CAEX   A                OUT status goes to A.
   LCHEX  380
   ?A=C   X                Are both lamps on?
   GOYES  CONTINUE         Yes, break out of loop.
   AR4EX                   OUT goes to R4, prev A goes to
                           A, C=380 (which is >095!)
   GOTO   LOOP1
     *
EXIT
   ?ST=0  7                Is lamp #1 to be left on?
   GOYES  EXIT2            Yes, branch to EXIT2.
   LCHEX  100              Turn lamp #1 on.
   OUT=C
   C=0 A                   Push status
   D1=D1- 5                (which =0 = lamp #1 on)
   DAT1=C A                onto the stack.
   RTNCC                   Ahhh...Gooba!
EXIT2
   LCHEX  200              Turn lamp #2 on.
   OUT=C
   LCHEX  00001            Push status bit
   D1=D1- 5                (which =1 = lamp #2 on)
   DAT1=C A                onto the stack.
   RTNCC                   So long!
   END

   FORTH
   WORD 'FADEOFF           (Lamp status  Time --   )
   INTOFF                  *** LAST COMMAND IN SLIDE
```

```
                          SHOW. ***
   SETHEX                 *** FADES CURRENT LAMP OFF.***
   LCHEX  00120
   R2=C
   LCHEX  00005
   R0=C                   Store wait time in R0.
   A=DAT1 A               Pop dissolve speed
   D1=D1+ 5               off stack.
   R1=A                   Store dissolve speed in R1.
   C=DAT1 A               Pop lamp status
   D1=D1+ 5               off stack.
   ?C=0 A                 Is lamp #1 on?  If so, branch
   GOYES  LAMP2           here to handle different
                          setup.
   LCHEX  280             Bit map for lamp #2.
   R4=C                   Store in R4.
   GOTO   FADE2
LAMP2
   LCHEX  180             Bit map for lamp #1.
   R4=C                   Store it in R4.
FADE2
   LCHEX  080             Enable zero crossings.
   OUT=C
   LCHEX  00004           Constant in D increases
   D=C    A               dissolve length.
SETUP
   A=R0                   Load wait time into A.
   C=R1
   D=D-1  A
   GONC   LOOP1
   A=A+C  A               Decrement wait time.
   C=R2
   ?A>=C  A
   GOYES  EXIT            Exit if at full brightness.
   LCHEX  00004
   D=C A                  Reset D counter.
   R0=A                   Store new wait time in R0.
LOOP1
   C=IN
   ?C=0   A               Wait for zero crossing.
   GOYES  LOOP1
```

```
   LCHEX  080
   OUT=C                  Shut off bulb, enable zero
   C=R4                   crossings.
LOOP2
   A=A-1  A               Delay for pre-determined
   GONC   LOOP2           amount of time.
   OUT=C
   GOTO   SETUP           Start again.
EXIT C=0  A               Shut off both bulbs
   OUT=C                  and disable zero crossings.
   RTNCC                  The show's over.
   END
```

Although I'm not a professional photographer, this dissolve unit has been a reliable tool that has enhanced my slide presentations tremendously. But even if you're not heavily into photography, you have learned one method of how computers can control the analog world, and most of all have obtained a better idea of the extended control capabilities of the 71.

Special thanks goes to Mr. Sergio Morales for his theoretical guidance and helpful suggestions while designing this project.



Chapter Eleven

AN INTRODUCTION TO RS-232

Standards are wonderful--Everyone should have one of his own.
--Anonymous

The HP82164A HP-IL/RS-232-C Interface is a most confusing device to use; not solely due to the haphazard ways it must be programmed by the controller; but also due to the confusing way RS-232 has evolved.

RS-232 is perhaps the most non-standard "standard" in the world, mostly since in the beginning each manufacturer had their own idea of how it ought to work and implemented it that way. In general, the only thing guaranteed about RS-232 is that it won't work on the first try.

If you want to interface your computer to ANY RS-232 device, a billion (well, maybe not) configuration options must be available, and the user had better know what they all mean in order to not give up the very first week. This is one area where HP-IL outshines RS-232 in terms of I/O for personal computers: The most basic functions like printers, displays, etc., are already taken care of for you and will always work on the first try, so a user never need concern themselves with low-level operation or connection thereof. (Other functions, admittedly, require knowledge of the protocol which is a little more complicated; but most people will never have to worry about it. You, having read this book and therefore wanting to implement the more difficult stuff, are obviously the exception.)

When a byte is sent by RS-232, only one wire is used instead of eight, and the information is transmitted serially (one bit after the other) rather than in parallel as with the GPIO. On an RS-232

link, each transmitted byte is also accompanied by the information in Fig. 11-1:

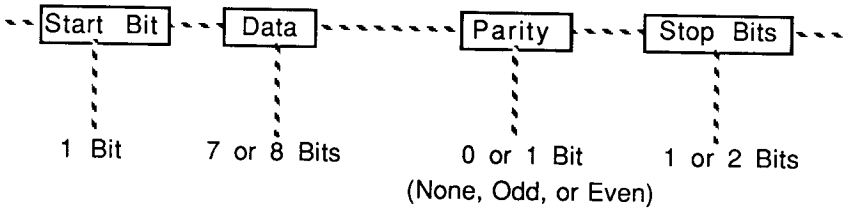


Figure 11-1
What Else is Sent Along with a Byte of Data

And the bit stream would look like this:

```
0110110011100110011001101111010011001101100111001100110
```

The data sent can be represented either by 7 bits (which is all that's required to represent the entire ASCII-defined character set) or 8 bits (the default nowadays, as it allows a full character set as well as a wide assortment of cursor control characters and graphics commands to be sent).

The Start bit is used to get the receiver ready to receive data by giving it something on which to synchronize. Even though the transmitting speed is known at the time of data transmission, the synchronization function must still be performed.

The parity bit is a simple form of error checking and it works like this: when Even parity is specified, the transmitting program adds up the total number of "1"s found in the data field. If there is an odd number of "1"s counted, the parity bit is set to 1; it is zero otherwise. Odd parity is just the opposite; the parity bit is set to "1" if there is an even (flagging an error) number of 1's counted. The idea behind the inclusion of a parity bit is to be able to recover from the most common type of transmission error: when a single data bit is missing. The receiving device (printer, modem, etc.) must, after receiving every byte, verify that the parity bit correctly

describes the number of 1's in the data field. (If it doesn't match, the device is supposed to request the computer to retransmit the last byte via established protocol techniques.)

The stop bits simply signify the end of a transmitted byte. Either 1 or 2 bits is necessary to perform this function.

Because of the variety of valid options listed above, it is imperative that both the sender and receiver use the same format for representing data. Most computers have unfriendly configuration routines where you must answer prompts requesting how many data bits, what kind of parity (Odd, Even, or None?), etc. Other devices, such as the HP-IL/RS-232 Converter, must be configured via software (even more unfriendly than the above) to format the information the correct way. In many cases, it does not matter what you set the parameters to, as long as both devices on each side of the RS-232 line agree.

The HP-IL/RS-232 converter can be software configured in one of two ways: DDT and DDL commands (just like the parallel interface devices), and Remote mode commands. We'll cover those in a few pages, but first we must address the question as to why a computer and a serial printer seldom work right when hooked together on the first try.

Many implementations of RS-232 use only 3 wires as shown in Fig. 11-2:

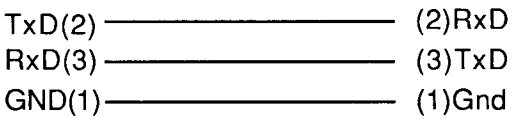


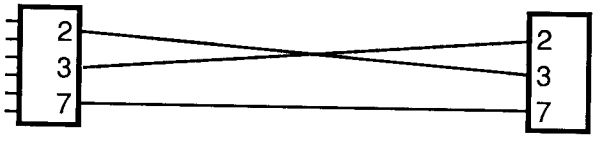
Figure 11-2
3-Wire RS-232
Scheme

The top line, TxD (Transmit Data), is used to send information from the computer to the RxD (Receive Data) line connected to the modem. The middle line performs the same function for information coming from the modem to the computer. Notice the different pinouts on both sides of the connecting cable: TxD is pin 2 on one side, and pin 3 on the other. The original idea behind doing

it this way was that all computers would universally be wired as shown on the left, (called DTE, for Data Terminal Equipment) and all peripherals would be wired up as in the modem on the right (also called DCE, for Data Communications Equipment). That way, only a "straight-through" RS-232 cable would ever be needed to hook any two devices together.

Well, that neat little idea certainly got out of hand. Today, manufacturers of computer equipment may use either DTE or DCE wiring on their RS-232 ports, and will interchangeably use a male or female 25-pin connector, male or female 9-pin connector, a DIN connector, modular phone jack, or anything else they can think of. So as you can see, the only existing standard is that nobody conforms to the existing standard.

As a result of this mass confusion, you will occasionally find that both the computer and the modem manufacturer installed their RS-232 port the same way--usually DTE configuration. The usual solution to this is to use a "null modem cable", which is simply a cable with at least pins 2 and 3 reversed to compensate for the sameness on both sides.



The full implementation of RS-232-C is actually more complicated than the 3-wire scheme described above. Handshaking is included with the serial lines to facilitate a printer that has no data buffering ability, and a reasonable implementation looks like Fig. 11-3 on the next page.

(Notice I said "reasonable implementation". The RS-232 standard actually allows for 20 of its 25-pin connector to be used ("So that's why the connector's so big!") and hooking up one of those can get to be really hairy!) And if both devices are configured as a DTE, a larger null modem cable must be wired up to look like Fig. 11-4.

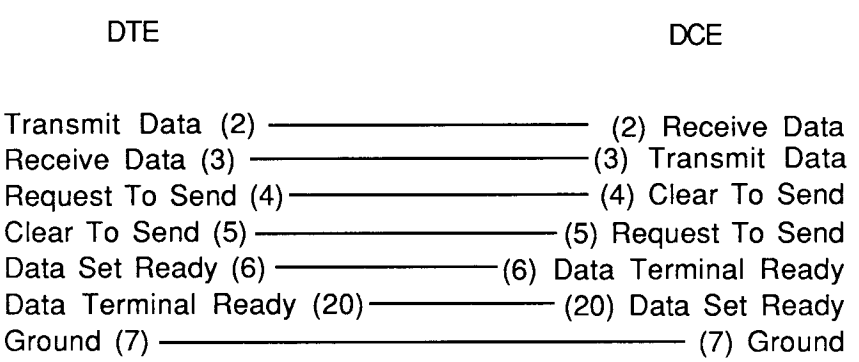


Figure 11-3
Closer to the
RS-232 Standard

Fortunately, null modem cables (usually an expensive item since the manufactures know darn well that the users don't have the time to sit down and fabricate their own) aren't necessary with the RS-232 Converter. Within its confines is what HP calls a configuration selector, which is actually a 16-pin jumper plug. When inserted one way, the device is wired as a DTE. When inserted end-for-end, it becomes a DCE. Voila!

The handshake lines, as diagrammed above, function in precisely the same way as they do for the 8-bit ports described in

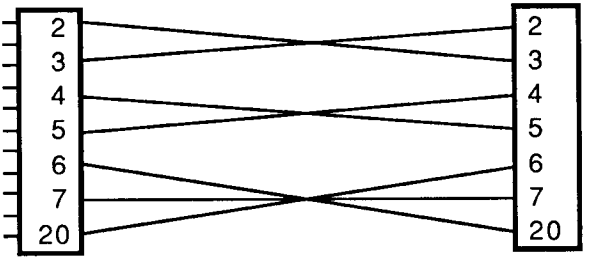


Figure 11-4
Null Modem Cable

Chapter 1. But in the more recent 3-wire applications, handshaking must be taken care of by software. The most common software handshaking scheme is called XON/XOFF, getting its name from the ASCII characters that the device sends in order to start/stop data transmissions from the computer.

It works like this: the device can tell the computer to stop transmitting by sending it an XOFF character (=decimal 19). (Unlike the 8-bit ports, the RS-232 interface allows you to send information in both directions at the same time. If this wasn't the case, the XOFF command would be lost.) If the computer has previously been programmed to respond to XON/XOFF protocol, it will stop transmitting until it receives an XON (= decimal 17) from the device.

Another handshake protocol similar to XON/XOFF is called ENQuire/ACKnowledge, and functions more like the hardware handshake: Before the computer sends a word, it sends the ENQ (= decimal 5) byte. If the device is ready to receive, it acknowledges the inquiry by sending an ACK (= decimal 6) back. Generally, this is done every time a block (an arbitrarily defined amount of bytes) is to be sent.

Well, this sure is getting complicated! The really tough part about interfacing with RS-232 is figuring out which combinations of options to throw together so the two ends of the cable will act harmoniously. (The toughest job of all was for HP, for they had to make a device that was versatile enough to hook up to ANY implementation of this molested standard.) Sometimes, as with hooking up a printer or modem, the communications parameters are designed in, so all you have to do is program the interface to match what the printer or modem is configured for. Fortunately, this aspect of interfacing is straightforward.

The HP-IL/RS-232 Interface can have its communications parameters software-specified by two different methods: DDL (Device Dependent Listen) commands as explained in Chapter 1, or Remote commands. Both are equally easy to implement. The control register descriptions can be found in Appendix D of the RS-232 Interface's Owner's Manual. Likewise, the Remote commands list starts on pg. 37 of the same manual. An example of each follows using the 71 as a loop controller.

```
10 ! PRTDRV print driver for the LaserJet Printer.
20 A=DEVADDR("RS232")
30 SEND UNT UNL LISTEN A MTA DDL 0 DATA 0,0,0,0,0,0,0,
   14 UNT UNL
40 PRINTER IS :RS232
```

In the example above, the interface is programmed via the DDL 0 command. The subsequent data bytes specify the following attributes:

Bytes 0-2 (R0-R2) are all 0, signifying that we don't care about service request conditions in this example.

Byte 3 is 0, indicating that we will not be deleting/replacing special characters in the data stream.

Byte 4 is 0, disabling the handshake lines that the LaserJet printer never pays attention to.

Byte 5 can be anything, since it only shows the status of three of the unused handshake lines when a STATUS is requested from the host. In this case, it was left as 0.

Byte 6 is 0, indicating one stop bit, eight data bits, and no parity.

Byte 7 is 14, indicating a transmitting speed of 9600 bits per second.

No more DATA bytes were sent, indicating that the default values for the remaining registers should suffice.

(A quick re-examination of Appendix D indicates that all the default values in the control registers were exactly what we specified; meaning the first three lines in the above program weren't needed at all. (This is a very rare case, mind you. The example was included here to illustrate more normal cases.)

In addition to the software configuration, the Interface must physically be changed to look like a DCE by reversing the control selector jumper plug inside. After this is done, only a straight through, pin-to-pin 25 line ribbon cable is needed to join the Interface and the LaserJet. The printer only uses one handshake line, DTR (Data Terminal Ready), to tell the computer that it's not ready to transmit. DSR (Data Set Ready), the computer's pin that receives the DTR signal, automatically reacts to this and doesn't

have to be initiated by the control registers.

The only drawback one might experience with this printer as driven by the 71 is that when anything is printed, the last page is NOT ejected from the printer because the 71 doesn't terminate PLIST commands with a FORM FEED (= decimal 12). Sure, programs that print should be responsible for generating the FORM FEED character, but PLIST and other functions require the following human intervention to complete: 1) Hit the ONLINE button to take the printer offline. 2) Hit the FORM FEED button to eject the last page. 3) Hit the ONLINE button to take the printer back online.

Let's try a different example. This time, we'll attempt to compensate for the 71's lacking keyboard and display by hooking it up to an IBM PC (a machine whose keyboard and display are also lacking, but not as much) and using it as a dumb terminal. Because the RS-232 parameters here are not fixed by either the IBM or the interface, I arbitrarily set them to the following values:

```
PArity      None        No parity bits
DAta Bits     8         8 data bits
SPeed       2400        2400 bits per second (baud)
STop          1         1 stop bit
PMode         1         Needed for Crosstalk
DUplex      Full        Needed for Crosstalk
OUtfilter    On         Needed for Crosstalk

Configuration plug set to DTE  (needed for IBM
RS-232 port)
```

Materials needed: KEYBOARD IS lexfile (available in the highly recommended FORTH/ASSEMBLER ROM), and communications software for the IBM, such as the popular CrossTalk package.

The KEYBOARD IS lexfile is one of those routines that is implemented almost perfectly. The idea was to allow a keyboard of realistic size and feel to provide input, and still allow the key reassignments to take effect. If all you're going to be typing are the alphanumeric characters, this is no problem. It is the f- and

g-shifted keystrokes that provide the difficulty.

HP decided on using 2-character ESCAPE sequences to define any one- or two-character 71 keystrokes by defining an "escape" buffer which is used like this:

```
ESCAPE "A",50
```

After the above assignment is typed into the 71, hitting the ESCAPE key and then the "A" key on the external keyboard will activate key #50 on the 71; the Up Arrow key. (Refer to the 71's Keyboard map in their instruction manual.) (You should have read this stuff in Chapter 8 anyway.)

The wonderful thing about CrossTalk is that any of the keys can be reprogrammed to send out a string of characters rather than just a single character. This way, hitting F1 on the IBM's keyboard will generate the arbitrarily defined two-character escape sequence the 71 expects to see. For this application, the IBM's ten function keys were redefined as follows using Crosstalk's key assignment file (note that Crosstalk interprets '^]' as meaning 'escape'):

F1	^[a
F2	^[b
F3	^[c
F4	^[d
F5	^[e
F6	^[f
F7	^[g
F8	^[h
F9	^[i
F10	^[j

Photo #1 Yet another use for the RS-232 Converter.

and the following 71 program makes use of these and the function key escape sequences normally sent out by Crosstalk:

```
5   ! Program IBM uses XTALK to add keyboard + display
10  RESET HPIL @ CLEAR @ REMOTE
20  OUTPUT :RS232 ;"SE0;SE3;SBA;" @ LOCAL
30  ESCAPE "A",50       ! Assigns 8 to Up Arrow
40  ESCAPE "B",51       ! Assigns 2 to Down Arrow
50  ESCAPE "D",47       ! Assigns 4 to Left Arrow
60  ESCAPE "C",48       ! Assigns 6 to Right Arrow
70  ESCAPE "a",43       ! Assigns F1 to ATTN
80  ESCAPE "b",150      ! Assigns F2 to Command Stack
90  ESCAPE "c",159      ! Assigns F3 to far Left
100 ESCAPE "d",160      ! Assigns F4 to far Right
110 ESCAPE "e",162      ! Assigns F5 to far Up
120 ESCAPE "f",163      ! Assigns F6 to far Down
130 ESCAPE "i",105      ! Assigns F9 to I/R
140 ESCAPE "j",104      ! Assigns F10 to -CHAR
150 KEYBOARD IS :RS232 @ DISPLAY IS :RS232 @ PRINTER
    IS :RS232
160 END
```

Line 20 above is the crucial one. Using Remote mode programming, it specifies the following parameters:

SE0: Disables all service requests.
SE3: Specifies service requests when the Receive buffer isn't empty.
SBA: Specifies 2400 baud, because characters get lost at higher speeds.

The other parameters-- one stop bit, No parity, eight data bits-- are all default settings and therefore don't need to be specified by the program.

When using this IBM-to-71 system, there are, unfortunately, a number of disadvantages. First is the IBM's keyboard, whose shift and return keys are located where no one would expect to find

them. Second is the slight incompatibility of the cursor commands between the 71's display and the IBM's. For example, the I/R and -CHAR functions delete the current line from the IBM's screen before performing their functions. And when a line longer than 80 columns is displayed onto the screen, the wraparound function works perfectly; but when the cursor is repositioned back to the beginning by a series of backspace commands, the IBM's cursor goes only to the beginning of the wraparound line.

By all admission, it ain't perfect. If you want perfection, see Chapter 8.

APPENDICES

Appendix A

BARCODE FOR 41 PROGRAMS

CHAPTER 3

PROGRAM:CAMERA 35 REGISTERS PROGRAM USES 19 ROWS

ROW 1 LINES (1-3)

ROW 2 LINES (3-6)

ROW 3 LINES (6-8)

ROW 4 LINES (8-14)

ROW 5 LINES (14-19)

ROW 6 LINES (20-24)

ROW 7 LINES (24-28)

ROW 8 LINES (28-32)

ROW 9 LINES (33-40)

ROW 10 LINES (41-45)

PROGRAM:CAMERA

ROW 11 LINES (46-51)

ROW 12 LINES (51-54)

ROW 13 LINES (55-55)

ROW 14 LINES (56-60)

ROW 15 LINES (61-71)

ROW 16 LINES (71-72)

ROW 17 LINES (72-78)

ROW 18 LINES (78-82)

ROW 19 LINES (82-85)

CHAPTER 4

PROGRAM:DKRM3 91 REGISTERS PROGRAM USES 49 ROWS

ROW 1 LINES (1-2)

ROW 2 LINES (2-6)

ROW 3 LINES (6-11)

ROW 4 LINES (11-15)

ROW 5 LINES (16-23)

ROW 6 LINES (23-29)

ROW 7 LINES (29-36)

ROW 8 LINES (37-42)

ROW 9 LINES (42-49)

ROW 10 LINES (49-53)

ROW 11 LINES (53-59)

ROW 12 LINES (60-70)

ROW 13 LINES (70-77)

ROW 14 LINES (77-81)

ROW 15 LINES (82-88)

PROGRAM:DKRM3

ROW 16 LINES (88-94)

ROW 17 LINES (94-100)

ROW 18 LINES (100-106)

ROW 19 LINES (106-111)

ROW 20 LINES (112-118)

ROW 21 LINES (118-123)

ROW 22 LINES (123-130)

ROW 23 LINES (130-133)

ROW 24 LINES (133-139)

ROW 25 LINES (140-143)

ROW 26 LINES (143-147)

ROW 27 LINES (147-153)

ROW 28 LINES (154-158)

ROW 29 LINES (158-161)

ROW 30 LINES (161-167)

PROGRAM:DKRM3

ROW 31 LINES (168-171)

ROW 32 LINES (172-180)

ROW 33 LINES (180-186)

ROW 34 LINES (187-190)

ROW 35 LINES (191-198)

ROW 36 LINES (199-204)

ROW 37 LINES (205-207)

ROW 38 LINES (208-210)

ROW 39 LINES (211-215)

ROW 40 LINES (215-222)

ROW 41 LINES (223-232)

ROW 42 LINES (232-240)

ROW 43 LINES (241-247)

ROW 44 LINES (247-253)

ROW 45 LINES (254-261)

PROGRAM:DKRM3

ROW 46 LINES (261-268)

ROW 47 LINES (268-273)

ROW 48 LINES (273-279)

ROW 49 LINES (280-285)

PROGRAM:DKRM4 138 REGISTERS PROGRAM USES 74 ROWS

ROW 1 LINES (1-2)

ROW 2 LINES (2-9)

ROW 3 LINES (9-16)

ROW 4 LINES (17-23)

ROW 5 LINES (23-28)

ROW 6 LINES (28-32)

ROW 7 LINES (33-40)

ROW 8 LINES (40-46)

ROW 9 LINES (46-52)

ROW 10 LINES (52-58)

ROW 11 LINES (58-65)

ROW 12 LINES (65-70)

ROW 13 LINES (70-74)

ROW 14 LINES (74-77)

ROW 15 LINES (77-84)

PROGRAM:DKRM4

ROW 16 LINES (85-94)

ROW 17 LINES (95-104)

ROW 18 LINES (105-114)

ROW 19 LINES (114-121)

ROW 20 LINES (121-126)

ROW 21 LINES (127-133)

ROW 22 LINES (133-138)

ROW 23 LINES (139-144)

ROW 24 LINES (145-149)

ROW 25 LINES (150-157)

ROW 26 LINES (157-161)

ROW 27 LINES (161-168)

ROW 28 LINES (169-173)

ROW 29 LINES (173-180)

ROW 30 LINES (180-186)

PROGRAM:DKRM4

ROW 31 LINES (187-189)

ROW 32 LINES (190-193)

ROW 33 LINES (193-199)

ROW 34 LINES (200-206)

ROW 35 LINES (206-210)

ROW 36 LINES (211-218)

ROW 37 LINES (218-221)

ROW 38 LINES (222-225)

ROW 39 LINES (226-232)

ROW 40 LINES (232-236)

ROW 41 LINES (236-244)

ROW 42 LINES (245-251)

ROW 43 LINES (251-255)

ROW 44 LINES (255-262)

ROW 45 LINES (263-268)

PROGRAM:DKRM4

ROW 46 LINES (269-277)

ROW 47 LINES (278-286)

ROW 48 LINES (287-295)

ROW 49 LINES (295-302)

ROW 50 LINES (303-310)

ROW 51 LINES (311-319)

ROW 52 LINES (319-327)

ROW 53 LINES (328-331)

ROW 54 LINES (331-338)

ROW 55 LINES (338-346)

ROW 56 LINES (347-355)

ROW 57 LINES (355-362)

ROW 58 LINES (363-369)

ROW 59 LINES (370-376)

ROW 60 LINES (377-383)

PROGRAM:DKRM4

ROW 61 LINES (384-389)

ROW 62 LINES (389-395)

ROW 63 LINES (395-403)

ROW 64 LINES (403-411)

ROW 65 LINES (411-418)

ROW 66 LINES (419-424)

ROW 67 LINES (425-437)

ROW 68 LINES (437-440)

ROW 69 LINES (440-444)

ROW 70 LINES (445-451)

ROW 71 LINES (451-457)

ROW 72 LINES (458-463)

ROW 73 LINES (463-466)

ROW 74 LINES (466-468)

CHAPTER 5

PROGRAM:HAPPY 34 REGISTERS PROGRAM USES 18 ROWS

ROW 1 LINES (1-4)

ROW 2 LINES (4-5)

ROW 3 LINES (6-6)

ROW 4 LINES (7-8)

ROW 5 LINES (8-11)

ROW 6 LINES (11-13)

ROW 7 LINES (13-15)

ROW 8 LINES (15-20)

ROW 9 LINES (21-26)

ROW 10 LINES (26-33)

ROW 11 LINES (33-36)

ROW 12 LINES (36-38)

ROW 13 LINES (38-40)

ROW 14 LINES (40-42)

ROW 15 LINES (43-47)

PROGRAM:HAPPY
ROW 16 LINES (47-49)

ROW 17 LINES (49-50)

ROW 18 LINES (51-53)

PROGRAM:TIMED 25 REGISTERS PROGRAM USES 14 ROWS

ROW 1 LINES (1-3)

ROW 2 LINES (4-12)

ROW 3 LINES (12-20)

ROW 4 LINES (20-28)

ROW 5 LINES (29-37)

ROW 6 LINES (38-44)

ROW 7 LINES (44-46)

ROW 8 LINES (46-51)

ROW 9 LINES (51-58)

ROW 10 LINES (59-68)

ROW 11 LINES (68-76)

ROW 12 LINES (77-85)

ROW 13 LINES (86-93)

ROW 14 LINES (94-96)

CHAPTER 7

PROGRAM:INTT2 35 REGISTERS PROGRAM USES 19 ROWS

ROW 1 LINES (1-2)

ROW 2 LINES (2-6)

ROW 3 LINES (7-13)

ROW 4 LINES (14-20)

ROW 5 LINES (21-26)

ROW 6 LINES (27-34)

ROW 7 LINES (35-41)

ROW 8 LINES (42-49)

ROW 9 LINES (50-56)

ROW 10 LINES (57-64)

ROW 11 LINES (65-72)

ROW 12 LINES (73-80)

ROW 13 LINES (80-88)

ROW 14 LINES (88-95)

ROW 15 LINES (96-103)

PROGRAM:INTT2
ROW 16 LINES (104-111)

ROW 17 LINES (112-119)

ROW 18 LINES (119-127)

ROW 19 LINES (127-130)

PROGRAM:GPIO 10 REGISTERS PROGRAM USES 6 ROWS

ROW 1 LINES (1-2)

ROW 2 LINES (2-8)

ROW 3 LINES (9-16)

ROW 4 LINES (16-20)

ROW 5 LINES (21-24)

ROW 6 LINES (24-24)

PROGRAM:ANSWER 61 REGISTERS PROGRAM USES 33 ROWS

ROW 1 LINES (1-2)

ROW 2 LINES (2-8)

ROW 3 LINES (9-15)

ROW 4 LINES (16-22)

ROW 5 LINES (22-28)

ROW 6 LINES (28-35)

ROW 7 LINES (35-41)

ROW 8 LINES (42-50)

ROW 9 LINES (51-59)

ROW 10 LINES (60-65)

ROW 11 LINES (65-69)

ROW 12 LINES (70-76)

ROW 13 LINES (77-82)

ROW 14 LINES (83-85)

ROW 15 LINES (85-89)

PROGRAM:ANSWER

ROW 16 LINES (89-93)

ROW 17 LINES (94-101)

ROW 18 LINES (101-104)

ROW 19 LINES (104-111)

ROW 20 LINES (111-117)

ROW 21 LINES (117-119)

ROW 22 LINES (119-126)

ROW 23 LINES (126-129)

ROW 24 LINES (129-134)

ROW 25 LINES (134-141)

ROW 26 LINES (142-146)

ROW 27 LINES (147-152)

ROW 28 LINES (152-158)

ROW 29 LINES (159-166)

ROW 30 LINES (167-174)

PROGRAM:ANSWER
ROW 31 LINES (175-181)

ROW 32 LINES (182-185)

ROW 33 LINES (186-189)

PROGRAM:INTT4 10 REGISTERS PROGRAM USES 6 ROWS
ROW 1 LINES (1-3)

ROW 2 LINES (3-11)

ROW 3 LINES (11-19)

ROW 4 LINES (19-27)

ROW 5 LINES (27-33)

ROW 6 LINES (33-34)

PROGRAM:4132 13 REGISTERS PROGRAM USES 7 ROWS

ROW 1 LINES (1-5)

ROW 2 LINES (5-8)

ROW 3 LINES (8-11)

ROW 4 LINES (11-13)

ROW 5 LINES (13-17)

ROW 6 LINES (17-21)

ROW 7 LINES (21-23)

PROGRAM:4111 23 REGISTERS PROGRAM USES 13 ROWS

ROW 1 LINES (1-3)

ROW 2 LINES (3-5)

ROW 3 LINES (6-14)

ROW 4 LINES (15-17)

ROW 5 LINES (18-22)

ROW 6 LINES (23-27)

ROW 7 LINES (27-35)

ROW 8 LINES (35-41)

ROW 9 LINES (41-48)

ROW 10 LINES (48-53)

ROW 11 LINES (54-57)

ROW 12 LINES (58-60)

ROW 13 LINES (60-61)

PROGRAM:READ1 10 REGISTERS PROGRAM USES 6 ROWS
ROW 1 LINES (1-4)

ROW 2 LINES (5-13)

ROW 3 LINES (14-18)

ROW 4 LINES (18-26)

ROW 5 LINES (27-34)

ROW 6 LINES (34-35)

PROGRAM:ACCHAR 4 REGISTERS PROGRAM USES 2 ROWS
ROW 1 LINES (1-3)

ROW 2 LINES (3-7)

PROGRAM:FLIP 5 REGISTERS PROGRAM USES 3 ROWS
ROW 1 LINES (1-4)

ROW 2 LINES (5-9)

ROW 3 LINES (9-11)

PROGRAM:MSG2 28 REGISTERS PROGRAM USES 15 ROWS

ROW 1 LINES (1-3)

ROW 2 LINES (3-4)

ROW 3 LINES (4-5)

ROW 4 LINES (6-7)

ROW 5 LINES (7-16)

ROW 6 LINES (16-16)

ROW 7 LINES (16-18)

ROW 8 LINES (18-20)

ROW 9 LINES (20-27)

ROW 10 LINES (27-29)

ROW 11 LINES (29-29)

ROW 12 LINES (29-33)

ROW 13 LINES (33-37)

ROW 14 LINES (37-39)

ROW 15 LINES (39-41)

CHAPTER 2

PROGRAM:TTONE 26 REGISTERS PROGRAM USES 14 ROWS

ROW 1 LINES (1-4)

ROW 2 LINES (4-7)

ROW 3 LINES (8-12)

ROW 4 LINES (12-14)

ROW 5 LINES (15-17)

ROW 6 LINES (18-18)

ROW 7 LINES (18-22)

ROW 8 LINES (22-23)

ROW 9 LINES (24-26)

ROW 10 LINES (27-33)

ROW 11 LINES (33-35)

ROW 12 LINES (35-37)

ROW 13 LINES (37-38)

ROW 14 LINES (38-42)

Appendix B

Sources for Non-Standard Items

This appendix lists several excellent sources of hardware, information, and services for and relating to HP handheld computers.

Hardware

Polaroid Sonic Ranging Module.....$60
SC-01 Speech Synthesis Chip.....$22
available from:

The Micromint, Inc.
25 Terrace Dr.
Vernon, CT 06066
(800) 635-3355

Teltone M-980 Call Progress Tone Detector
Teltone M-957 DTMF Receiver
available from:

High Technology Semiconductors
Tustin, CA
(714) 544-4871
(408) 942-0600

IL Converter Parts
HP-71 Torx wrench available from:

EduCALC Mail Store
27953 Cabot Road
Laguna, Niguel, CA 92677
(714) 582-2637

Otrona Keyboard available from:
Advanced Computer Products
P.O. Box 17329
Irvine, CA 92713

1310B E. Edinger
Santa Ana, CA 92705
(800) 854-8230

Custom Conversions, (41+71), and any HP-specific modifications:
S.O.S.
1850 E. 17th St. Suite #102
Santa Ana, CA 92701

Books

The following books are available from:

EduCALC Mail Store
27953 Cabot Road
Laguna, Niguel, CA 92677
(714) 582-2637

The HP-IL System

THE HP-IL SYSTEM: An introductory Guide to the Hewlett-Packard Interface Loop by Gerry Kane, Steve Harper, David Ushijima; Osborne/McGraw-Hill, 1982

71 IDS Volumes

Highly technical, complete documentation of the 71B hardware, bus, and operating system. The Internal Design Specification comes in five volumes:

#71-900068	Vol. 1, Detailed Description	$ 50.00
#71-900069	Vol. 2, Entry Points	50.00
#71-900070	Vol. 3, Source Code	200.00
#71-900071	Vol. 4, Hardware Specification	200.00
#82401-90023	Vol. 5, HP-IL Module Source Code	50.00

Appendix C

Dissertation as to Why Positive Handshake Logic is Not Worth Pursuing

HP says their GPIO and IL Converter interfaces can be programmed to have positive handshake logic, while I say it's so difficult to implement that it's hardly worth the trouble. (Some people might also feel this way about Chapter 10.) What follows is a detailed discussion as to what is involved when implementing positive handshake and why I avoid it.

The whole problem stems from the fact that 1) positive handshake logic must be initiated via DDL commands sometime after power is applied to the GPIO, and 2) if the handshake lines are held in the improper states, (if the outside world looks as if it's not ready), the 32-register buffer fills and locks up all communication on the loop as well as the controlling computer's program. This same "ready" condition must also exist in order for DDL commands to execute properly, which means your external circuitry must use negative handshake logic before the switch and positive handshake afterwards.

For example, here's a step-by-step account of what happens when positive handshake is requested:

When you first power up, the GPIO is in its default negative handshake mode (which means 0v is interpreted as "1", and 5v is interpreted as "0"; just the opposite of its data lines), and the two incoming handshake lines (RDYI and DACI) must be grounded in order to establish data transfers or initiate any Device Dependent instructions.

When the positive handshake logic is set via the DDL 0

command (see Chapter 1), it instantly takes effect and the GPIO then hangs and waits for the two incoming handshake lines to go to 5v (the new definition of "I'm ready") before proceeding.

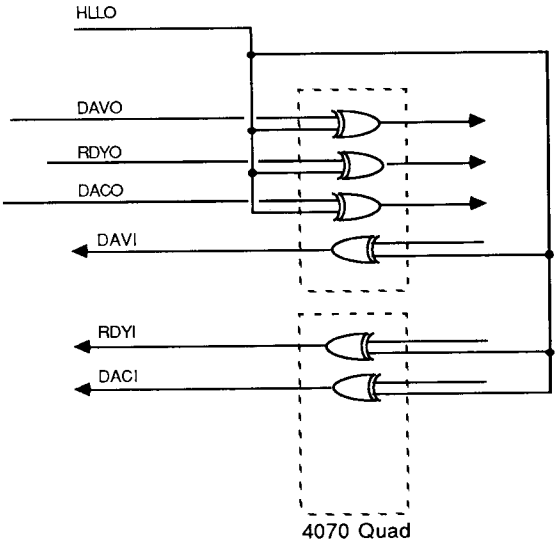
If this outside event doesn't occur, the DDL command is never completed, and everything (including the commanding computer) freezes up.

This requires some sort of external circuit which knows precisely when the positive logic bit is being set so that it can change the status of the handshake lines the instant it's required. How does the external circuit know precisely when to do this? Because the original IL Converter thoughtfully possessed the HLLO line, which instantly switches states to alert the circuit of precisely this condition!

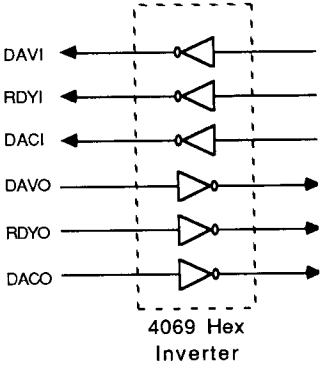
However, the 82165A HP-IL/GPIO interface doesn't even provide the HLLO line. (This deficiency is further aggravated by the unit's lack of PWRDN and WKUP lines, large size, and its insistence on being powered from an AC outlet.) According to HP, the only way to get positive handshake on it is by human interference: First, run the program. Then, connect the external device to the interface when it locks up. Again, very inconvenient and no benefits are obtained.

The only way I can possibly imagine to implement HP's suggested solution is shown in the figure top right. Six XOR (Exclusive OR) gates in series with all handshake lines, are all controlled by the HLLO signal. When the HLLO line goes low (signalling that positive handshake logic has been specified by the computer), it goes into all of the XOR gates and inverts their otherwise negative output. Not a difficult solution, but certainly less efficient than the bottom figure, which uses only one chip, less wiring, and doesn't even need a complex DDL 0 command. (This is substantial when the 41 is the controller.)

I use a simple solution for both interfaces as shown in the lower figure: just attach one extra component, a Hex Inverter, to all six handshake lines, and stick to the default negative logic. This keeps all configuration automatic and retains versatility, and in many cases eliminates the need for an IL Converter configuration routine.



One Way to Achieve Positive Handshake Logic Using the HLLO Line.

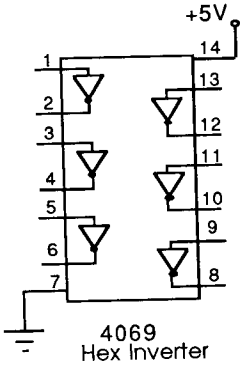


A Simpler Way Which Works on All 8-Bit Ports.

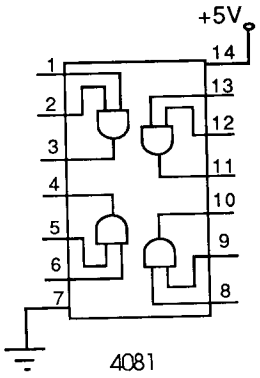
I believe that the positive handshake option exists because Hewlett Packard designed the interface to be truly versatile, and for that they should be applauded. However, using the hex inverter is a much better method to achieve this because it simplifies the software and removes the need for human intervention.

Appendix D

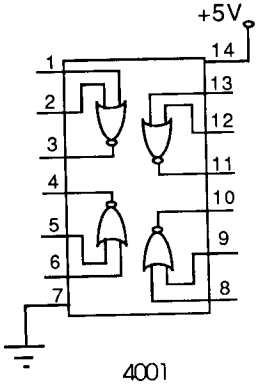
PINOUTS OF COMMON ICS



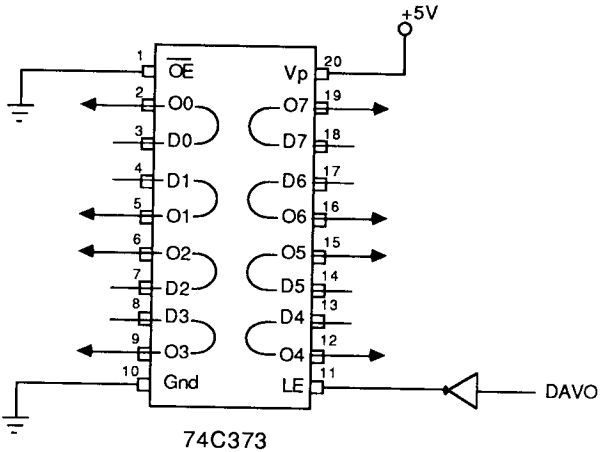
4069
Hex Inverter



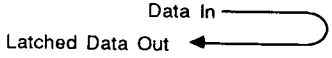
4081
Quad AND Gate

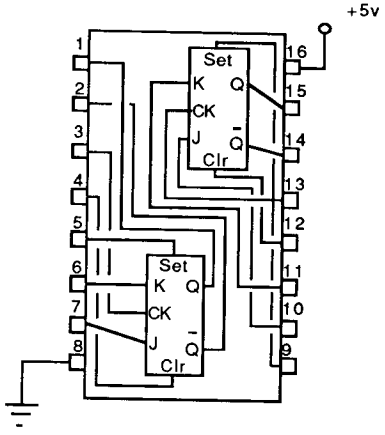


4001
Quad NOR Gate

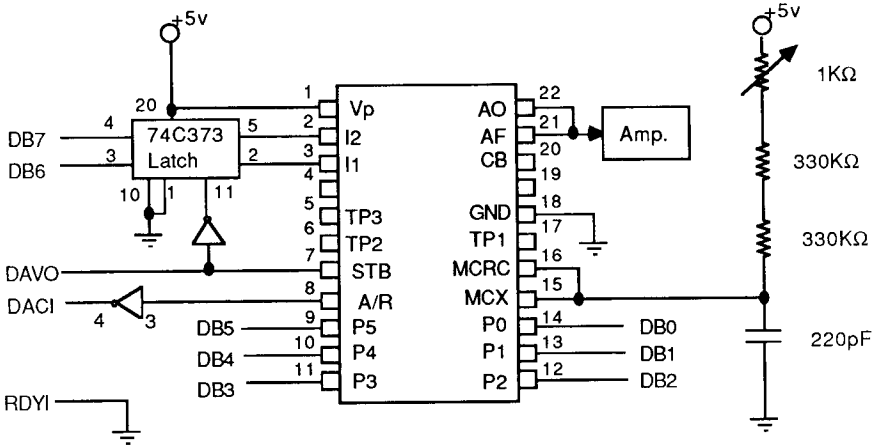


74C373
Octal Latch

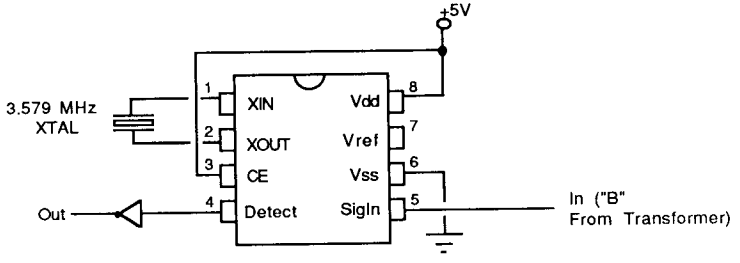




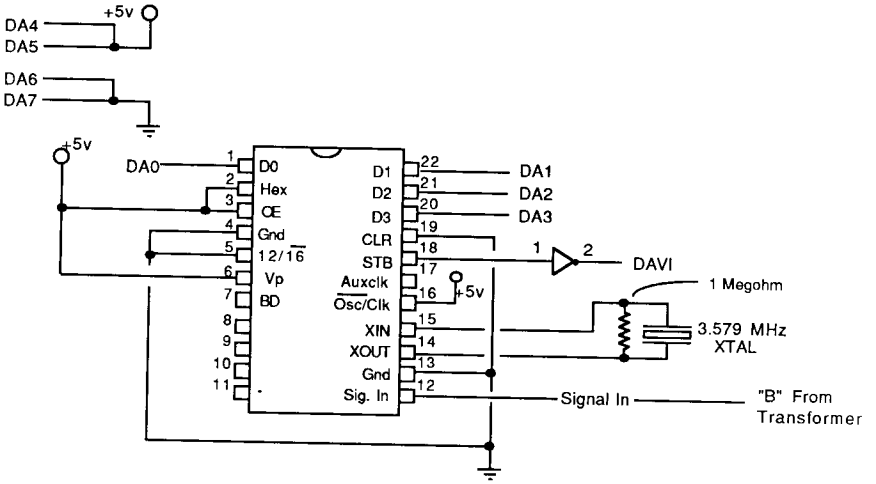
4027 Dual
J-K Flip Flop



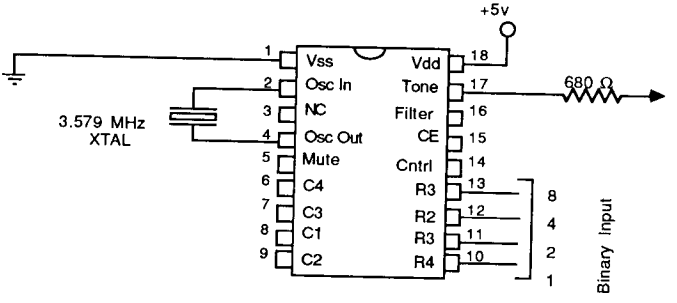
SC-01 Speech Synthesizer



M-980
Call Progress Tone Detector



M-957 Touch Tone® Decoder IC



MM5395 Touch Tone® Generator

Appendix E

GLOSSARY

A/D Converter

Analog to Digital Converter. One useful tool that takes real world information (usually measured via an analog sensor, such as a microphone for sound, a photo cell for light, etc.) and converts it to a digital form, which is something the computer can work with.

Accessory ID

One of two ways an HP-IL peripheral can identify itself. Accessory IDs are comprised of a number (rather than a name) that classifies it as a type of device (i.e. printer, video interface, etc.). This makes it easy for a computer when you tell it to print something; it just look for the first printer and sends the information there, without caring whether its a thermal printer or a ThinkJet variety.

ASCII

American Standard Code for Information Exchange. A recognized standard for representing alphanumeric characters by 1's and 0's.

Bit

A single signal that can only have two states: "1" or "0"; (sometimes referred to as "on" or "off".) Most computers are loosely based on this concept.

Byte

1) A "computer" magazine that isn't even aware of HP's existence. 2) A collection of 8 bits, normally assembled to

represent an ASCII character or a number. BYTE used to refer to the computer's internal word size back in the days when micros had an 8-bit architecture.

CMOS

Complimentary Metal Oxide Semiconductor. Fancy name for an integrated circuit fabrication technique whose chief attribute is very low power consumption. Rival fabrication techniques include TTL (Transistor- Transistor Logic), and NMOS.

CR/LF

Carriage Return/Line Feed. These are two ASCII characters automatically sent after each line of transmitted data, and are analogous to the return bars on the old manual typewriters. Well designed computers allow you to either disable this automatic sending feature or replace the CR/LF with special characters of your own choosing.

DCE

Data Communications Equipment. The fancy name used to describe the wiring configuration of an RS-232 device. DCEs can only communicate with a device wired as a DTE.

DDL

Device Dependent Listen. An HP-IL command that instructs peripherals that the following data should be interpreted as configuration instructions rather than data. An HP41 must have either an Extended I/O ROM or an IL Development ROM to generate this important command.

DDT

Device Dependent Talk. An HP-IL command that instructs peripherals to send some information about its status to the listener, rather than sending what it usually does when configured as a talker. The HP41 requires either the Extended I/O ROM or the IL Development ROM in order to generate this important command.

DTE
Data Terminal Equipment. The fancy name used to describe the wiring configuration of an RS-232 device. DTE devices can only talk to a device wired as a "DCE".

EDTEXT
The name of the 71's line editor. This program is available in either the FORTH/Assembler ROM or the Text Editor ROM.

Escape Sequence
A cryptic string of characters, preceded by the "escape" character (decimal 27), which is often used to tell a peripheral how to configure itself. Printers often employ escape sequences to let the host computer set their boldface, double wide, or italics mode. There is no real standard for escape sequences; it seems every peripheral manufacturer arbitrarily defines some random string when designing their peripherals, therefore most printers are not really compatible with each other.

Extended I/O ROM
One of two 41 plug-in ROMs that allows finer control of HP-IL. Since it uses the ALPHA register as a transmit buffer and generally provides higher-level commands than its counterpart (the IL Development ROM), it is considered the friendlier of the two.

frames
Another word for "HP-IL message".

GPIO
General purpose input/output, a term specifically referring to the HP82165A 8-bit port. Often, to save my breath, this term will be used as a generic term to describe ANY of the three 8-bit ports as discussed in Chapter 1.

HP-IL
Hewlett Packard Interface Loop. An interface scheme where all desired peripherals are strung together with a

2-conductor "thread", allowing selective communication with any or all peripherals on the loop and instant error checking. Both the 41 and 71 employ this method of I/O.

I/O

Input/Output. (Most people erroneously associate it with one of Jupiter's moons.) Catch-all phrase for the way the computer communicates with the outside world. The GPIO is one form of I/O; so are the keyboard and display.

IDY

Identify. The equivalent of idle small talk when nothing's happening on the loop.

IEEE

Institute of Electrical and Electronics Engineers.

IL Development ROM

One of 2 41 plug-in ROMs that allows finer control of HP-IL. Amongst its offerings is absolute low-level control of IL messages, a SCOPE mode for monitoring messages traveling through the loop, and a large general purpose data buffer for sophisticated I/O. Much more powerful than its counterpart (the Extended I/O ROM), and therefore more difficult to use.

IL Module

An optional peripheral that allows the 41 and 71 to access the Hewlett-Packard Interface Loop. Both the 41 and 71 use different attachments to add this capability; many other portable computers come with them built in.

KEYBOARD IS

An extra feature of the FORTH/Assembler ROM that allows a larger, external keyboard to replace the 71's own Smurf-sized keyboard. This LEXFILE is also available separately from HP.

LED

Light Emitting Diode. Functions like a normal diode except it gives off light when forward-biased.

LEXFILE
Short for Language Extension File. LEX files are assembly language routines that extend the 71's BASIC language environment, and are capable of adding new keywords, responding to system polls, rearranging the keyboard layout, and translating error messages to another language.

Opto-Isolator
Short for Optical Isolator. A method of switching a high voltage item from a low-voltage signal without exposing the signal to high voltage hazards (analogous to a relay's function). Opto-Isolators work by having the small signal drive an LED, and having the heavy load driven by a photosensitive transistor. Two common types are available: 3010, with a triac driver output, and the T11, with a photo-darlington pair.

phoneme
The basic sound components that comprise the English language. These are not quite the same as vowel and consonant sounds; for example the sound "I" can be broken down into two phonemes: "Aaa""Eee" (Aaa as in father, Eee as in Ellipsoid).

PWM
Pulse Width Modulation. An unobvious way to dim an AC lamp by turning it on and off very quickly rather than reducing the lamp's peak voltage.

reorder
One of the possible sounds that can be heard after dialing a telephone number. A reorder sounds exactly like a busy signal except it is twice as fast; and is used to signify that "all the circuits are busy now; please try your call again later."

ROM
Technically an acronym for Read Only Memory, originally meant to distinguish it from RAM, meaning Random Access Memory. (The label is somewhat misleading; as the information in both types can be randomly accessed.) ROM

has its information programmed in at the factory, and, unlike most RAM, retains the information forever. This makes it an ideal distribution medium for application software, which is commonly sold as a "Plug-in ROM".

Service Request

When a device on the loop wants the controller's attention, (such as when someone presses the PRINT key on the IL thermal printer when it's attached to the 41), the device must set a bit on the passing frame to alert the controller that someone needs attention. Usually, the passing data bytes are a valid transport mechanism, but when there is no routine traffic, the controller can constantly send IDY (Identify) commands during otherwise idle times to allow peripherals to make their needs known.

Synthetic Instructions

HP-41 instructions that cannot be entered into memory by normal means but execute flawlessly once there. This powerful technique has been the subject of many books (refer to Chapter 1), including one that the publisher is willing to endorse.

Touch Tone (®)

A trade name referring to an efficient tone-signaling method of telephone dialing. Rather than utilizing pulse trains as in the olden days, the digits are specified using two simultaneous, non-harmonic sine waves that sound to many people like musical notes in the earpiece.

ZENROM

A plug-in application ROM developed in Great Britain to allow direct-entry synthetics and an MCODE programming environment for the HP-41.

AFTERWORD

"Writing a book to increase the world's knowledge is like taking an eyedropper to the Pacific Ocean and saying, 'Here.'"
-- G. Friedman

Well, I finally got *that* book off my chest. Most people will view the contents as an illumination into the world of computers. However, it should be viewed from a little larger perspective.

Every time we notice a quasar's double image separated by a couple of arc minutes, we can infer another gravitational lens due to the mass of about a million previously unnoticed galaxies. A million galaxies here and a million galaxies there; pretty soon it adds up to real mass. According to Friedman [1], the universe is either open or closed, depending on its total mass. As we discover more mass lurking out there in the dark corners, we can increasingly look forward to a collapsing closed universe, instead of a perpetually expanding (open) universe. Thus, our eventual end will come in a "gigantic crunch" in about 10^11 years rather than the thermodynamic "heat death" that would have taken over 10^200 years. This is serious business; it's not every day that we discover our future is cut back by a factor of 10^189!

First of all, we are now about 10% of the way from the big bang to the big crunch. This, therefore, is probably the last book -- and the last afterword -- to be published during the first tenth of the life of this universe.

Next -- and much more important -- certain computations which were possible in an open universe are impossible in a closed one. For example, consider the insightful paragraph within the

rectangle at the bottom of this page. This message has about 100 characters, or 600 bits (assuming 6 bits per byte, which is all that is needed in this case). In order to construct this message by random search, 2^600 (which is roughly 10^180) distinct states must be tested. In the open universe of 10^200 years, we would have plenty of time; in the closed universe of only 10^11 years, we would not have nearly enough time, even if we converted all the universe's 10^90 atoms into computers running at 10^30 Hertz.

Thus, extremely convergent processes -- rather than random ones -- must be at work. Friedman [2] has shown that evolution and constraint theory have sufficiently powerful convergence over purely random processes to produce observed results in times compatible with closed universes.

Finally, just prior to the big crunch, when the Encino sky will be as uniformly bright as the sun, the proximity of the 10^90 atoms will permit some truly massive parallel computing at a decent cycle time. "Picture that!" Friedman [3] said.

As we rush to conclude, a note regarding motivations: Friedman [4] favors financial gain. On the other hand, Friedman [5] favors love. The truth, however, is the insight that Friedman [6] brings:

> The universe is all 1's and 0's.
> All else is illusion. The bits
> are out there to be crunched --
> Go get 'em!!

References

[1] Alexander Friedman, a cosmologist with the same name as my grandfather (but a little older) who developed the first models of the expanding universe.
Barrow, "The Anthropic Cosmological Principle", Oxford 1986

[2] George Friedman, my father, who erroneously thinks that he has influenced me through either heredity or environment. MS Thesis UCLA 1956; PhD Dissertation UCLA 1967.

[3] Gary Friedman, photographer for the Los Angeles Times, and 2nd best photographer bearing that name.

[4] Milton Friedman, Nobel Laureate in Economics, whose hypothesis is that money is important in the affairs of mankind.

[5] David Friedman, the president of Adult Films Association -- the same name as my brother (but a little older) -- whose hypothesis is that sex is important in the affairs of mankind.

[6] Gary Friedman, me.
This book, this page, this line, this this.

INDEX

ORDER BLANK

	Price per copy	Qty	Amount
For HP-71'S **HP-71 Basic Made Easy**, by Joseph Horn	$18.95	____	_____
For HP-71'S & HP-41'S **Control the World with HP-IL**, by Gary Friedman	$24.95	____	_____
For HP-41'S **HP-41 Advanced Programming Tips**, by A. McCornack & K. Jarett	$20.95	____	_____
HP-41 M-Code for Beginners, by Ken Emery	$24.95	____	_____
Inside the HP-41, by Jean-Daniel Dodin	$12.95	____	_____
Extend Your HP-41, by W. Mier-Jędrzejowicz	$29.95	____	_____
HP-41 Extended Functions Made Easy, by Keith Jarett	$16.95	____	_____
HP-41 Synthetic Programming Made Easy, by Keith Jarett (Includes one Quick Reference Card)	$16.95	____	_____
Quick Reference Card for Synthetic Programming	$2.00	____	_____
Synthetic Quick Reference Guide (SQRG)	$5.95	____	_____
For HP-10C, 11C, 15C, AND 16C **ENTER (Reverse Polish Notation Made Easy)**, by J.Dodin	$4.95	____	_____
Humor **It's Amazing How These Things Can Simplify Your Life: The Harold Guide to Computer Literacy**	$4.95	____	_____
ROM's **Barcode Generating ROM** by Ken Emery	$199.95	____	_____
AECROM by Redshift Software	$ 99.00	____	_____
Sales tax (California orders only, 6 or 7%)			_____

Shipping	1st book	Add'l books
within USA, book rate (4th class)	$1.50	$0.50
USA 48 states, United Parcel Service	$2.50	$1.00
USA, Canada, air mail	$3.00	$1.50
elsewhere, book rate (6 to 8 week wait)	$2.00	$1.00

elsewhere, air mail $12.05 for **Extend Your HP-41**, $6.05 for others

Free shipping for **ENTER** and **It's Amazing...** with purchase of any other book
Free shipping for QRC plastic cards or SQRG (any number)
Free shipping for ROM's

Enter shipping total here $_____

Total due $_____

Checks must be in U.S. funds, and payable through a U.S. bank.

Name_____
Address _____
_____
City_____ State_____ Zipcode_____
Country_____

Mail to:
SYNTHETIX, P.O.Box 1080, Berkeley, CA 94701-1080, USA Phone (415) 339-0601

CONTROL THE WORLD WITH HP-IL

Want to try something different than the usual programming applications with your HP-41 or HP-71? "Control the World with HP-IL" will show you how to use HP-IL to build and control such diverse items as an intelligent telephone autodialer/answering machine (complete with a speech synthesizer!), an automated photographic darkroom controller, an ultrasonic distance measurement unit, a slide projector dissolve controller for two projectors, and more. Imagine being able to accomplish all this with your battery powered, portable calculator/computer!

All the applications are explained in clear, crisp terms, with an easy-to-read informal style. Photos, illustrations, and circuit diagrams are used throughout the book to make all project instructions easy to follow. The general purpose building blocks and the concepts behind them are so clever and creative that you'll find dozens of your own uses for them.

Complete program listings and barcode included for all necessary programs.

ISBN 0-9612174-9-9