# H8 System Support II

March, 2023 by TAS v0.2.0

*A note about why this board exists:*

*The System Support II board has three reasons for existence. First, I wanted a single board that would add some basic functionality to a barebones legacy system. The serial port was designed as a console port to bring up a legacy system with no H8-4. The USB and RTC functions were designed to bring some of the most desirable features of Norberto's replacement boards to a system with only legacy hardware. The USB capability is likely to be much easier to use than some other methods of getting data onto a startup legacy system. Those three functions work with existing HDOS and CP/M software in the usual way and should require little explanation.*

*The second purpose of this board was to bring together things I have used as a "digital playground" onto one board. Features like I2C, PWM, and the cascade interrupt controller are experimenter's functions. The hardware is robust and stable, but what that hardware does in your own system is entirely up to you. I have provided code snippets from my own experiments that might provide you with some ideas about how to do your own explorations. But there is no polished and well-documented library of programs that you can use with the experimental functions of this board. As you look at the code I have provided, you'll see little structure, sharp inconsistencies in methods, and some things that simply don't work. I make no representation otherwise. This board is your sandbox, and you are the architect of whatever you choose to build with it.*

*The third purpose of the board was to add a couple of "fun" functions to any H8 – legacy or new technology – without taking up additional card slots on the bus. The Blinkenlights controller and Digitalker/EMIC2 voice box fall into that category. Blinkenlights has a delightfully small software footprint as it monitors the H8 bus directly with no requirement for CPU cycles. This document contains no new information about Blinkenlights, as it is already documented elsewhere. The principal advantage of Digitalker speech is that it is extremely simple to implement and carries a very low software burden. Its canned vocabulary is not extensive, but it's adequate for some interesting applications. A talking clock is fun and useful, and with I2C functions it's easy to have a talking thermometer or voltmeter. It's a great reminder of where voice technology was in the 1970s. The EMIC2 is a more recent device, dating back to 2012. It offers easy text-to-speech conversion, and provides an interesting contrast between its phoneme-based synthesis and the sampled voice of the Digitalker.*

# Contents

## General:

This board combines eight utility functions for the Heathkit H8 computer:

1) "Blinkenlights" activity indicator
2) Digitalker speech and/or Parallax EMIC2 Text-to-Speech
3) I2C controller (PCA9665)
4) PWM controller (PCA9685)
5) Real-time Clock (Epson RTC-72421
6) USB ports (FTDI VDIP1 or V2DIP1)
7) Serial port (16C550)
8) Cascade interrupt controller

All port addressing is assigned by 16V8 GALs. Each function of the board is individually enabled by jumper. Most sections of the board are independent of others so components can be omitted for "a la carte" construction. A "soft reset" function is provided to generate a local on-board RESET signal without having to reset the entire H8 computer.

## Port Assignments

These are the ports assigned in the sample GAL. These may conflict with other functions in your system. You should plan to modify the GAL as you need, using WinCUPL to edit the logic source.

| | |
|---|---|
| Digitalker data: | 100Q |
| USB data: | 331Q |
| Status/Control: | 332Q (for Digitalker and USB functions) |
| Soft reset: | 066Q |
| Serial UART: | 350Q-357Q |
| RTC: | 240Q-257Q |
| I2C: | 110Q-113Q |
| Interrupt Enable: | 104Q (IRQ 0-3), 106Q (IRQ 4-7) |
| Interrupt Mask: | 105Q |

## Soft Reset

Any I/O WRITE operation to the soft reset port (066Q by default) will cause a RESET signal for all (resettable) devices on the board except for the serial port UART. The VDIP in particular is prone to occasional glitching, and this can almost always be resolved by issuing the soft reset. This same soft reset port is used on the HA-8-3 color graphics video card to reset the TMS99x8 and AM9511 on that card. Soft resets are very handy to recover from untold misadventures without having to do a master reset of the entire H8 system.

"CLS" (Clear Screen, so named because the soft reset clears the TMS99x8 screen) is a tiny utility that will issue the soft reset from the HDOS command line.

## Cascade Interrupt Controller

This portion of the SSII is based on an interrupt handler design used by Heath in the HERO-I robot. Eight interrupt inputs (referred to as "IRQ" lines to distinguish them from the native H8 "INT" inputs) are broken into two groups of four shared interrupts. An interrupt input received on any of the four IRQs in a group will trigger a single output interrupt that can be routed to native H8 INT 3 thru 7. All IRQ interrupts are disabled at power-on. Each group of shared IRQs must be specifically enabled by software after power on, making this an ideal way to experiment with interrupts without having to deal with ROM upgrades. Each of the eight IRQ interrupts may be individually masked, providing further experimental flexibility.

To utilize the cascade interrupts, you must provide your own interrupt handler which would typically be located in a device driver. I have a device driver for the real time clock which is loaded in my PROLOGUE.SYS, and I have included code in that driver to handle the cascade IRQ controller. The steps to use the controller are:

Initialize the IRQ handlers

> I used a 24-byte area of shared low memory to store JMP instructions for eight individual IRQ handlers. I initialize all of these to RET op codes.

Register the main INT handler in the HDOS INT table

> The cascade interrupt controller can be jumpered to one or two native H8 INT lines. Install the address of your INT handler into the slot in the HDOS interrupt vector space (UIVEC) for however your jumpers are set. Sample contents of this INT handler are described below.

Reset all interrupts

> Write all zero bits to the IRQ mask register (105Q by default)

Enable the interrupt controller

> Write any value to the IRQ enable registers (104Q for IRQ4-7, 106Q for IRQ0-3). This "turns on" the interrupt controller, but no interrupts will be processed until the IRQ mask is set.

When you are ready to use any of the cascaded IRQs:

> Load a JMP to the address of your IRQ handler into the appropriate slot of the 24-byte IRQ handler area
>
> Unmask the IRQ by writing a "one" bit to the IRQ mask register (Dn = IRQn)

What the native H8 INT handler must contain:

> Upon entry to the INT handler, read the IRQ status register to determine which IRQs have been triggered. *More than one may have triggered at the same time, and you must handle all of them before exiting the INT handler!*
>
> Immediately reset and mask out any triggered IRQs. This is easy – read the IRQ status register (105Q by default) to determine which IRQs are requesting service, save that value, complement it and write the complement out to the IRQ mask register.

Immediately unmask any interrupts you want to continue to handle so that none are missed while we process the current cycle. The H8 interrupts are disabled when entering the ISR, so there's no worry about recursion in this cycle.

Service the active IRQs by CALLing the appropriate handler(s) you registered in the 24-byte IRQ handler area. Remember that you may need to service more than one shared IRQ in a given interrupt cycle.

When all IRQs have been processed, re-enable interrupts and RETurn from the native H8 interrupt.

To stress test the cascade controller, I connected three interrupting devices: The once/second RTC, the I2C interrupt line which can generate many hundreds of interrupts/second when I2C commands are active, and lastly the output from a variable frequency square wave generator. The H8 front panel display was used to confirm the once/second RTC interrupts – if they're working, the clock displayed on the panel will advance exactly once/second. A long-running servo movement routine was used to confirm the I2C interrupts were handled properly. And the stress was added by throwing a 10KHz square wave at the controller, generating 10,000 interrupts/second. With this extreme interrupt load, the front panel refresh was occasionally disrupted at 2MHz, but the I2C servo program ran normally and there was never a system crash or hang. At 16MHz, the front panel refresh was indistinguishable from the "no interrupt" load.

To keep track of interrupt activity, the INT handler was modified to keep 16-bit counts of interrupts handled and collisions (the number of times the INT handler was entered with more than one IRQ requesting service). Program "IRQACT" was written to dump the contents of those counters in real time. Even with this extreme stress test interrupt load, it is still unusual to find more than one IRQ request per interrupt cycle. I have not experienced a single system crash or hang that could be attributed to a failure of the cascade IRQ controller. Servo activity during testing was normal – missed I2C interrupts would be immediately noticed as a hang in the servo movement when the expected I2C response was not picked up.

Sample INT handler code for IRQ 4-7 jumpered to INT 5:

```
ISR:  CALL  $SAVALL    ;save all registers
      IN    IRQSTAT    ;get current IRQ request bits
      ANI   0F0H       ;only look at IRQ 4-7 in this handler
      STA   SAVSTAT    ;save this for processing
      CMA              ;complement IRQ status
      OUT   IRQSTAT    ;reset all active
      MVI   A,0F0H
      OUT   IRQSTAT    ;unmask IRQ4-7 now so we don't miss any
      LDA   SAVSTAT
      ANI   80H        ;IRQ7 requesting?
      JZ    IRQ6       ;rock skip
      CALL  USRIRQ+21  ;location of IRQ7 handler

IRQ6: LDA   SAVSTAT
      ANI   40H        ;IRQ6 requesting?
      JZ    IRQ5
      CALL  USRIRQ+18

IRQ5: LDA   SAVSTAT
      ANI   20H        ;IRQ5 requesting?
      JZ    IRQ4
      CALL  USRIRQ+15

IRQ4: LDA   SAVSTAT
      ANI   10H        ;IRQ4 requesting?
      JZ    IRQEX
      CALL  USRIRQ+12

IRQEX: POP PSW
      POP   B
      POP   D
      POP   H
      EI
      RET
```

## USB (VDIP1 or V2DIP1-48)

The SSII USB port can be used just like the USB port on Norberto's boards, with one addition. SSII implements the VDIP "Command/Data" mode toggle, which allows the VDIP to be used for direct serial communications with a peripheral device. You might use this to replace an RS232 serial connection or to communicate directly with a keyboard or mouse. The USB port resets to COMMAND mode, so it is fully compatible with Glenn's V-Utilities.

To enter DATA mode, set the high order bit in a write to the control register. When the VDIP module has successfully entered DATA mode, the high order bit in the status register will be set. Anything sent to the VDIP in this mode is not processed by the Vinculum command interpreter – it's simply pushed out over the USB data lines. You might need to initialize the remote FTDI device before using this command. The Vinculum documentation describes how you do this.

The sequence below initializes a remote FT232 device to 9600bps then enters DATA mode so that the VDIP can send serial data directly to a peripheral attached to the remote FT232:

Find the USB device ID of the attached FT232 device. This can be done programmatically. See the "USBTTY" program for some ideas about how to do this. Then use the Vinculum "SC" (Set Current) command to set the current device number to the ID you found.

```
FBD $384100      Sets the remote FT232 baud rate to 9600bps
FMC $0303        Sets the handshaking lines on the FT232
FFC $04          Sets the state of the FT232 flow control
```

Write to the control port (default 332Q) with the high order bit set. Vinculum documentation refers to this as DATAREQ.

Read the status port (default 332Q) and loop until the high order bit is set, acknowledging DATA mode. Vinculum documentation refers to this as DATAACK.

At this point, the VDIP is in DATA mode. You now have a direct duplex link to the remote serial device. Whatever you send to the VDIP is sent without modification to the serial device attached to the remote FT232, and vice-versal.

To terminate the DATA link, write to the control port with the high order bit cleared and wait until a read of the status port indicates the high order bit has cleared.

The entire sequence can be done programmatically. The "USBTTY" program can be used as an example.

## Serial Port (16C550 UART)

The SSII serial port is intended for use as a console port for those H8 installation that don't have access to an H8-4. For that use, you would set the interrupt jumper for INT3. The SSII port will work fine with either CP/M or HDOS as a console port.

Console terminals, like the H19, do not generally use hardware flow control. The SSII serial port implements one hardware handshake/flow control pair, brought out to the CTS/RTS signal pair. The H8-4 card implemented two handshake pairs. Serial protocols can vary, but most often the DTR/DSR pair is used like "presence detect" – is an attached device connected and powered on? - while the CTS/RTS pair is used for flow control. Devices like the H19 don't use hardware handshaking at all, but instead rely on X-ON and X-OFF in-band signaling. One handshake pair (or none!) is fine for H19 console terminal use.

There are jumpers for DTR/DSR handshake loopbacks. If you have software that doesn't work with this simplified serial port with just the CTS/RTS hardware handshake pair, you can often fool it by looping the program's own handshake. For example, a program designed for a modem might raise DTR and wait for the modem to handshake by raising DSR. If you have a program (or peripheral) that expects the DTR/DSR handshake, you may be able to fool it by looping DTR back to DSR – when the program raises DTR, it sees that signal looped back as DSR and life is good. Of course, this procedure doesn't work if the program expects the attached peripheral to initiate the handshake. I don't think there are many programs that can't get along with flow control using CTS/RTS and a looped DTR/DSR handshake.

The SSII board provides three physical connections to the serial port:

1) 15-pin DTE and DCE headers, like the H8-4 card uses (for the WH8-41 cable)
2) 10-pin (2x5) pin header, such as the IBM PC uses (for a standard PC motherboard serial cable)
3) 5-pin KK254 header, with send/receive plus one handshake (for a custom built cable)

A word about DCE and DTE labeling:

> DCE (Data Communications Equipment) is typically a modem
> DTE (Data Terminal Equipment) is typically a terminal or printer

DCE connects to DTE; DTE connects to DCE. These two have complementary signals and directions – receive goes to transmit, DTR goes to DSR, and RTS goes to CTS. A "straight thru" cable is used for this complementary connection; a "null modem" cable that crosses these pairs is used when connecting like devices. While there are variations on the theme, usually a port configured as DCE uses a female connector; a port configured as DTE uses a male connector.

There are (at least) two major sources of confusion when looking at serial connections:

1) Does a "DCE" label mean the port is wired as "DCE", or that it connects to "DCE"?
2) Is a computer considered "DCE" or "DTE"?

For the first question, in the H8 world the H8-4 "DCE" port is wired as DCE and is used to connect straight thru to DTE (and vice versa). That is the convention that is used on the SSII board as well: "DCE" does not mean "this port connects to DCE" – it means "this port is wired as DCE and connects to DTE".

The answer to the second question is "it depends". In the IBM PC world, serial ports are almost universally wired as "DTE", with a male connector designed for straight thru connection to a modem acting as "DCE". With

the H8-4 and WH8-41 cable, Heath recommends using the “DCE” header for connection to “DTE” (the H19 terminal, for example). But things quickly get confusing because you can plug a WH8-41 cable with a female DB25 into either the DCE or DTE header. The female connector on a WH8-41 doesn’t mean that the connection is “DCE” unless the cable is plugged into the “DCE” header.

While the idea of using a pre-built IBM PC cable with the 10-pin header for the serial port saves some labor, I have avoided it because I typically want to connect to a “DTE” device – a terminal, or a PC acting as a terminal. The most consistent way to make that connection is to use a female connector wired as DCE. The typical IBM PC cable is the opposite of that – a male connector wired as DTE.

## Real Time Clock (Epson RTC-72421)

When configured at ports 240Q-257Q, the RTC on the SSII board is identical in function to the RTC found on Norberto's newer boards. Glenn's V-Utilities know what to do with this RTC to timestamp USB files.

On the SSII board, you may also use the RTC's interrupt capabilities. Refer to the Epson datasheet for details on how to control the interrupts. The Epson chip is capable of generating interrupts at four different frequencies: 64 times/second, once/second, once/minute, and once/hour. The RTC interrupts can be directed either to a native H8 INT line, or to one of the IRQ lines on the cascade interrupt controller. If you want to use interrupts connected to a native H8 INT line, you will need a modified ROM that makes sure the RTC interrupts are off at boot time. Douglas Miller has included that initialization code in his ROM. The cascade interrupt controller disables its interrupts upon reset, so it is possible to use any ROM (like PAM37) with clock interrupts when connected to one of the IRQ inputs.

## "Blinkenlights" activity indicator

This function displays up to 8 activity LEDs on a single display board that may be mounted external to the H8 case. A GAL is programmed to monitor combinations of I/O port addresses and I/O control signals (READ/WRITE) directly from the H8 bus. No interconnections to other cards are required. A single 10-conductor ribbon cable connects the "Blinkenlights" controller to a display board with up to 8 LEDs.

A sample GAL configuration is provided, but this should be customized by each user to display the activities of interest.

## Digitalker speech processor

The National Semiconductor Digitalker chipset provides a very simple way of adding simple speech to the H8 computer. Digitalker is a fixed-vocabulary processor with 273 words in its combined ROM. The vocabulary is especially well-suited for industrial control projects. Digitalker makes it very easy to add simple speech to such applications as clocks, voltmeters, or other environmental monitoring.

Using Digitalker requires only outputting a single byte containing the number assigned to the desired word. To string words together, a "busy" status line can be monitored to determine when Digitalker is ready to speak the next word. Two vocabulary sets are included on the board – the base set (called SSR1/SSR2 in the National Semiconductor literature), and an extended set (SSR5/6). Originally, the 16K vocabulary sets were released on a pair of proprietary 8Kx8 ROMs, either SSR1/2 or SSR5/6. On this board, both vocabulary pairs (four 8K images) have been combined onto a single 27C256 or 28C256 32Kx8 ROM. One bit (D0) of a latched control port selects whether the SSR1/2 or SSR5/6 vocabulary set is used.

Either a UV-erasable ROM (27C256) or an electrically-eraseable EPROM (28C256) may be used. Set jumper JP3 to match the ROM type installed. The "Speaker" connector on the board should be connected to an 8-ohm speaker.

| The Digitalker speech ROMS contain copyrighted material. Please contact me if you need a source for these ROMs. The Digitalker speech chip itself is often available on eBay. |
|---|

The speech procedure is:

WRITE to control port: D0=0 for SSR1/2 or D0=1 for SSR5/6 vocabulary set
WRITE vocabulary index to Digitalker port
READ status port:
    D0=1: Digitalker is speaking
    D0=0: Digitalker is ready for next word

Note that the vocabulary set bit is latched, so it's only necessary to write a value to the control port when the desired vocabulary set is changed.

There are example BASIC programs in the MicroMint/MicroMouth manual.

Here is a sample BASIC program that will speak:

“EMERGENCY ALARM”, tone, “FUEL FLOW IS TOO HIGH”, tone, ‘EVACUATE, EVACUATE!”

```
10 CTRL=&O332
20 DTALK =&O100
30 READ A$
40 IF A$="0" THEN 150
50 IF LEFT$(A$,1)="A" THEN ROM=0
60 IF LEFT$(A$,1)="B" THEN ROM=1
70 S=VAL(MID$(A$,2))
80 PRINT ROM,S
90 GOSUB 110
100 GOTO 30
110 IF (INP(CTRL) AND 1)=1 THEN 110
120 OUT CTRL,ROM
130 OUT DTALK,S
140 RETURN
150 END
160 DATA "B35","B3","A65","A66"
170 DATA "A84","A83","A96","A2","A91"
180 DATA "A65","A66","B40","B40","A65","A66"
190 DATA "0"
```

## EMIC2/DECTalk Speech Generator

An alternative speech function for the H8 is provided by the EMIC2 Text-to-Speech board, with DECTalk (v5) capability. This is a more recent hardware feature, dating to 2012. The board is manufactured by Parallax, and you can find links to documentation and sample programs at http://www.parallax.com. EMIC2 had previously been carried by Adafruit and Sparkfun, but Parallax now appears to be the sole source for the board.

EMIC2 uses a TTL serial interface – with 0 and 5v signal levels, not the typical ~10v +/- RS232 signal levels. If you don't need the SS2's serial port for a console, you can use the 16C550 UART to provide the TTL serial connection to the EMIC2. A 6-pin female header is provided on the board for that purpose. Jumpers JP11 and JP12 select whether the TTL output from the UART should be routed to the EMIC2, or to an external serial device through the MAX232 RS-232 driver. The EMIC2 1/8" stereo jack output can share the audio amplifier and speaker circuit of the Digitalker through connector J36 ("Audio In"). You should not connect the EMIC2 speaker outputs (SP+ and SP-) to J36, as the EMIC2 SP- output is not at ground potential. The EMIC2 stereo jack carries a single-ended output with a blocking capacitor and is safe to connect to J36.

An alternative method of connecting to the EMIC2 is to use a FTDI USB cable. This cable converts the USB serial data to TTL serial levels, and with minor programming can be used with the VDIP1 module. The DATA mode toggle on the VDIP that is provided by the SS2 board is ideal for use with such a USB-to-serial connection. In DATA mode, a direct conversation can be established between the VDIP and EMIC2, requiring no special processing to send or receive characters. When DATA mode is active, what you send to the VDIP will be transparently passed to the EMIC2 and vice-versa. The VDIP connection would be ideal for a EMIC2 mounted in an external box with its own speaker.

In addition to text-to-speech capabilities, the EMIC2 board supports DECTalk. Many people know DECTalk as the voice behind the *Moonbase Alpha* game. DECTalk offers direct phoneme-level control of the EMIC2 board (similar to the Votrax SC-01), as well as the much easier to use ASCIII text-to-speech functionality. DECTalk supports an interesting - if not practical - "singing" mode, with each phoneme being generated at a specific pitch, from C2 through C5. The tone pitch will not be "MIDI-perfect", but with some effort you can assemble phonemes into a reasonable facsimile of a singing voice. There are many examples of DECTalk songs on the web but be aware that most of them were written for an earlier version of DECTalk (v4). There are a handful of phoneme coding differences between v4 and v5 versions – you will almost certainly have to do some manual patching of v4 source to get it to work without error on v5.

The DECTalk v5 source for "Daisy Bell" (think HAL-9000) is on the next page, and the song is included in the EMIC2's firmware. The EMIC2/DECTalk buffer is limited to ~1,000 bytes, so this song is broken up into a number of phrases, each ending with `"[:n0]."` Each phrase must be sent to the EMIC2 individually, waiting for the board to respond with its `':'` prompt indicating it is ready for the next phrase to be loaded. A simple program in the high-level language of your choice can handle the transmit and handshake task.

```
[:phone arpa speak on][:n0][dey<650,22>ziy<600,19> dey<650,15>
ziy<600,10> gih<200,12>v miy<200,14>yurr<200,15> ae<400,12>
nsax<200,15>r duw<750,10> ][:n0].
[:n0][ay<600,17>m hxae<500,22>f kr ey<650,19>ziy<600,15>
ao<200,12>ll fao<200,14>r dhax<200,15> llah<400,17>
v ao<200,19>v yu<750,17> ][:n0].
[:n0][ih<200,19>t wow<200,20>nt biy<200,19> ax<200,17>
stay<500,22>llih<200,19>sh mae<350,17>rih<400,15>jh<150,15>][:n0].
[:n0] [ay<200,17> kae<400,19>nt ax<200,15>fow<400,12>
rd ax<200,15> kae<350,12>rih<400,10>jh<150,10>][:n0].
[:n0][bah<200,10>t yu<500,15>d lluh<200,19>k swiy<400,17>
t ah<200,10>p ao<500,15>n dhax<200,19>siy<200,17>t ao<200,17>
v ah<200,19> bay<200,22>six<200,19>kel<200,15> bih<400,17>
llt fao<200,10>r tuw<800,15>][:n0].
```

## PCA9665 I2C Controller

The PCA9665 opens up the world of I2C devices to the H8. The chip may be operated in either a polled or interrupt-driven environment. The Cascade Interrupt Expander on the System Support II board can be used to simplify experimentation with and development of interrupt-driven I2C applications.

The Texas Instruments "I2C Primer" provides a good introduction to I2C protocols. Even with the PCA9665 doing the bitwise manipulations, I2C communication remains a software-intensive task. The HDOS "I2C9665.ACM" (polled mode) or "PCAINT.ACM" (interrupt mode) files contain a sample library of I2C routines (HDOS assembler) that can be used as a starting point for your own exploration. An I2C conversation typically requires you to specify the unique I2C address that has been assigned in hardware to the device, a register number within the device to be read or written, and a count of the bytes to be transferred. Most I2C messages are very short – one or two data bytes. I2C is a "master-slave" protocol. As implemented in these .ACMs, the H8 is the master and initiates all conversations (read or write) with attached slave devices.

Three High-level I2C functions are provided in both .ACM libraries. Before calling them, register pair <HL> should be loaded with the address of a byte string containing:

I2C device address
number of bytes to transfer, 'n'
register number to write or read
for WRITES only:
0-'n' bytes to be written

| | |
|---|---|
| I2CRNB | Read up to 5 bytes from an I2C device, and place the results in buffer I2RST |
| I2CWRT | Write 'n' bytes to an I2C device, doing byte at a time (conventional) transfers |
| I2BWRT | Write 'n' bytes to an I2C device, using buffered transfer mode (fastest) |

Examples of using these library functions:

```
        LXI     H,PWMINI0
        CALL    I2BWRT          Buffered write
        … …
PWMINI0 DB      80H, 2, 0, 12H, 34H

sends "12H", "34H" to device with I2C address "80H", register 0
```

```
        LXI     H,PWMVAL0
        CALL    I2CRNB          Read 0-5 bytes
        LDA     I2RST           <A> contains first byte returned
        … …
PWMVAL0 DB      80H, 2, 0H

reads two bytes from device "80H", register 0, puts them at [I2RST]
```

Every I2C device will have a unique set of registers and initialization requirements. You will need to refer to the device's datasheet for the particular steps to follow. A typical use sequence for an I2C sensor would be:

(Usually) Initialize the device by sending one or more bytes using I2CWRT or I2BWRT

(Sometimes) Start a measurement on the device by sending a command byte using I2CWRT

Read the result using I2CRNB

For devices returning 16-bit values, be aware of what is sometimes referred to as "gender": Is the high order byte returned first or last? Intel stores numbers as "low order, low address", but not all sensors follow that convention.

A debug flag is stored in low shared memory that can be set to force the I2C library routines to display the details of the current I2C conversation. This is a very good way to explore how I2C works – and to troubleshoot why my sloppy software may not be returning an expected value. Running "I2CDBUG" toggles the flag on and off. The HDOS programs I run from PROLOGUE.SYS clear this flag on boot. Without those programs, you will find this flag is randomly cleared or set when you start your system.

Try this: set the debug flag by running "I2CDBUG" (depending upon your startup conditions, you may need to run it twice to get the flag SET!). Then run the program "I2CPWMLD". This sample program exercises the PCA9685 PWM controller (it is further explained in the section describing the PWM controller).

You should see a display something like this:

```
>I2CPWMLD
 S[ *08 00 *18 06 *28 ]P *F8
 S[ *08 80 *18 00 *28 80 *28 ]P *F8
 S[ *08 80 *18 00 *28 10 *28 ]P *F8
 S[ *08 80 *18 FE *28 67 *28 ]P *F8
 S[ *08 80 *18 01 *28 14 *28 ]P *F8
 S[ *08 80 *18 00 *28 A0 *28 ]P *F8

[A-H]nnn: A50
A2000  06 S[ *08 80# 06# 00# 00# D0# 07# *28 *06 ]P *F8

[A-H]nnn: ^C
Exiting
>
```

The first group of lines are initialization bytes to the PCA9685 controller. Examine the second line:

```
S[ *08 80 *18 00 *28 80 *28 ]P *F8
```

S[ indicates the I2C controller was asked to send an I2C "START" sequence

*08 is the controller's response ("START ACKNOWLEDGED")

80 the I2C controller is asked to send address "80H" on the bus

*18 is the controller's response ("ADDRESS ACKNOWLEDGED")

00 the I2C controller is asked to send register "00H" to the slave device at address "80H"

*28 is the controller's response ("BYTE TRANSMIT ACKNOWLEDGED")

80 the I2C controller is asked to send data byte "80H" to the slave device

*28 controller's response ("BYTE TRANSMIT ACKNOWLEDGED")

]P the I2C controller is asked to send the I2C "STOP" sequence

*F8 controller's response ("STOP ACKNOWLEDGED")

*This entire sequence transmitted just one byte of information to the slave device* – a "80H" to be stored in register "00H". To do that, we had to initiate five requests to the controller, and monitor the controller after each request for an ACK response before continuing to the next phase of the sequence. This conversation resulted from code like this:

```
            LXI   H,CMDSTR
            CALL  I2CWRT
            … …
CMDSTR      DB    80H,01H,00H,80H
```

The PCA9665 is capable of “buffered transmit mode”, where a bunch of bytes are loaded into the chip’s buffer and with one transmit request they are all sent. This helps reduce the software load. Typing “A50” at the prompt tells the program to set PWM channel 0 (“A”) to ~50% duty cycle. The instructions to the PCA9685 PWM controller to do this are sent in buffered mode:

```
06 S[ *08 80# 06# 00# 00# D0# 07# *28 *06 ]P *F8
```

06 we tell the controller we want it to buffer 06H bytes

S[ START command, as before

*08 ACK after START, as before

80# 06# 00# 00# D0# 07#
this is a string of 6 bytes to be sent in buffered mode to the slave
80H is the slave address
06H is the register number
00H 00H D0H 07H are the data bytes to be placed in that register

*28 ACK is received, but this ACK is for the entire string sent with one controller command

*06 the controller reports the total number of bytes sent

]P STOP command, as before

*F8 ACK after STOP, as before

Reading data from an I2C device introduces another complexity – the "repeated START" sequence. This is the I2C conversation to read two bytes from the PCA9685, starting at register 0:

```
S[ *08 80 *18 00 *28 S[ *10 81 *40 20 *50 14 *58 ]P *F8
```

| | |
|---|---|
| S[ | START command |
| *08 | ACK, START sent |
| 80 | I2C address of the PCA9685 |
| *18 | ACK, address sent |
| 00 | Starting register number |
| *28 | ACK, byte sent |
| S[ | Repeated START – a second START sent before a STOP sequence |
| *10 | ACK, repeated START sent |
| 81 | I2C READ address of PCA9685 – has low order bit SET |
| *40 | ACK, READ address sent |
| 20 | first data byte returned |
| *50 | ACK sent to slave |
| 14 | second data byte received |
| *58 | NAK sent to slave to indicate last byte requested |
| ]P | STOP command sent |
| *F8 | ACK, stop sent |

The PCA9665 state engine is a good match for an interrupt service routine. The PCA9665 signals completion of a step with an interrupt. The ISR examines the PCA9665 status byte and using the state diagram can determine what should happen next. There is a state diagram posted on the board's web page that shows the sequence of states. The SSII board is an ideal playground for experimenting with the PCA9665 interrupt mode because it allows you to use interrupts that are fully software controllable (that don't depend upon hardware initialization in ROM code).

## Pulse-Width Modulation Controller (PCA9685)

The PCA9685 chip on this board is connected to the I2C bus of the PCA9665. The PCA9685 datasheet contains full details on initialization and use of the chip. The PWM controller could have three uses in the H8:

LED brightness and color control
Servo control
General analog output for slowly varying voltages

The easiest of these to demonstrate is LED control. Simply connect an LED between the "+5v" and Channel "0" pins of the PWM output header. Dropping resistors are built into the circuit board. The program "I2CPWMLD" is a simple demo of PWM output control. After loading, the program will prompt:

```
[A-H]nnn:
```

Enter a channel identifier (A = channel 0, H = channel 7) and a relative output value from 0 to 100 (%), like "`A50`". The PCA9685 has 1/4096 resolution – this demo program keeps the math simple by multiplying the input by 40 to get a range of 0-4000. You should find you can run the LED brightness from OFF (0) to full brightness (100). You'll probably not notice much difference in observable brightness between a value of 50% and 100% - most of the control will occur at the low end of the value range.

To control LED color, you would use an RGB LED (common anode), with common wired to +5v, and R, G and B wired to channels 0, 1 and 2. You would use a command like "`A10 B50 C25`" (space between each section) to adjust the relative brightness and overall color of the RGB LED.

Servo control is similar, but servos operate within a very narrow band of pulse width – and they can be damaged if a pulse width outside their normal operating range is applied for any length of time. I have a program written specifically for the Arduino Braccio 6-DOF robotic arm, but it can be used with many common servos. This is a fairly sophisticated program that accepts either keyboard inputs or joystick inputs via the ADC port on the System Support I card to compute positions of the six servo motors on the Braccio arm. The movements are handled by background tasks processed during the 2ms interrupt cycle. A reasonably complete (keyboard) command interpreter is implemented that supports scripting for complex automated movements. If you decide to explore servo control from the H8, you can request a copy of this (HDOS) program.

A simple RC filter on the PWM outputs will produce variable analog voltage outputs that may be good enough for some purposes that don't require rapidly changing voltages. You can experiment with different R-C constants to see what works for you. With a scope on the PWM output and the output of the R-C filter, you have a great way of demonstrating R-C circuit behavior.

PWM Dropping Resistors: The schematic and silkscreen specify 220 ohm series resistors for the PWM outputs. This is the *minimum* value you should use to protect the PCA9685 outputs. I usually run with a 1K resistor pack in place for extra protection, and I find that LED brightness is quite acceptable even with the higher value.