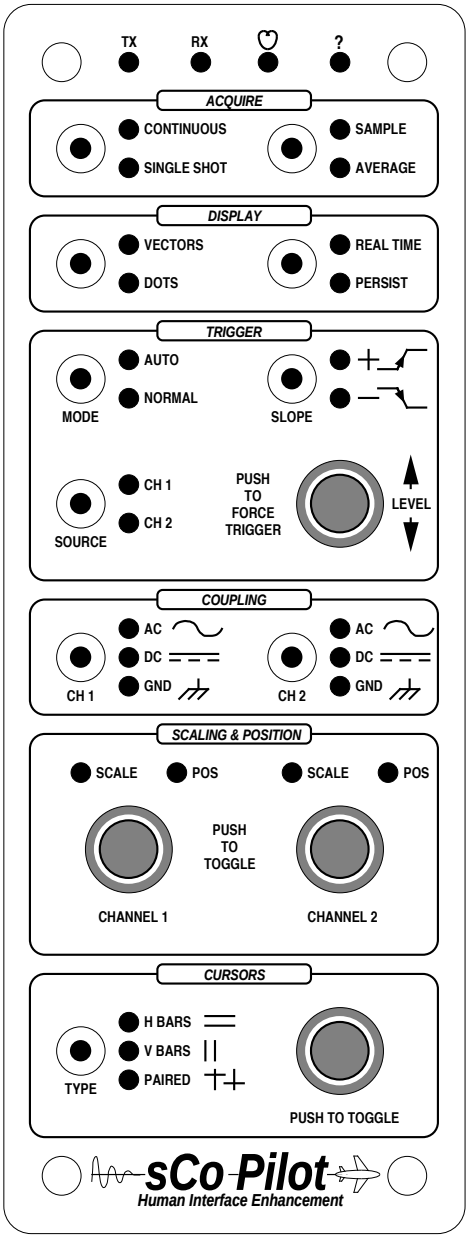
*The*

# sCo-Pilot

*(Co-pilot for scopes)*




***A 68HC908GP20 based project***

***submitted for consideration in the***

***Eleventh Circuit Cellar Design Contest***

***May 27, 1999***

sCo Pilot
Human Interface Enhancement

Table of contents

sCo-Pilot
Human Interface Enhancement

Project Abstract

The unprecedented functionality of modern digital oscilloscopes (as well as other test equipment) is unquestionable. Many of the modern digital scopes are eclipsing their analog counterparts in applications ranging from digital systems debug to RF design. However, the advantages of many of these advanced capabilities can be somewhat negated by cumbersome user interfaces. Sometimes there are not enough buttons (function keys) on the scope, or quantities intuitively associated with knobs require multiple keystrokes to adjust.

As the name implies, the sCo-Pilot is a device intended to function as a "copilot" for operating an oscilloscope. Targeted at the Tektronix THS series of handheld digital oscilloscopes, the sCo-Pilot functions as a "copilot" between the user and the scope. In this capacity, the sCo-Pilot provides the following features:

- Non-invasive human interface enhancement. All interfacing is accomplished via the scope's RS-232 port. No modification to the scope is necessary to use the sCo-Pilot.
- Convenient (non-invasive) attachment to the existing THS handle brackets (see photos).
- *Single button access* to commonly used features which are normally buried in hierarchical sub menus.
- *Detented optical encoders* (digital knobs) for adjusting quantities traditionally associated with analog knobs.
- All knobs recognize *acceleration* to enhance the "analog" feel.
- *At-a-glance LED indication* of current instrument status. This supplements the lack of screen indication on the THS series.

Although the sCo-Pilot project was designed specifically for use with the Tektronix THS series, it is important to emphasize that the sCo-Pilot concept could be applied to many other digitally controlled instruments.

The sCo-Pilot utilizes the RS-232 instruction set of the THS series to translate user actions into scope commands. On the surface, this task may appear to be fairly trivial. However, the command structure (syntax) of the THS series is not conducive to embedded control. It requires floating point capability and unique handling for almost every different command. Additionally, the command processor needs to be fast enough to perform rudimentary floating point operations while providing the user with instant operational feedback. These obstacles were surmounted through the use of the 68HC908GP20 (herein after referred to as the 'GP20). With its high operating speed (203ns instruction cycle @ 19.660MHz) and vast FLASH program space (20K bytes), the 'GP20 made implementation of the sCo-Pilot practical.

The sCo-Pilot system consists of two major components...namely the 'GP20 and an EEPROM based CPLD – the Vantis (AMD/Lattice) MACH211 64 macrocell CPLD. The 'GP20 handles all of the user interface and command processing while the CPLD functions strictly as an SPI-based I/O expander to drive 29 of the sCo-Pilot's 31 status LEDs (see block diagram). The firmware for the CPLD was written in ABEL and compiled using the Vantis Synario starter software. This component was selected based on the availability of sample parts and development tools (the starter kit was already on hand). Likewise, all 'GP20 software was written in assembly to minimize the cost of development tools – there was no budget (or time) for a C compiler in this project.

Most of the development effort went into the 'GP20 software. As mentioned earlier, interfacing an

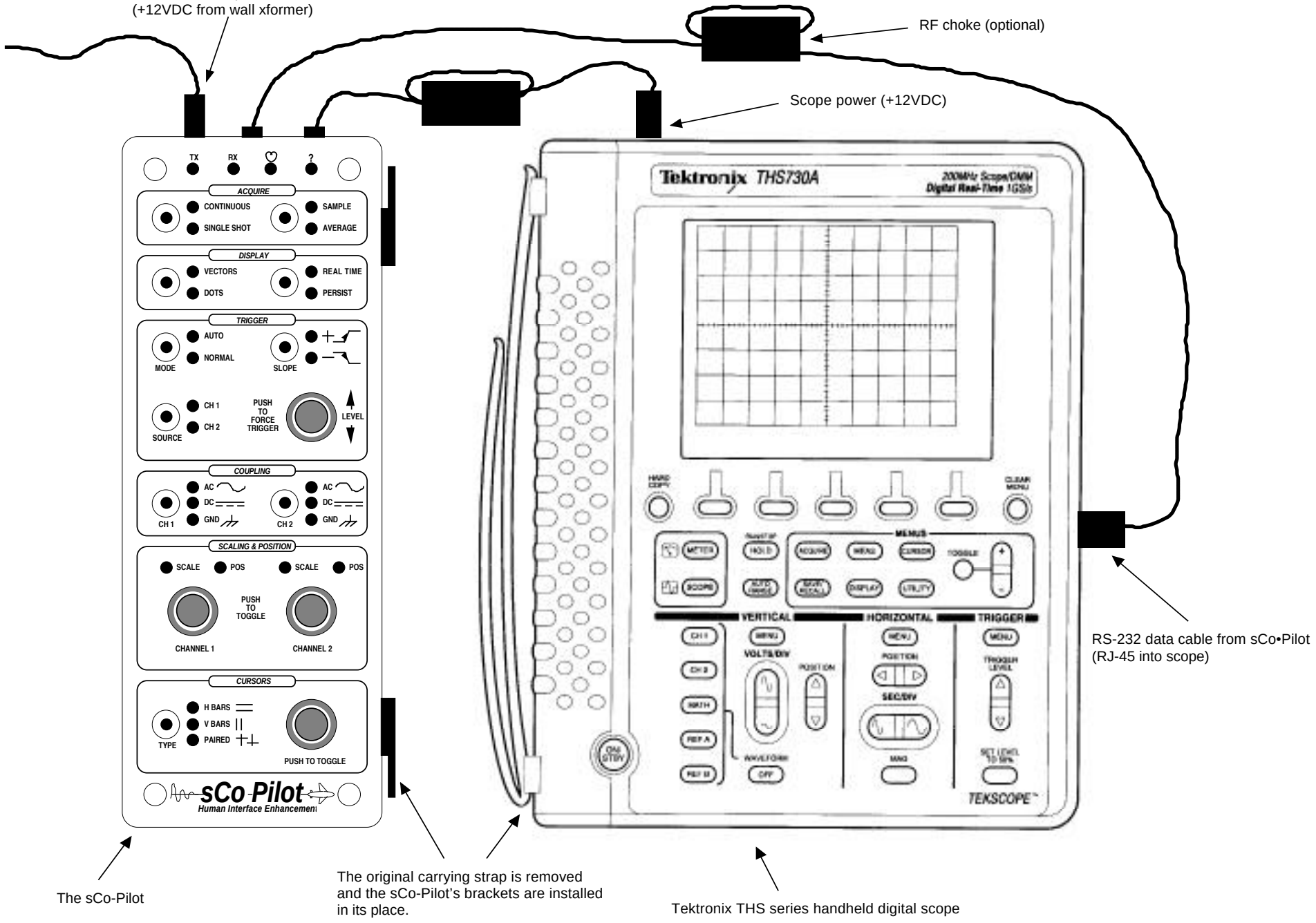
embedded CPU to the THS scope is not straight forward. To address this issue, the software was written in a "state machine" style in which the CPU continuously executes a main loop whose functions are state machines based on user commands. This style of code structure enables each different action (button press or knob turn) to have its own custom state machine which is independent of any others – thus eliminating many if-then comparisons and exceptions to rules.

When the user presses a button, a state machine handler for that button becomes part of the main loop. The execution of the state machine "locks-out" the execution of all others until the currently running state machine is finished. This prevents multiple state machines from running simultaneously and confusing the scope. However, multiple button presses are "queued" and executed when the previous button handler is complete.

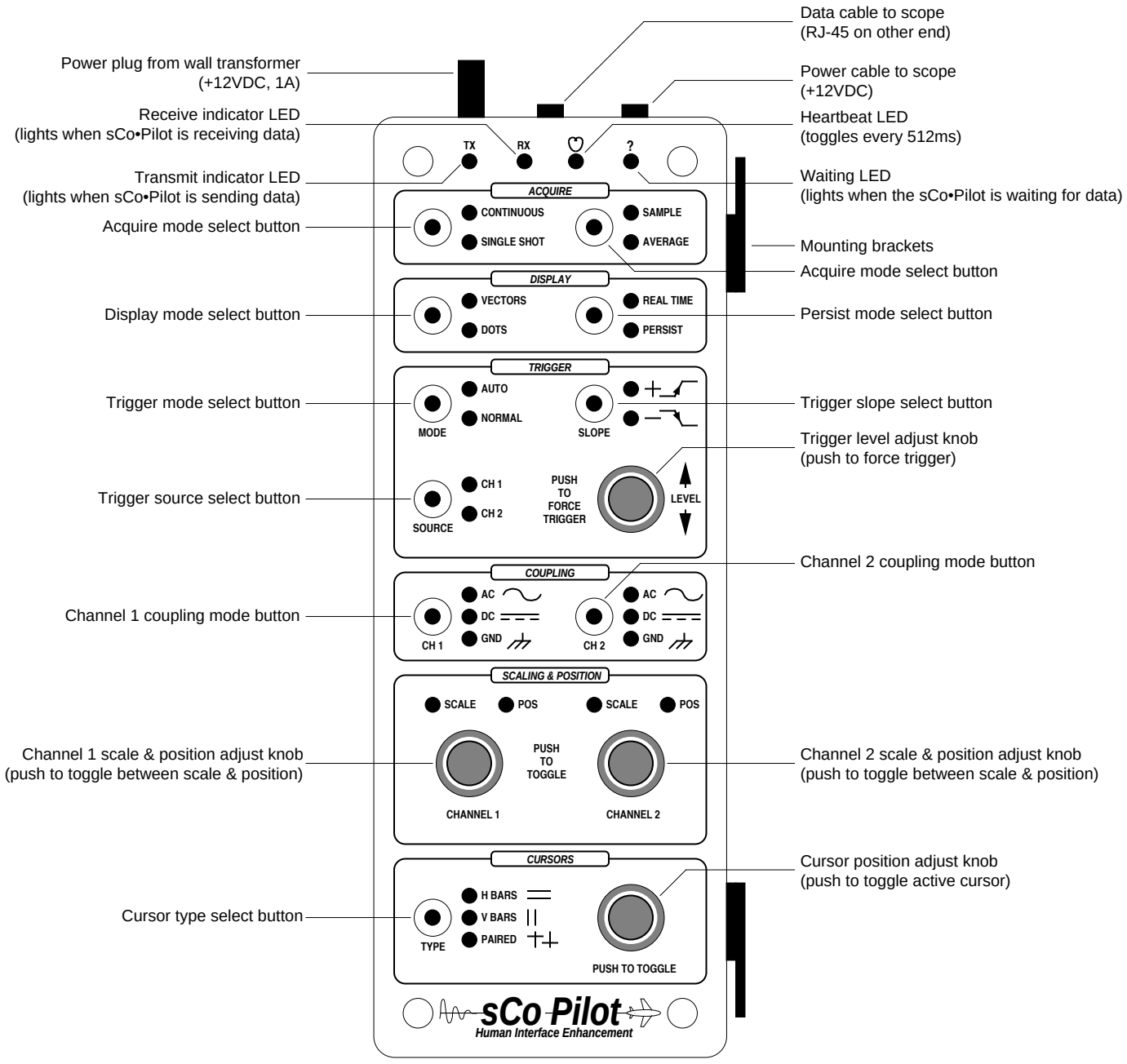
The main loop runs continuously with a nominal loop time of about 250$\mu$s. This high speed loop allows the incoming 9600BPS serial data to be polled (through the SCI) rather than interrupt driven. The only active interrupt is the timer interrupt – which is used to keep real time by providing a "tick" every millisecond. Other activities performed as part of the main loop include updating LEDs, debouncing keys, reading knobs, and counting idle time. If the sCo-Pilot is idle for a given amount of time, it automatically polls the scope for current status information. This effectively re-synchronizes the sCo-Pilot with the scope in case the user changed something on the scope's front panel.

Operationally, the sCo-Pilot project was successful in providing human interface enhancement to the THS scope. The sCo-Pilot has eliminated the frustration and confusion of using the scope's native "soft keys" to cycle through hierarchical menus. The single button access to commonly used features, knobs (with acceleration) for adjusting analog quantities, and LED status indication have dramatically increased user efficiency and usability of the THS scopes.

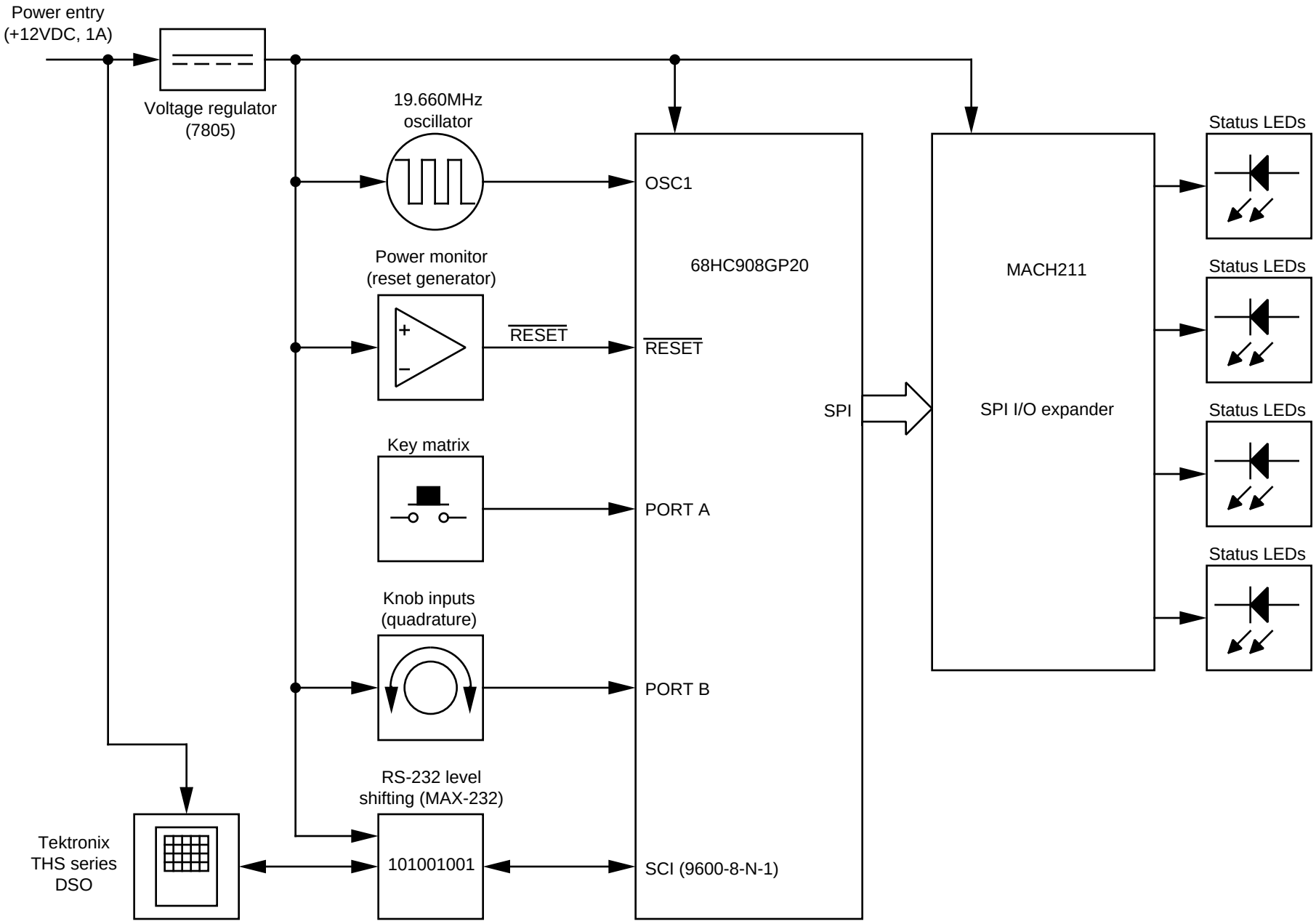
# sCo-Pilot setup and connection diagram



# *sCo-Pilot feature diagram*



# *sCo-Pilot circuit block diagram*



sCo-Pilot
Human Interface Enhancement

Hardware Overview

The sCo-Pilot system consists of two major components...namely the 'GP20 and an EEPROM based CPLD – the Vantis (AMD/Lattice) MACH211 64 macrocell CPLD. The 'GP20 handles all of the user interface and command processing while the CPLD functions strictly as an SPI-based I/O expander to drive 29 of the sCo-Pilot's 31 status LEDs (see block diagram & schematic).

Power for the sCo-Pilot is derived from the scope's wall transformer. The output of the wall transformer is a regulated +12VDC capable of supplying 1A or more. The +12VDC input is regulated by U4 to produce the +5V required by all of the sCo-Pilot's ICs. Capacitor C17 provides local bulk energy storage while C16 provides local high frequency bypass. Diode D1 prevents the output of U4 from exceeding the input by more than a diode drop. Under nominal operating conditions, the sCo-Pilot requires about 275mA. A parallel tap from the incoming +12V is provided as a sCo-Pilot output to power the scope.

At the heart of the sCo-Pilot system is the 68HC908GP20 FLASH based MCU (U3). All power pins on this device incorporate broadband power supply bypassing. The master clock for the 'GP20 is provided by oscillator Y1 at 19.660MHz. This master clock frequency produces an internal cycle time of about 203ns. Resistor R1 limits oscillator currents and reacts with the 'GP20's input capacitance to round-off the high frequency edges. The oscillator incorporates local broadband bypassing.

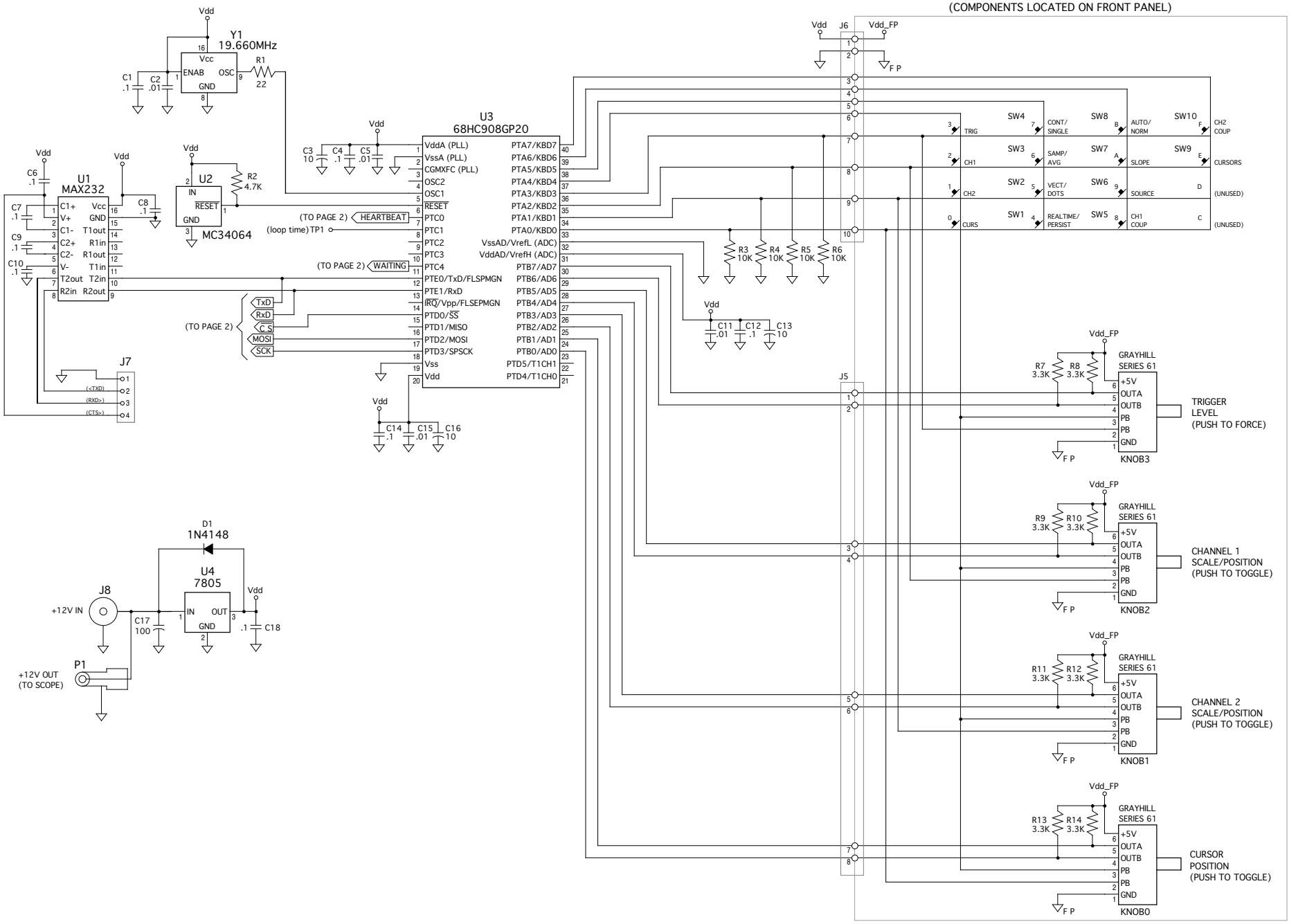
The \RESET signal for the 'GP20 is generated by U2 – an MC34064 three terminal power monitor. Upon power-up, U2 holds the CPU in \RESET until the supply voltage has exceeded 4.6V. At that time, the \RESET signal is released by U2's open collector output and pulled-up to Vcc by R2. This configuration provides a consistently clean \RESET signal during power-up.

Since the THS series DSOs required RS-232 signal levels for serial communication, it is necessary to level shift the serial signals to/from the CPU. U1, a MAX232, is used to accomplish the RS-232 level shifting. This ubiquitous part internally generates the required ±10V rails required by RS-232. Capacitors C6 through C10 comprise a switched-capacitor voltage doubler driven by the U1.

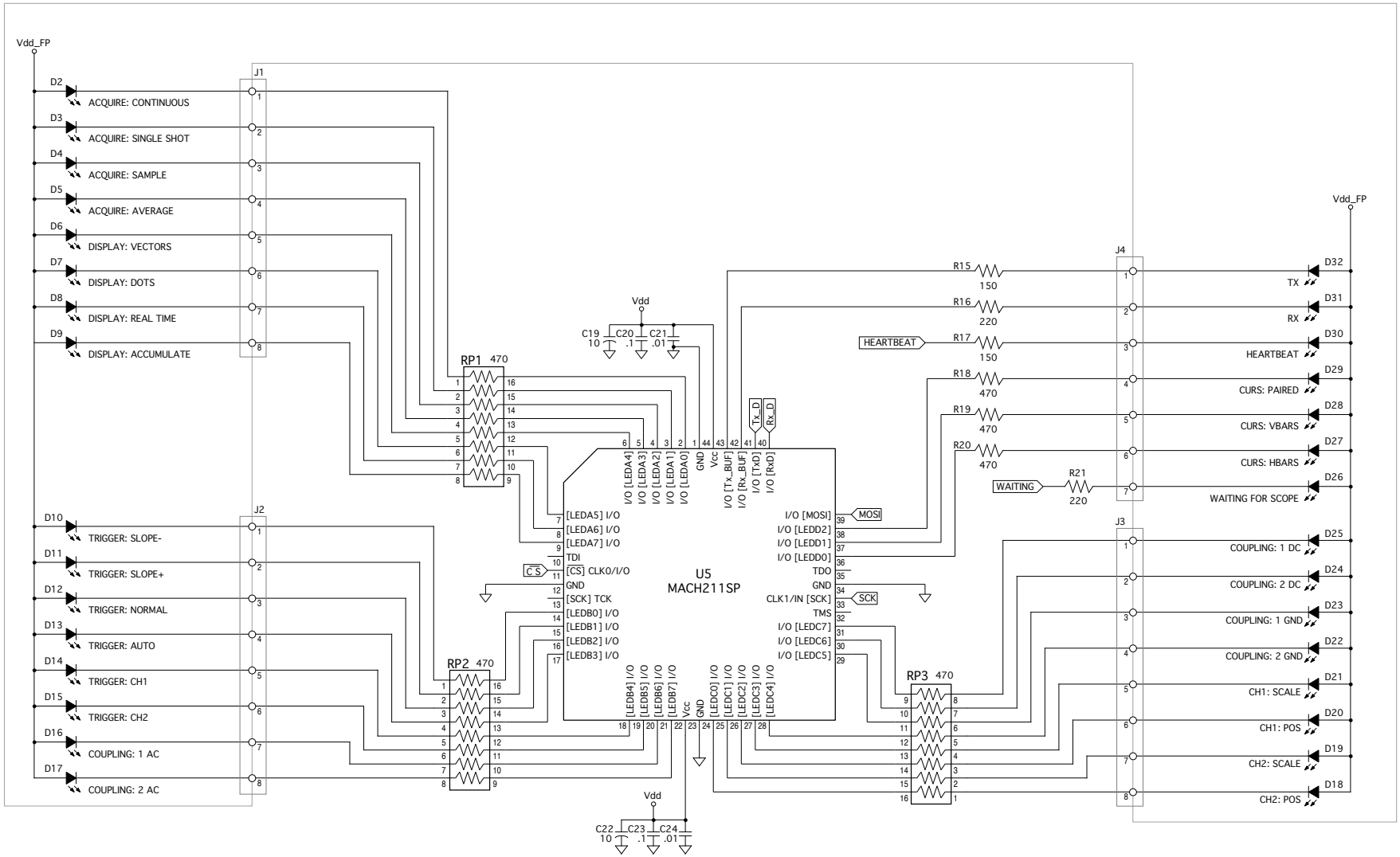
The sCo-Pilot's knobs are Grayhill Series 61 detented optical encoders (datasheet attached). These devices provide quadrature outputs which are acquired at port B of the CPU. The quadrature outputs are open-collector and are pulled-up by R7 through R14. The knobs also provide additional functionality in the form of momentary buttons integrated into the shafts.

The sCo-Pilot's function buttons are connected in a matrix to port A on the 'GP20. This matrix scanning arrangement enables 16 keys to be decoded by one 8 bit port. The keys labeled 0 through 3 are integral to the optical encoders while the others are discrete pushbuttons on the sCo-Pilot's front panel. Resistors R3 through R6 pull-down the key row signals when keys are not pressed.

The final component in the sCo-Pilot hardware is the SPI I/O expander CPLD. This device, U5, is a 64 macrocell EEPROM-based CPLD. A detailed description of the internal configuration of this component can be found the in CPLD section. As with all other ICs in this design, U5 incorporates local broadband power supply decoupling. U5 receives SPI data from the CPU and latches it into four LED ports. Resistor packs RP1 through RP3 provide current limiting for the 8 bit LED ports. Current limiting for all other LEDs is provided by resistors R15-R21.


(COMPONENTS LOCATED ON FRONT PANEL)
Y1
19.660MHz
U1
MAX232
U2
MC34064
U3
68HC908GP20
U4
7805
J6
J5
J7
J8
P1
D1
1N4148
SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8
SW9
SW10
KNOB0
KNOB1
KNOB2
KNOB3
GRAYHILL
SERIES 61
TRIGGER
LEVEL
(PUSH TO FORCE)
CHANNEL 1
SCALE/POSITION
(PUSH TO TOGGLE)
CHANNEL 2
SCALE/POSITION
(PUSH TO TOGGLE)
CURSOR
POSITION
(PUSH TO TOGGLE)
+12V IN
+12V OUT
(TO SCOPE)
HEARTBEAT
WAITING
(loop time) TP1
TxD
RxD
C_S
MOSI
SCK
Vdd
Vdd_FP
F P
GND

(COMPONENTS LOCATED ON FRONT PANEL)



## SERIES 61B
## 16, 24 or 32 Positions
## With or Without Pushbutton

### FEATURES
- Positions Screen Cursor
- More Friendly than Keyboards
- Permits Visual Concentration
- Economic Touchscreen Alternative
- Pushbutton for Entry Function
- Detent for Tactile Feedback and Minimal Backlash
- Optical Coupling for Long Life
- Rugged Construction

### APPLICATIONS

**Display Input**

The Series 61 rotary encoder switch can move cursor or icon on a display. Use the rotary and pushbutton switch to simply select a menu item and enter it, or write more elaborate display software. Use the Series 61 to input limit settings for a monitored function. Change an item on a checklist to a new value while viewing the remainder of the list.

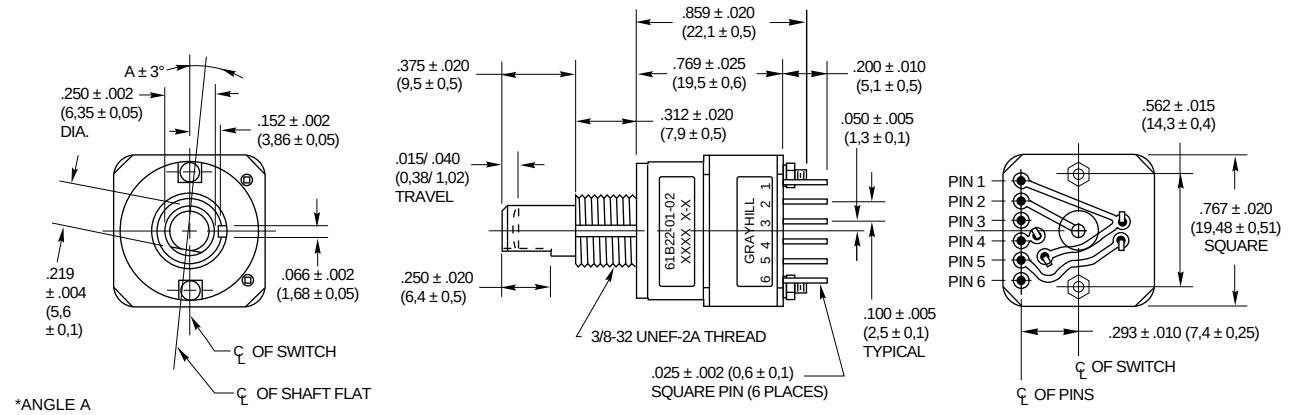
**Incremental Input**

Use the Series 61 with an interface chip to provide step by step input for setting radio frequency, drill depth, RPM, etc. These changes are usually a few steps, and you need not turn the switch several revolutions for the desired value. Some examples are as follows:

- Robot Position
- Radio Tuning
- Motor Control
- Volume Setting
- Limit Setting

### DIMENSIONS In inches (and millimeters)

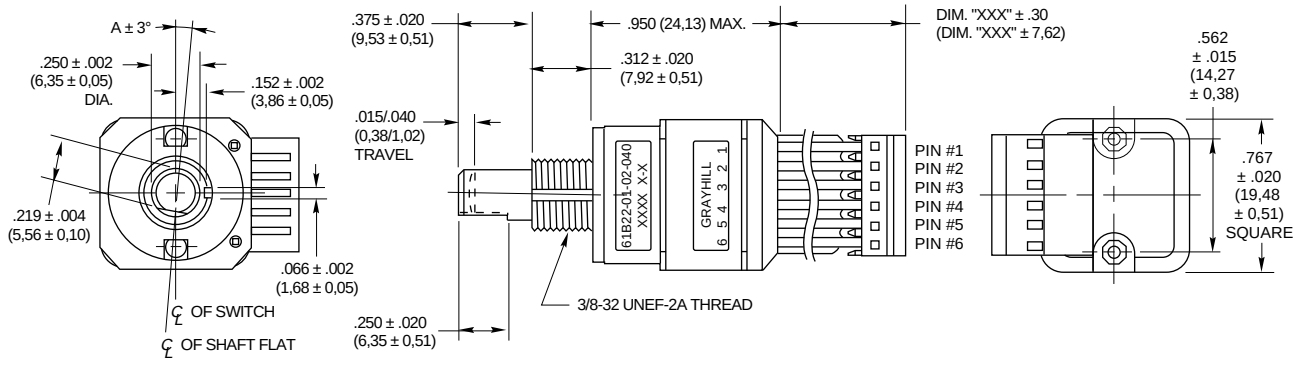
Diagram illustrates a 16-position switch with integral pushbutton switch.




*ANGLE A
Centerline of switch to perpendicular of shaft flat

| | |
|---|---|
| 32 Pos. | 5.63° |
| 24 Pos. | 7.50° |
| 16 Pos. | 11.25° |

Grayhill part number and date code marked on label. Customer part number marked on request.




Unless otherwise indicated, tolerances are ± .010 (0,25).

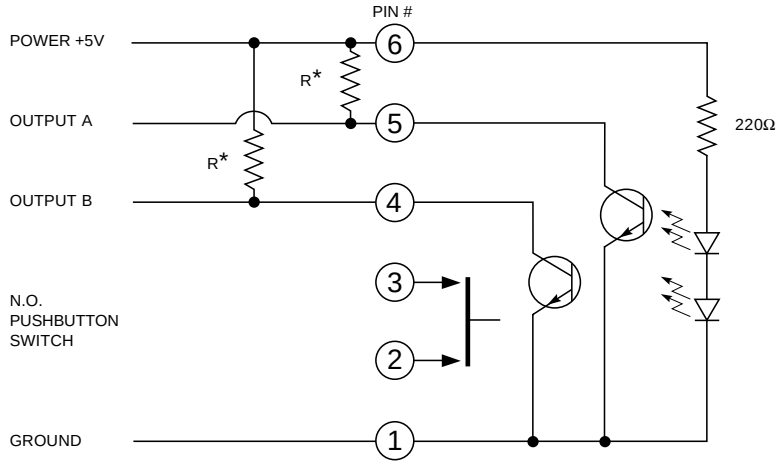
## SPECIFICATIONS

**Pushbutton Switch Ratings**
**Rating:** at 5 Vdc, 10 mA, Resistive
**Contact Resistance:** <10 (TTL or CMOS Compatible)
**Voltage Breakdown:** 250 VAC between mutually insulated parts.
**Contact Bounce:** Less than 4 milliseconds at make and less than 10 milliseconds at break.
**Actuation Life:** 3,000,000 operations.
**Actuation Force:** Maximum actuation force of 450 grams and a minimum actuation force of 300 grams.

**Rotary Encoder Ratings**
**Coding:** 2 bit quadrature coded output.
**Operating Voltage:** 5V ± .25VDC
**Supply Current:** 30 mA Maximum @ 5 VDC
**Logic High:** 3.8V Minimum
**Logic Low:** 0.8V Maximum

**Logic Rise and Fall Times:** Rise Time < 30 ms @ 16.6 rpm. Fall Time < 30 ms @ 16.6 rpm.
**Operating Torque:** 1.5 in-oz ± 50%
**Rotational Life:**>1,000,000 cycles of operation (1 cycle is rotation through all positions and return)
**Shaft Push Out Force:** 50 lbs min.
**Mounting Torque:** 20 in-lb max.
**Environmental**
**Operating Temperature Range:**-40°C to 65°C
**Storage Temperature Range:** -40°C to 65°C
**Vibration Resistance:** Tested at 10–2000 Hz at 15g or 0.060" double amplitude.
**Mechanical Shock:** Tested at 100 g's, 6 ms, half sine, 12.3 ft/s and 100 g's, 6 ms, sawtooth, 9.7 ft/s.
**Humidity:** 90–95% Relative Humidity @ 40 C for 96 hrs.
**Materials and Finishes**
**Detent Cover:** Thermosetting plastic
**Bushing:** Zinc casting, Cadmium plated per QQP–416, Class 2, Type II
**Shaft:** Reinforced thermoplastic
Note: Earlier versions may have electropolished stainless steel shafts. (Still available in customs only)
**Detent Balls:** Steel, nickel plated
**Detent Spring:** Tinned music wire
**PC Boards:** NEMA Grade FR-4
**Board Terminals:** Copper alloy, CDA No. 725
**Through Bolts:** Stainless steel, unplated
**Through Bolt Nuts:** Stainless steel
**Switch Assembly Cover and Code Rotor:** PBT polyester thermoplastic
**Mounting Hardware:** One brass, Cadmium plated nut and lockwasher supplied with each switch. Nut is 0.094" thick by 0.562" across flats
**Aperture:** Brass, black oxide finish
**Strain Relief:** PBT polyester thermoplastic (cable version only)
**Cable:** 26 AWG, stranded/tinned wire, PVC coated on .100 (2,54) centers (cable version only)

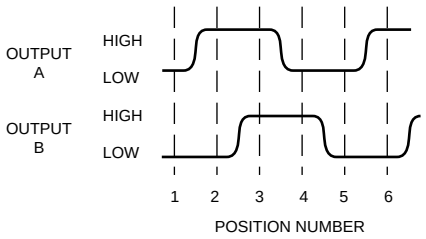
## CIRCUITRY, TRUTH TABLE, AND WAVEFORM–Standard Quadrature 2-Bit Code




*External pull up resistors required for operation.
8.2 kΩ is suggested for TTL; 3.3 kΩ is suggested for CMOS.

| Clockwise Rotation | | |
|---|---|---|
| Position | Output A | Output B |
| 1 | | |
| 2 | ● | |
| 3 | ● | ● |
| 4 | | ● |

● Indicates logic high; blank indicates logic low. Code repeats every 4 positions.




## ORDERING INFORMATION


**Series**
**Style:** B = Standard, unsealed
**Angle of Throw:** 11 = 11.25° or 32 Positions
15 = 15° or 24 Positions
22 = 22.25° or 16 Positions
**Coding:** 01 = Quadrature
**Pushbutton Option:** 01 = Without pushbutton, 02 = With pushbutton

**Termination:** Blank (no dash or numbers) = pins as described in drawing
Cable Termination 020 = 2.0 inches minimum to 250 = 25 inches maximum
Provided in increments of 1/2 inch. Example 035 = 3.5", 060 = 6 inches. Cable is terminated with standard Amp Connector 640442-6. Use any 6 position .100 center header to mate with the cable assembly. Contact Grayhill for Series 61B without a detent.

## ACCESSORIES

See page E-14.

## OPTIONS

**Not available thru Distributors**
Different shaft and bushing lengths, shaft/panel seal, and additional supply voltages are possible.

**Available from your local Grayhill Distributor**
For prices and discounts, contact a local Sales Office, an authorized local Distributor, or Grayhill.


Grayhill INC. An ISO-9001 Company
561 Hillgrove Avenue • LaGrange, Illinois 60525 • USA • Phone: (708) 354-1040 • Fax: (708) 354-2820 • http://www.grayhill.com

sCo Pilot
Human Interface Enhancement

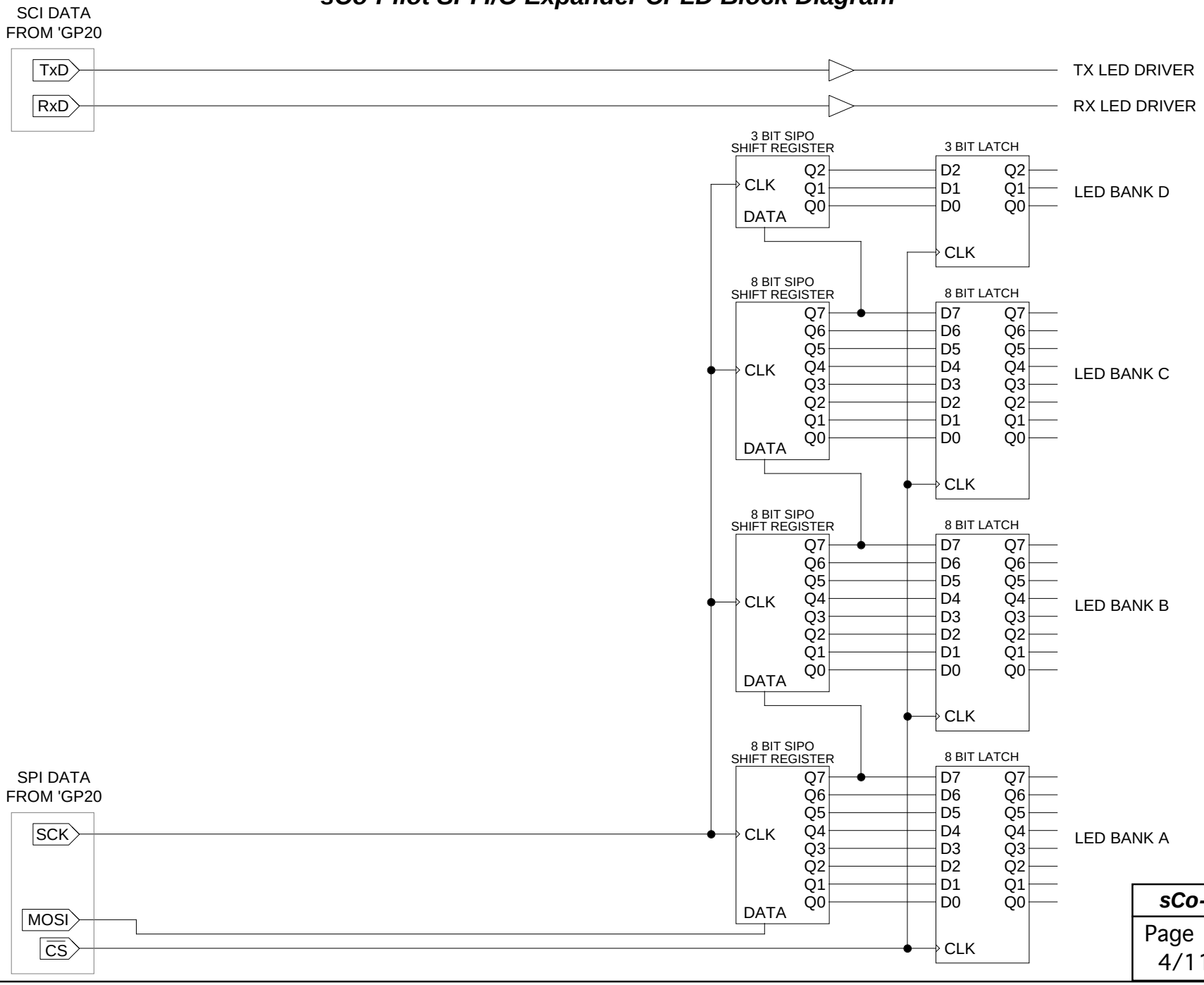
SPI I/O Expander CPLD Overview

Because of the large number of front panel LEDs required by the sCo-Pilot, an additional set of LED ports was needed. There just aren't enough port bits on the 'GP20 to support all of the LEDs in addition to the keyboard and quadrature (knob) inputs.

The SPI I/O expander was designed to alleviate the shortage of LED port bits. This design, targeted at a Vantis (AMD/Lattice) MACH 211 64 macrocell CPLD, provides three additional 8 bit LED ports, one additional 3 bit LED port, and buffering for RS-232 TX and RX LEDs. This device was chosen based on availability of sample parts and inexpensive development tools. The starter kit, including programmer, was already on-hand for the sCo-Pilot project. Serendipitously, it happens that the MACH211 is a perfect fit for this application. It has exactly enough internal resources and external pins to accomplish its two functions (LED ports and LED buffering).

The design was captured in ABEL and compiled using the Vantis-specific Synario starter software. A block diagram depicting the functionality of the CPLD and the associated ABEL code follow. The ABEL source file is available on the enclosed floppy as SPI_IO.ABL.

# sCo-Pilot SPI I/O Expander CPLD Block Diagram

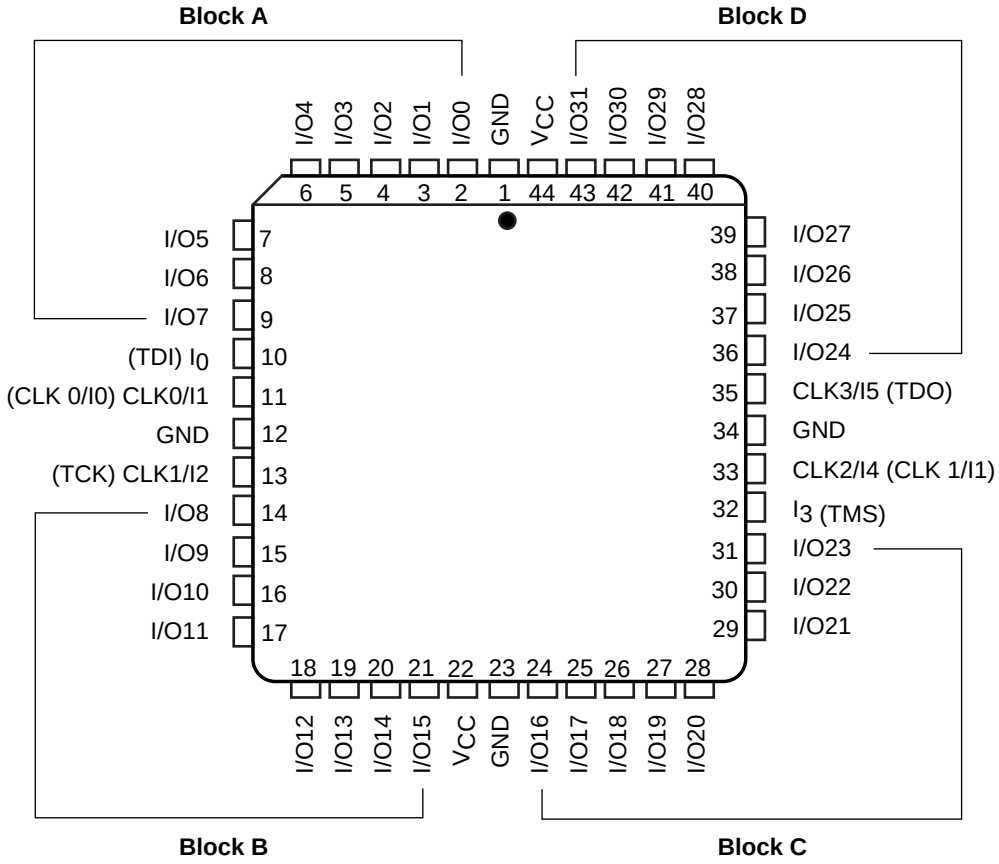



| sCo-Pilot |
| --- |
| Page 1 of 1<br>4/11/99 |

## CONNECTION DIAGRAM (MACH211-7/10/12/15 AND MACH211SP-6/7/10/12/15)

### Top View

**44-Pin PLCC**




14051K-023

## PIN DESIGNATIONS

CLK/I = Clock or Input
GND = Ground
I = Input
I/O = Input/Output
$V_{CC}$ = Supply Voltage
TDI = Test Data In
TCK = Test Clock
TMS = Test Mode Select
TDO = Test Data Out

***Note:***

*1. Pin designators in parentheses ( ) apply to the MACH211SP*

```
MODULE SPI_IO
TITLE 'SPI I/O Expander'
"  Author: Derek Matsunaga
" Project: Sco-Pilot
"Revision: 4/11/99
"  Target: Vantis MACH211 (PLCC44)

"Note: Consolidating explicit signal series into SETs causes improper compilation.

"Operation:
"LED data is sent to this device via the standard SPI signals MOSI, SCK, and \CS.
"The data is sent as four consecutive eight bit bytes, MSB first.  The first
"byte ends-up in port D and its upper five bits are lost (since port D is only
"three bits wide).  The second byte is port C, the third byte is port B, and the last
"byte is port A.  The data is clocked into an internal shift register on the rising
"edge of SCK.  The rising edge of \CS loads the data from the internal shift register
"into the LED port pins.  This part also contains combinatorial buffers for the Tx
"and Rx LEDs.

declarations

true = 1;
false = !true;

"LED drive pins:
LED_PORTA0..LED_PORTA7       pin 2,3,4,5,6,7,8,9          istype 'reg_d,buffer';       "LED port A
LED_PORTB0..LED_PORTB7       pin 14,15,16,17,18,19,20,21  istype 'reg_d,buffer';       "LED port B
LED_PORTC0..LED_PORTC7       pin 24,25,26,27,28,29,30,31  istype 'reg_d,buffer';       "LED port C
LED_PORTD0..LED_PORTD2       pin 36,37,38                 istype 'reg_d,buffer';       "LED port D

"TX and RX LED buffers:
RX_IN                        pin 40;
TX_IN                        pin 41;
RX_BUF                       pin 42 istype 'com';
TX_BUF                       pin 43 istype 'com';

"SPI signals:
MOSI                         pin 39;       "Serial data in.
CS                           pin 11;       "Update port pins.
SCK                          pin 33;       "Data clock.

"Internal shift register:
A0..A7                       node istype 'reg_d,buffer';  "Internal node for clocking-in port A data.
B0..B7                       node istype 'reg_d,buffer';  "Internal node for clocking-in port B data.
C0..C7                       node istype 'reg_d,buffer';  "Internal node for clocking-in port C data.
D0..D2                       node istype 'reg_d,buffer';  "Internal node for clocking-in port D data.

equations

"The buffered Tx and Rx signals:
TX_BUF = TX_IN;
RX_BUF = RX_IN;

"Clock enables:
LED_PORTA0.ce = true;
LED_PORTA1.ce = true;
LED_PORTA2.ce = true;
LED_PORTA3.ce = true;
LED_PORTA4.ce = true;
LED_PORTA5.ce = true;
LED_PORTA6.ce = true;
LED_PORTA7.ce = true;
```

```
LED_PORTB0.ce = true;
LED_PORTB1.ce = true;
LED_PORTB2.ce = true;
LED_PORTB3.ce = true;
LED_PORTB4.ce = true;
LED_PORTB5.ce = true;
LED_PORTB6.ce = true;
LED_PORTB7.ce = true;

LED_PORTC0.ce = true;
LED_PORTC1.ce = true;
LED_PORTC2.ce = true;
LED_PORTC3.ce = true;
LED_PORTC4.ce = true;
LED_PORTC5.ce = true;
LED_PORTC6.ce = true;
LED_PORTC7.ce = true;

LED_PORTD0.ce = true;
LED_PORTD1.ce = true;
LED_PORTD2.ce = true;

"Output enables:
LED_PORTA0.oe = true;
LED_PORTA1.oe = true;
LED_PORTA2.oe = true;
LED_PORTA3.oe = true;
LED_PORTA4.oe = true;
LED_PORTA5.oe = true;
LED_PORTA6.oe = true;
LED_PORTA7.oe = true;

LED_PORTB0.oe = true;
LED_PORTB1.oe = true;
LED_PORTB2.oe = true;
LED_PORTB3.oe = true;
LED_PORTB4.oe = true;
LED_PORTB5.oe = true;
LED_PORTB6.oe = true;
LED_PORTB7.oe = true;

LED_PORTC0.oe = true;
LED_PORTC1.oe = true;
LED_PORTC2.oe = true;
LED_PORTC3.oe = true;
LED_PORTC4.oe = true;
LED_PORTC5.oe = true;
LED_PORTC6.oe = true;
LED_PORTC7.oe = true;

LED_PORTD0.oe = true;
LED_PORTD1.oe = true;
LED_PORTD2.oe = true;

"Clock definitions for LED ports:
LED_PORTA0.clk = CS;
LED_PORTA1.clk = CS;
LED_PORTA2.clk = CS;
LED_PORTA3.clk = CS;
LED_PORTA4.clk = CS;
LED_PORTA5.clk = CS;
```

```
LED_PORTA6.clk = CS;
LED_PORTA7.clk = CS;

LED_PORTB0.clk = CS;
LED_PORTB1.clk = CS;
LED_PORTB2.clk = CS;
LED_PORTB3.clk = CS;
LED_PORTB4.clk = CS;
LED_PORTB5.clk = CS;
LED_PORTB6.clk = CS;
LED_PORTB7.clk = CS;

LED_PORTC0.clk = CS;
LED_PORTC1.clk = CS;
LED_PORTC2.clk = CS;
LED_PORTC3.clk = CS;
LED_PORTC4.clk = CS;
LED_PORTC5.clk = CS;
LED_PORTC6.clk = CS;
LED_PORTC7.clk = CS;

LED_PORTD0.clk = CS;
LED_PORTD1.clk = CS;
LED_PORTD2.clk = CS;

"Clock enables for shift register nodes:
A0.ce = true;
A1.ce = true;
A2.ce = true;
A3.ce = true;
A4.ce = true;
A5.ce = true;
A6.ce = true;
A7.ce = true;

B0.ce = true;
B1.ce = true;
B2.ce = true;
B3.ce = true;
B4.ce = true;
B5.ce = true;
B6.ce = true;
B7.ce = true;

C0.ce = true;
C1.ce = true;
C2.ce = true;
C3.ce = true;
C4.ce = true;
C5.ce = true;
C6.ce = true;
C7.ce = true;

D0.ce = true;
D1.ce = true;
D2.ce = true;

"Output enables for shift register nodes:
A0.oe = true;
A1.oe = true;
A2.oe = true;
A3.oe = true;
```

```
A4.oe = true;
A5.oe = true;
A6.oe = true;
A7.oe = true;

B0.oe = true;
B1.oe = true;
B2.oe = true;
B3.oe = true;
B4.oe = true;
B5.oe = true;
B6.oe = true;
B7.oe = true;

C0.oe = true;
C1.oe = true;
C2.oe = true;
C3.oe = true;
C4.oe = true;
C5.oe = true;
C6.oe = true;
C7.oe = true;

D0.oe = true;
D1.oe = true;
D2.oe = true;

"Clock definitions for shift register nodes:
A0.clk = SCK;
A1.clk = SCK;
A2.clk = SCK;
A3.clk = SCK;
A4.clk = SCK;
A5.clk = SCK;
A6.clk = SCK;
A7.clk = SCK;

B0.clk = SCK;
B1.clk = SCK;
B2.clk = SCK;
B3.clk = SCK;
B4.clk = SCK;
B5.clk = SCK;
B6.clk = SCK;
B7.clk = SCK;

C0.clk = SCK;
C1.clk = SCK;
C2.clk = SCK;
C3.clk = SCK;
C4.clk = SCK;
C5.clk = SCK;
C6.clk = SCK;
C7.clk = SCK;

D0.clk = SCK;
D1.clk = SCK;
D2.clk = SCK;


"The actual shift register:
A0 := MOSI;
```

```
A1 := A0;
A2 := A1;
A3 := A2;
A4 := A3;
A5 := A4;
A6 := A5;
A7 := A6;

B0 := A7;
B1 := B0;
B2 := B1;
B3 := B2;
B4 := B3;
B5 := B4;
B6 := B5;
B7 := B6;

C0 := B7;
C1 := C0;
C2 := C1;
C3 := C2;
C4 := C3;
C5 := C4;
C6 := C5;
C7 := C6;

D0 := C7;
D1 := D0;
D2 := D1;

"Define how the shift register is loaded into the output pins:
LED_PORTA0 := A0;
LED_PORTA1 := A1;
LED_PORTA2 := A2;
LED_PORTA3 := A3;
LED_PORTA4 := A4;
LED_PORTA5 := A5;
LED_PORTA6 := A6;
LED_PORTA7 := A7;

LED_PORTB0 := B0;
LED_PORTB1 := B1;
LED_PORTB2 := B2;
LED_PORTB3 := B3;
LED_PORTB4 := B4;
LED_PORTB5 := B5;
LED_PORTB6 := B6;
LED_PORTB7 := B7;

LED_PORTC0 := C0;
LED_PORTC1 := C1;
LED_PORTC2 := C2;
LED_PORTC3 := C3;
LED_PORTC4 := C4;
LED_PORTC5 := C5;
LED_PORTC6 := C6;
LED_PORTC7 := C7;

LED_PORTD0 := D0;
LED_PORTD1 := D1;
LED_PORTD2 := D2;
END
```

sCo-Pilot
Human Interface Enhancement

Software overview

The software for the sCo-Pilot was written in "state machine" style completely in CPU08 assembly. Assembly language was chosen strictly based on the availability of free development tools. Being a "personal" project, there was no budget for a C compiler. However, a combination of C and assembly would have been more optimal for this project.

The sCo-Pilot software requires 139 RAM locations to store global and temporary variables. All of these RAM location exist on the zero page to take advantage of the CPU08's direct addressing mode. With 139 RAM variables located sequentially starting at address $0000, this leaves 116 bytes available for the CPU stack. Since the stack grows downward from $00FF, this is more than enough room to accommodate many nested subroutine calls and stack PUSHes. The sCo-Pilot software nests a maximum of four subroutines at any given time. The remainder of the 'GP20's 512 bytes of RAM is located above $00FF. This space has 20 bytes reserved for an outgoing RS-232 buffer. All other RAM locations are unused.

When the sCo-Pilot comes out of reset, it first clears all RAM locations. Then the stack pointer is reset to $00FF. Following the stack pointer reset, the watchdog is disabled and the SCI, SPI, and TIM modules are initialized by calling the appropriate subroutines. Then, some miscellaneous variables are initialized and the software produces a power-on "light show" to enable the user to verify that all LEDs are working. Following the light show, the software sets the "inquiry bit" for each state machine handler (diagrams attached, discussion follows) in order to obtain the current status of the scope – thereby lighting the appropriate front panel LEDs. With the inquiry bits set, the software begins execution of the main loop. Since the software operates in a "state machine" style, there is not waiting between setting the inquiry bits and entering the main loop. In fact, the execution of the main loop is required for the initial inquiry process to complete. The initial state inquiry is paced by the scope.

The sCo-Pilot's main loop is continuously executed according to the following code. Because of the "state machine" structure of the software, there are no "if-then" comparisons in the main loop. All elements of the main loop are executed *every time* through the loop. Because all of the main loop subroutines execute as independent state machines or operate on global variables, the calling order is irrelevant and there is no advantage to calling a specific subroutine before or after any other.

```
main_loop:  bclr    1,PTC           ;  Set loop time pin.
            bset    1,PTC           ;  Clear loop time pin.
            jsr     update_LEDs     ;  Send the LED data to the IO expander.
            jsr     read_keys       ;  Read the key matrix.
            jsr     debounce_keys   ;  Debounce any pressed keys.
            jsr     do_keys         ;  Perform key and knob actions if necessary.
            jsr     read_knobs      ;  Read the knobs.
            jsr     heartbeat       ;  Update the heartbeat.
            jsr     out232          ;  Send pending RS-232 output (if any).
            jsr     in232           ;  See if there is RS-232 input available.
            jsr     update_accel    ;  Update the knob acceleration values.
            jsr     update_idle     ;  If sCo-Pilot is idle for a certain time, update status.
            jsr     check_waiting   ;  Determine if the sCo-Pilot has been waiting for data for too long.
            bra     main_loop       ;  Continue to loop.
```

This code segment is presented literally as it appears in the source file. The function of each subroutine is described by its name and the comments that follow. Under normal operating conditions, the main loop requires about 270$\mu$s to execute. This loop time changes, however, depending on the status of each state machine subroutine. The main loop continuously executes until power is removed from the sCo-Pilot. A "ping" on port C bit 1 is used to indicate loop time.

The do_keys routine selects a state machine key (or knob) handler to execute if a key is pressed or if a knob is turned. If there are no pending key actions, no state machine key handler is selected. Each of the sCo-Pilot's 14 keys are assigned independent state machine handlers. For example, the trigger slope select button is handled by a completely different state machine than trigger source select. This method of handling keys and knob turns was selected to deal with the variable control format of the THS series DSOs.

The RS-232 command format of the THS series is not conducive to embedded control. It requires vastly different commands for similar functions and returns exponential floating point values as well as text, depending on the inquiry commands and scope mode. As a result, it was most convenient to implement each key response in a different state machine handler. The large FLASH array of the 'GP20 made this approach realizable.

The key handler routines are listed as follows:

```
key_0           ;  Key 0 handler (cursor knob toggle).
key_1           ;  Key 1 handler (horizontal knob toggle).
key_2           ;  Key 2 handler (vertical knob toggle).
key_3           ;  Key 3 handler (trigger knob toggle).
key_4           ;  Key 4 handler (realtime/persist button).
key_5           ;  Key 5 handler (vectors/dots button).
key_6           ;  Key 6 handler (sample/average button).
key_7           ;  Key 7 handler (continuous/singleshot button).
key_8           ;  Key 8 handler (channel 1 coupling button).
key_9           ;  Key 9 handler (trigger source button).
key_10          ;  Key 10 handler (trigger slope button).
key_11          ;  Key 11 handler (auto/normal trigger button).
key_12          ;  Key 12 handler (unused).
key_13          ;  Key 13 handler (unused).
key_14          ;  Key 14 handler (cursor type select button).
key_15          ;  Key 15 handler (channel 2 coupling button).
knob_3CW        ;  Knob 3 clockwise handler (trigger level increase).
knob_3CCW       ;  Knob 3 counterclockwise handler (trigger level decrease).
knob_2CW        ;  Knob 2 clockwise handler (vertical increase).
knob_2CCW       ;  Knob 2 counterclockwise handler (vertical decrease).
knob_1CW        ;  Knob 1 clockwise handler (horizontal increase).
knob_1CCW       ;  Knob 1 counterclockwise handler (horizontal decrease).
knob_0CW        ;  Knob 0 clockwise handler (cursor increase).
knob_0CCW       ;  Knob 0 counterclockwise handler (cursor decrease).
```

When the do_keys routine is called as part of the main loop, it selects the appropriate state machine handler from a jump table and stores the address of the handler as the next handler to run when do_keys is called. If no actions are pending (no key or knob handlers running), the do_keys routine stores its own address as the next handler to run.

While a key state machine handler is running, it "locks-out" the execution of other handlers, thus preventing different multiple actions from running simultaneously and confusing the scope. At the beginning of each state machine handler is a test to determine if there is RS-232 data to be sent or if the state machine is waiting for RS-232 data from the scope. If either of these cases is true, the state machine will return control to the main loop until next time through (i.e. nothing will be done this time). When a state machine handler has completed, it clears it's associated "action required" bit which allows the next key handler (if any) to run. In this way, only one state machine handler can run at any given time but multiple key presses will be "queued" and executed when the previous handler has finished.

Key presses are detected and conditioned by the subroutines read_keys and debounce_keys. The

read_keys routine scans the key matrix and sets key state bits if keys are pressed and clears them if not. This is the only function performed by the read_keys routine. The remainder of the key processing tasks are accomplished in debounce_keys. This routine examines the key state bits to determine if each key's debounce timer should be incremented or reset. When a key is continuously pressed for the debounce time (32ms), the "action required" bit for that key is asserted. This informs the do_keys routine to invoke the state machine handler for that key. The "action required" bit for that key is cleared when the state machine execution is complete.

The read_knobs routine determines if a knob is being turned and if so, which direction. Since the knob inputs are optical, it is not necessary to debounce them. The quadrature outputs from the knobs are decoded by means of a lookup table. The previous state of the knob forms the upper two bits of the LUT address while the present state of the knob comprises the lower two bits. A value indicating clockwise or counter clockwise is fetched from a lookup table based on this four bit address. As with a key press, the knob's "action required" bit is set when it is turned. Responses to knobs are dealt with using state machine handlers, just as with the keys.

When a knob is turned, an acceleration countdown timer is started. This 8 bit timer value is decremented by the update_accel routine every millisecond. If the knob is turned again before the countdown timer expired, an acceleration value is retrieved from a lookup table based on how much time elapsed since the last knob turn. This acceleration value indicates how many times the addition or subtraction of the increment value will occur in the state machine handler. In other words, if the acceleration value is 10, the state machine handler will execute its increment or decrement function 10 times during a single pass through the handler. The lookup table values may be modified (in the code) to change the "feel" of the acceleration.

RS-232 communication between the sCo-Pilot and the scope is polled rather than interrupt driven. Because the main loop executes in less than 300$\mu$s, it is possible to poll incoming 9600BPS serial data without the possibility of losing data. The in232 routine checks the SCI for a received byte in the input buffer. If the SCI input buffer is empty, in232 returns to the main loop. Otherwise, the incoming character is retrieved and placed in the sCo-Pilot's input buffer, inbuf. This process continues until the scope's termination character (ASCII 10) is received. At this point, the waiting flag is cleared to allow the state machine handlers to continue executing. The "waiting" LED on the front panel is controlled by this waiting flag.

Outgoing RS-232 data is also handled in a polled manner. If there is output data pending, a maximum of one character per call is transmitted. Pending output data is indicated by a non-zero high byte of the out_data_ptr. This requires that all outgoing messages be located above the zero page in memory. Because of the fast loop time of the main loop, it is possible that the previous output character transmission may not be completed when out232 is called. If so, the out232 is exited until the next time through the main loop. When the SCI's transmit data buffer is empty, the next character in the buffer pointed to by out_data_ptr is loaded into the transmitter. This process is repeated until out232 encounters the null termination character ($00). At this point, the high byte of out_data_ptr is cleared to indicate the end of transmission. This polled arrangement using in232 and out232 enables RS-232 data to be sent and received at full speed with no waiting.

The update_LEDs routine is used to send all four LED status bytes to the SPI I/O expander. This routine executes identically each time it is called. All four LED status bytes are sent to the SPI I/O expander at full SCI speed. The chip select signal is used to clock-in the four data bytes into the SPI I/O expander IC.

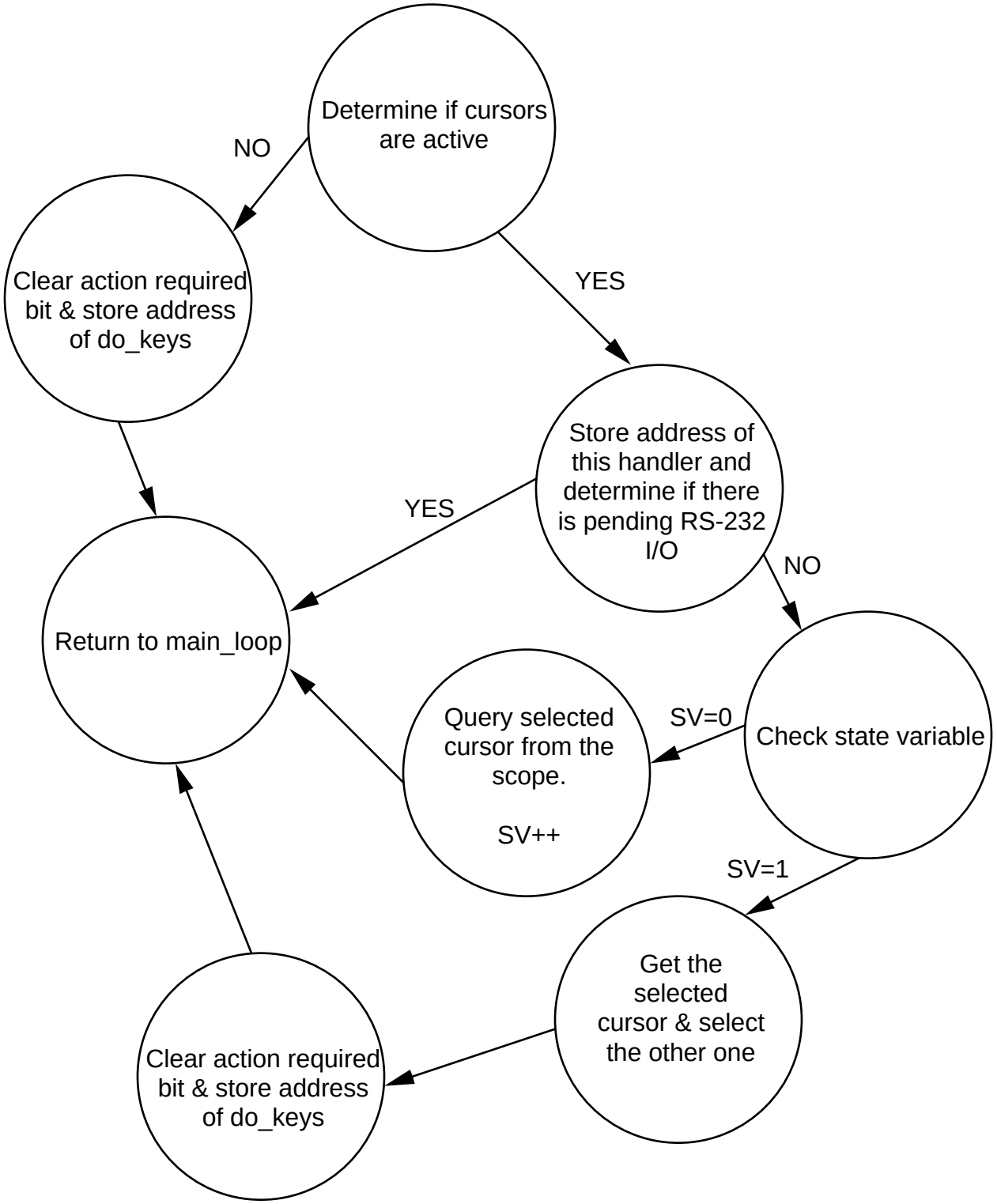
When the sCo-Pilot sits idle for a predetermined amount of time (30 seconds at the time of this writing), the inquiry request is sent to all pertinent state machine handlers. This is the same process by

which the sCo-Pilot initially ascertains the scope's status. Performing this action during idle time helps to ensure that all front panel LEDs are synchronized with the scope in case the user changed a quantity or mode from the scope's front panel. As part of the main loop, the `update_idle` routine is responsible for keeping track of the idle time. The idle time is reset whenever a key press or knob turn is detected.

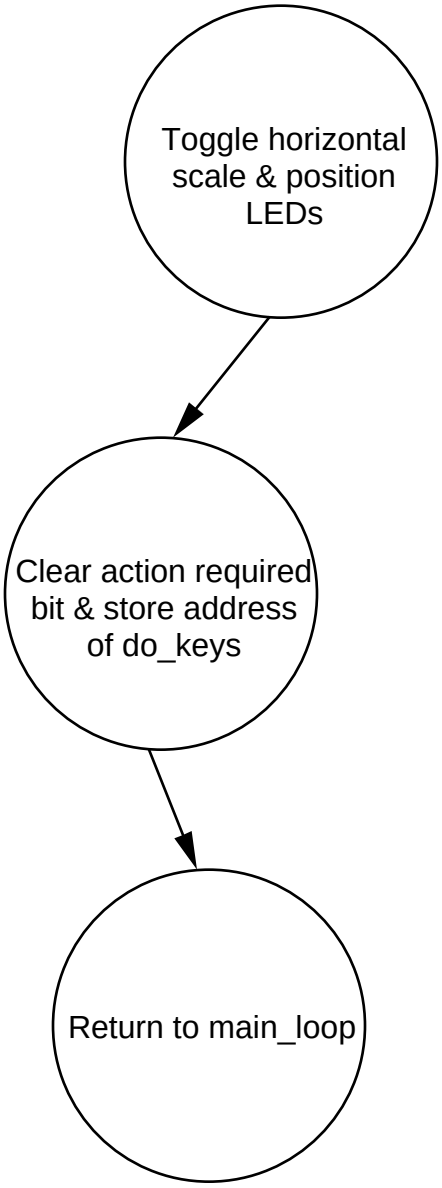
The final significant component of the main loop is the `check_waiting` subroutine. This routine keeps track of how much time the sCo-Pilot has been waiting for a response from the scope. This limit is currently set at 3 seconds. If the data cable becomes disconnected or the scope cannot respond within 3 seconds for some other reason, the scope's termination character is transmitted to reset the scope's input buffer. Also, the global state variable (`temp_state`) is reset to zero to cause the currently running state machine handler to "start over". This process will continue until communication with the scope is reestablished. With this feature, it is possible to issue a command to the scope, unplug the data cable during a transmission, reconnect the data cable, and guarantee that the command will be completed in its entirety.

The latest version of the sCo•Pilot software, dated May 21, 1999, has been stable and running since that date. It has been put through extensive testing using the sCo•Pilot hardware and the THS scope. No known bugs or improvements exist at this time. Note that the source code is heavily commented and modularized for future maintenance. The compiled code requires 5927 bytes of code space (including vector table), utilizing approximately 30% of the available FLASH in the 'GP20 (leaving plenty of room for expansion). Because the entire 5927 bytes was written in assembly language, it was important to include numerous detailed comments as part of the source file. The source file is 170K bytes, thus indicating a high comment-to-code ratio and presumably easy code maintenance in the future.

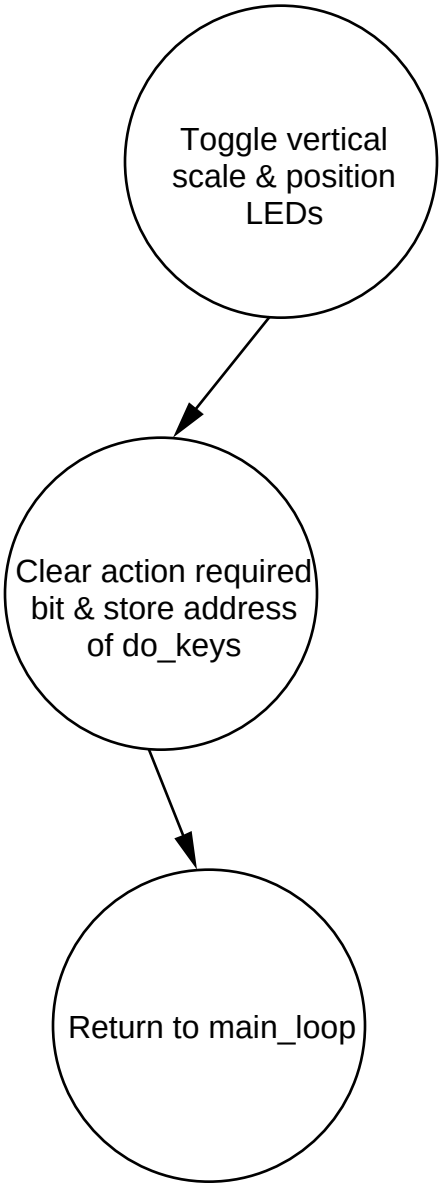
State transition diagram for the key_0 handler
(cursor knob toggle)



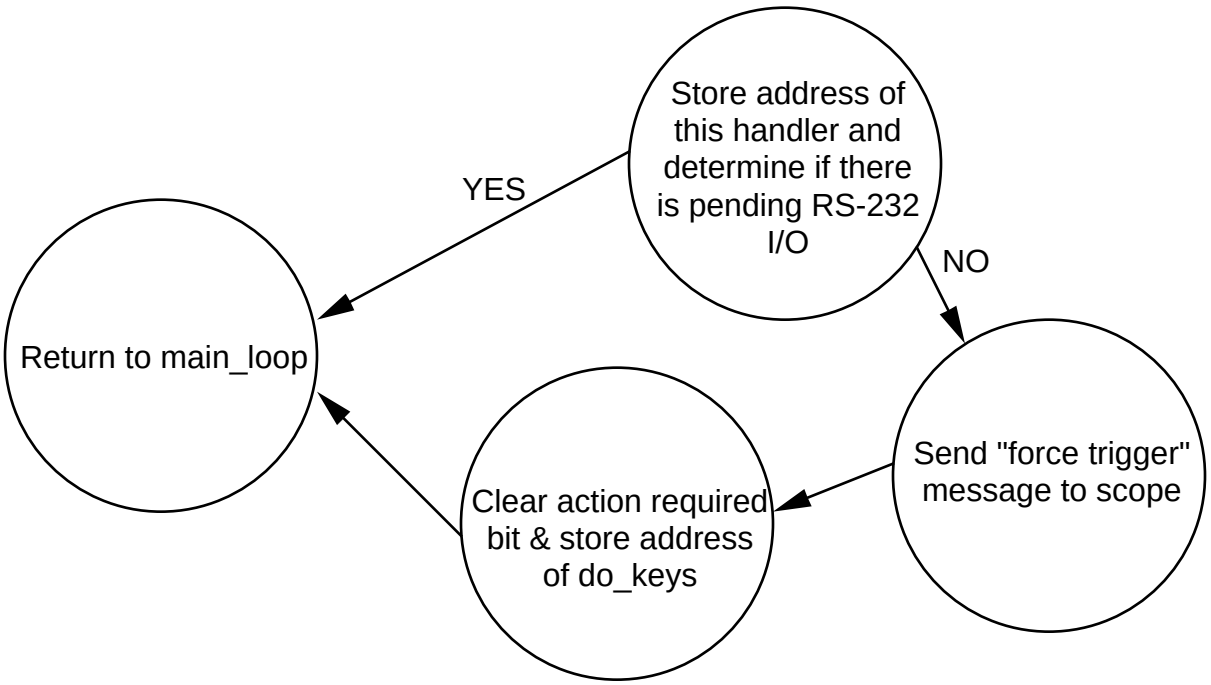
State transition diagram for the key_1 handler
(horizontal knob toggle)
(not a real state machine because there is no state variable)



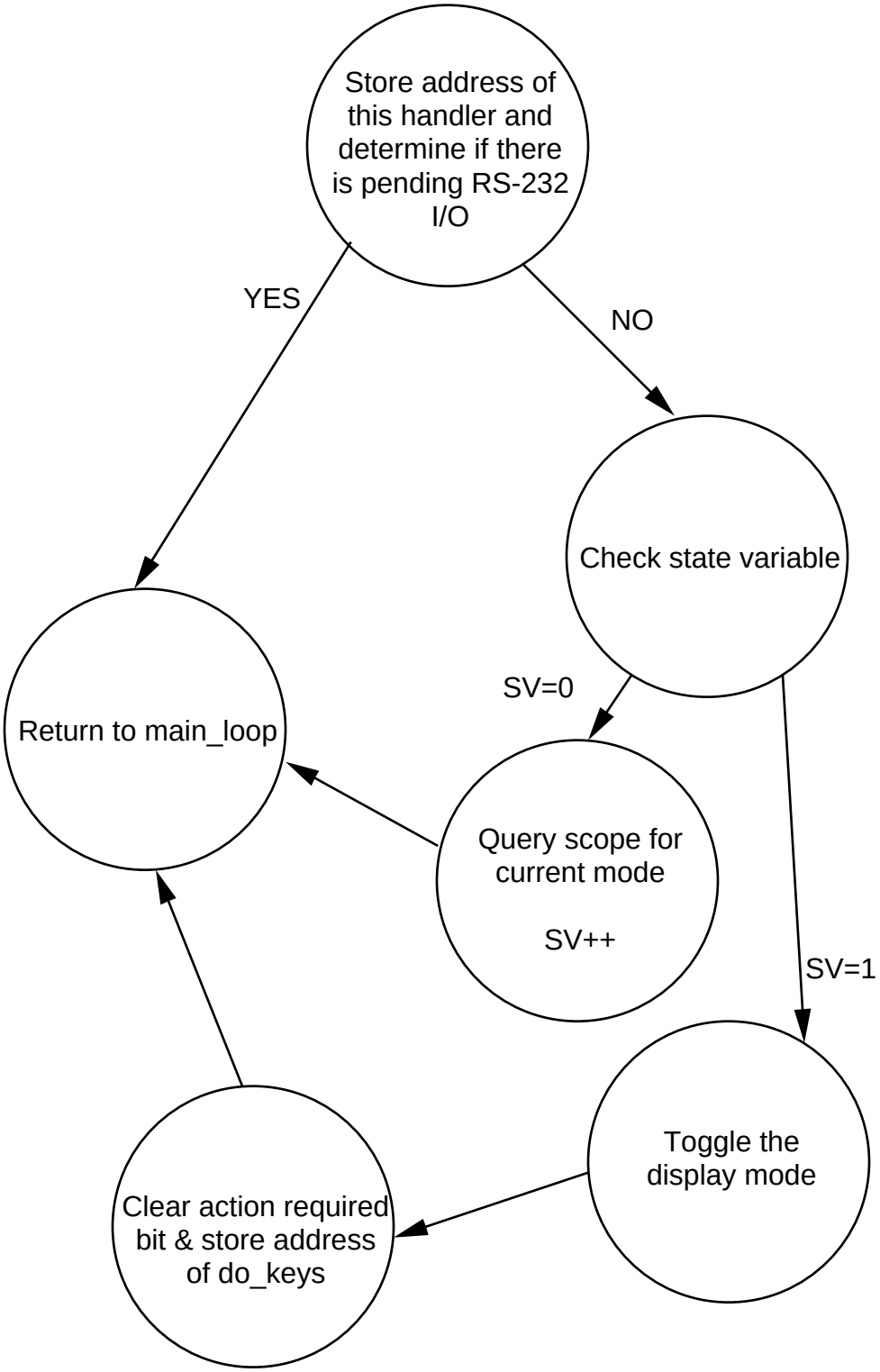
State transition diagram for the key_2 handler
(vertical knob toggle)
(not a real state machine because there is no state variable)



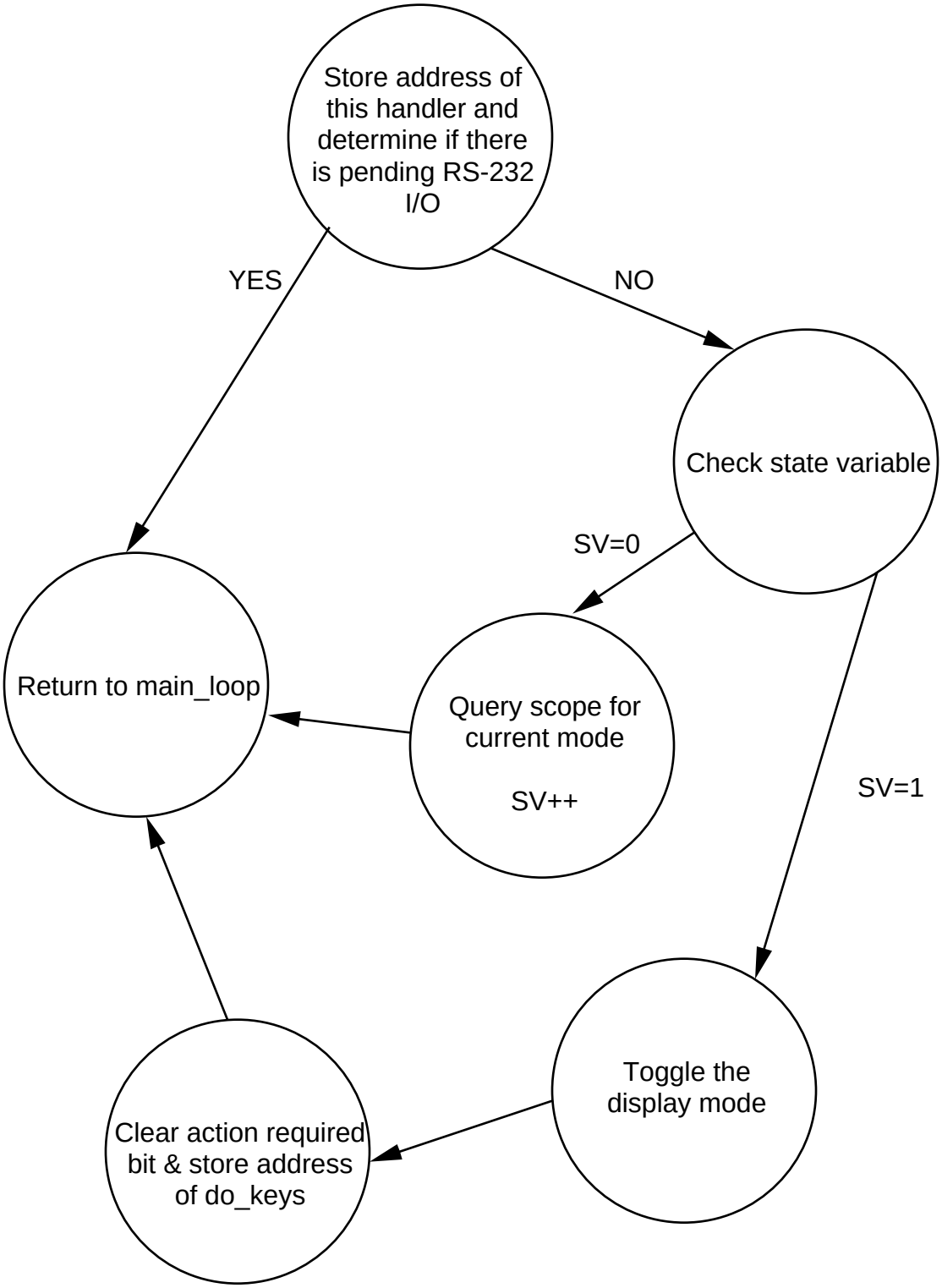
State transition diagram for the key_3 handler
(trigger knob toggle)



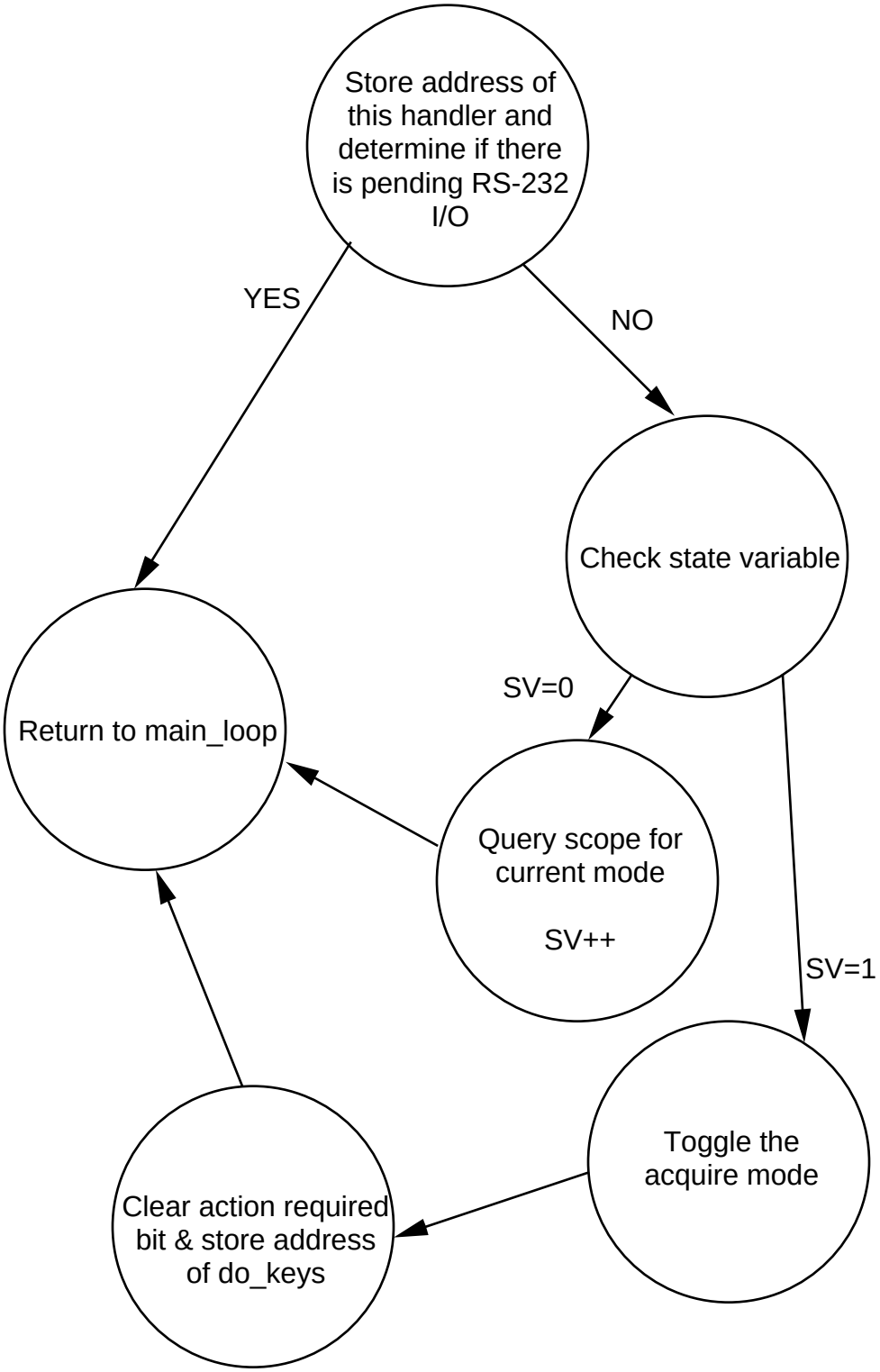
State transition diagram for the key_4 handler
(real time / persist button)



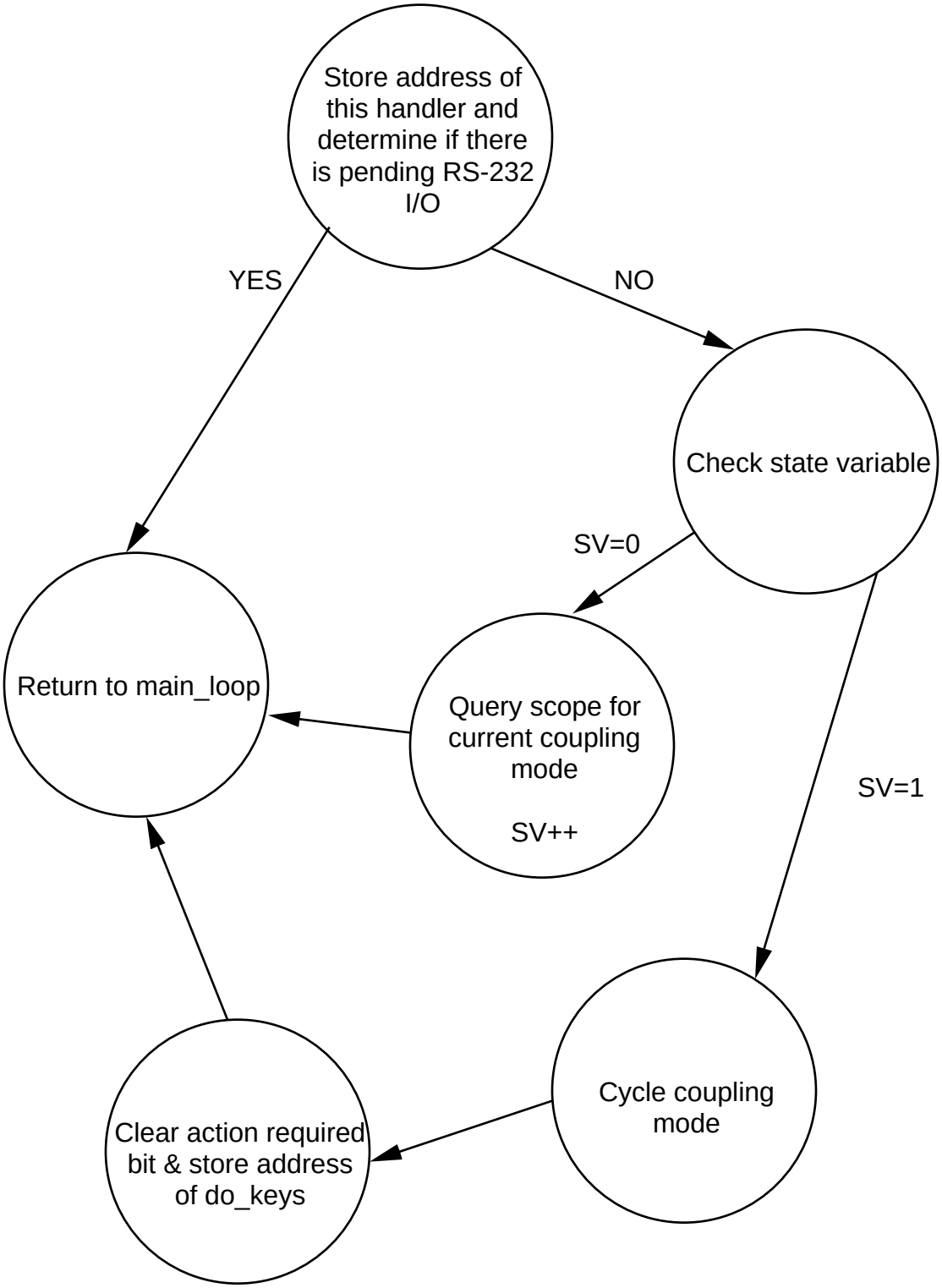
State transition diagram for the key_5 handler
(vectors / dots button)



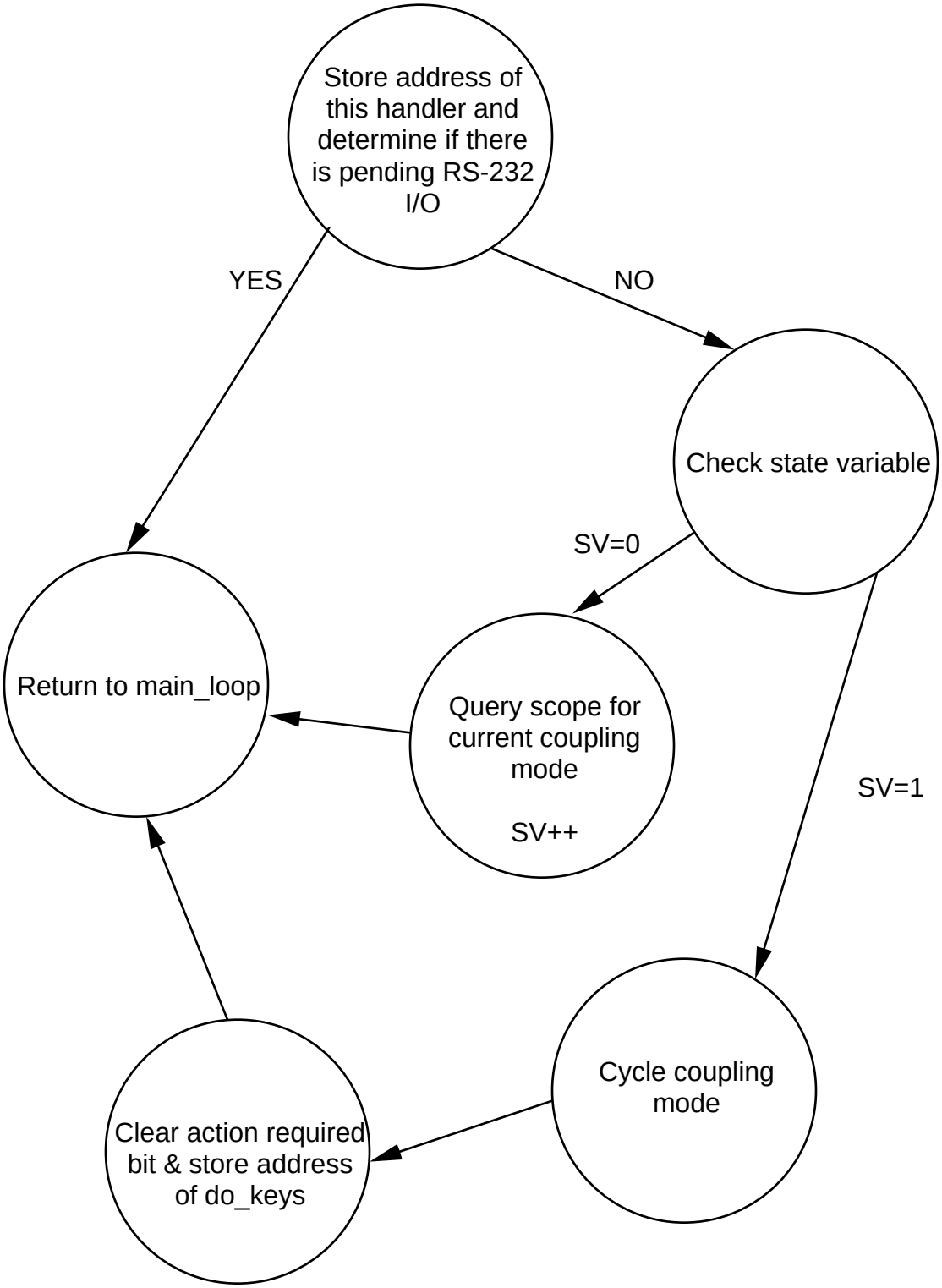
State transition diagram for the key_6 handler
(sample / average button)



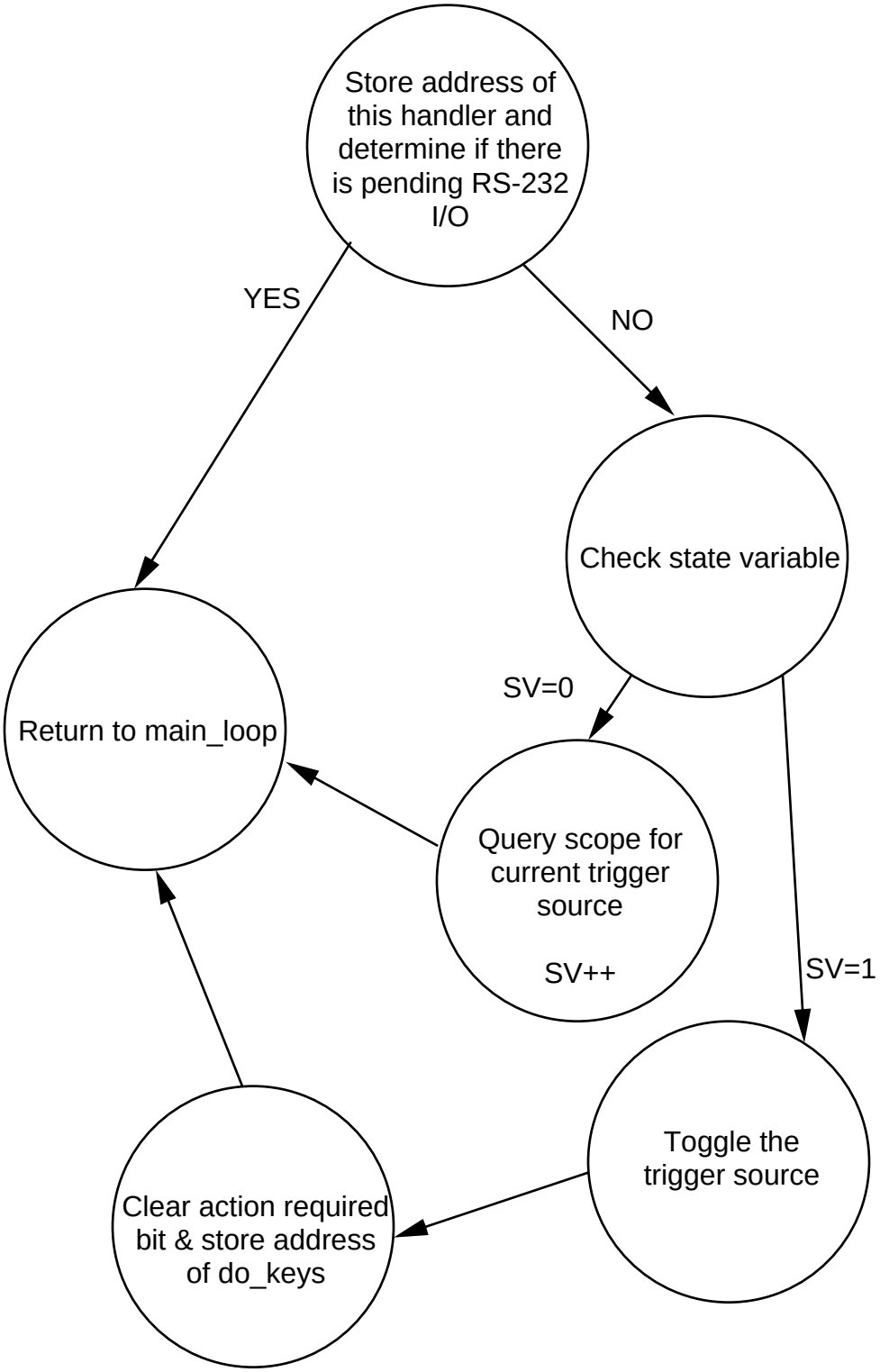
State transition diagram for the key_7 handler
(continuous / singleshot button)



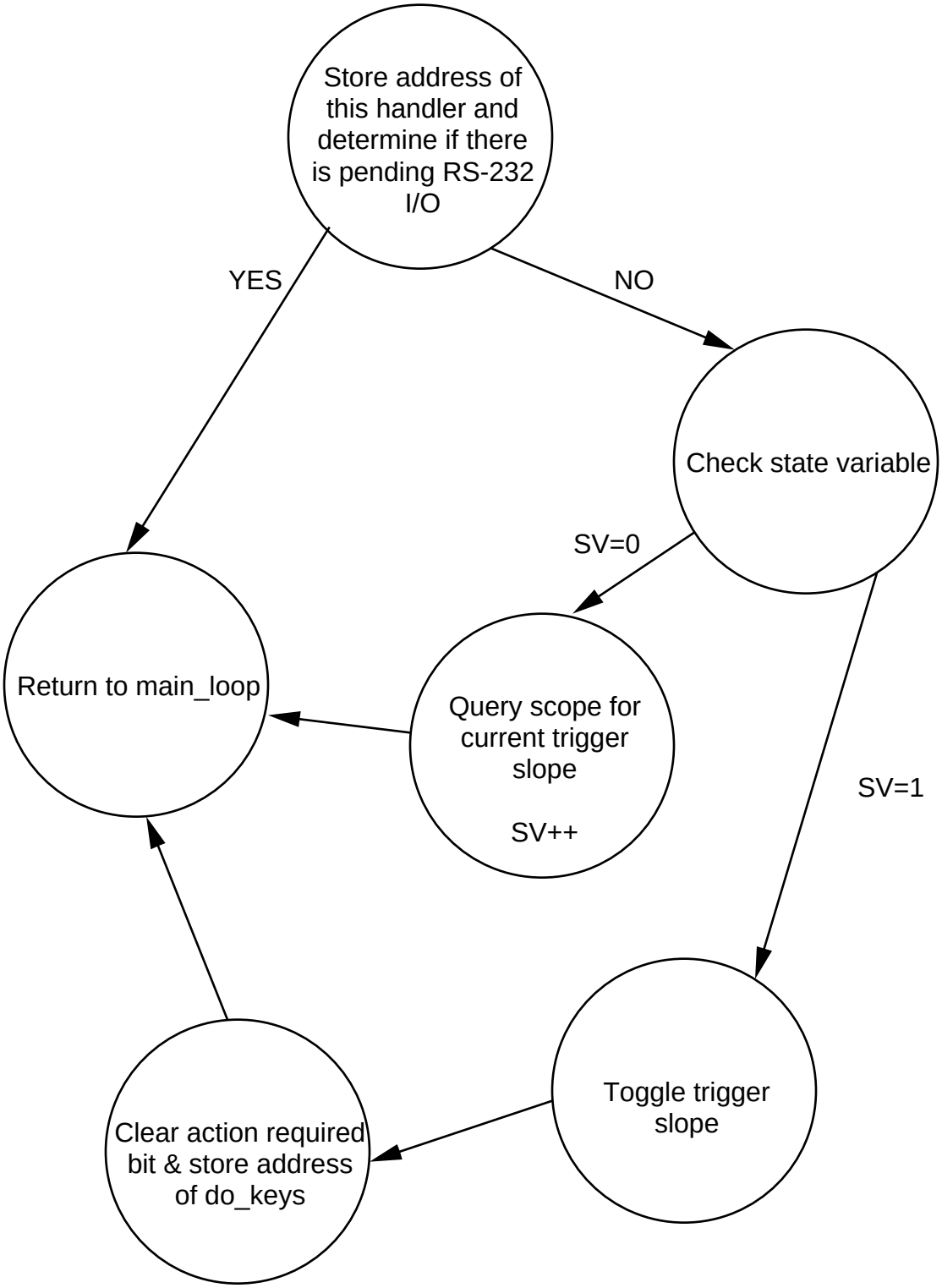
State transition diagram for the key_8 handler
(channel 1 coupling button)



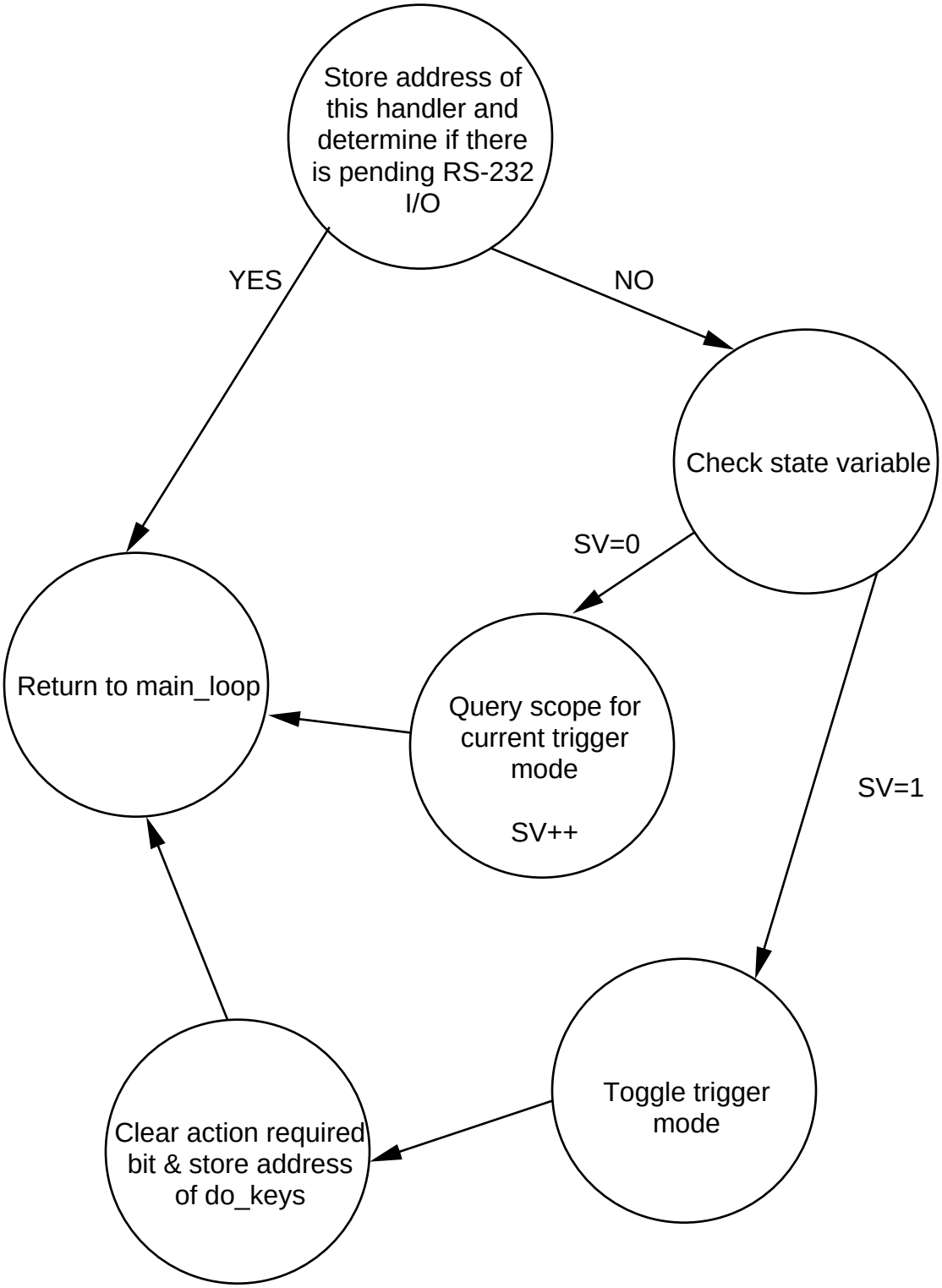
State transition diagram for the key_9 handler
(trigger source button)



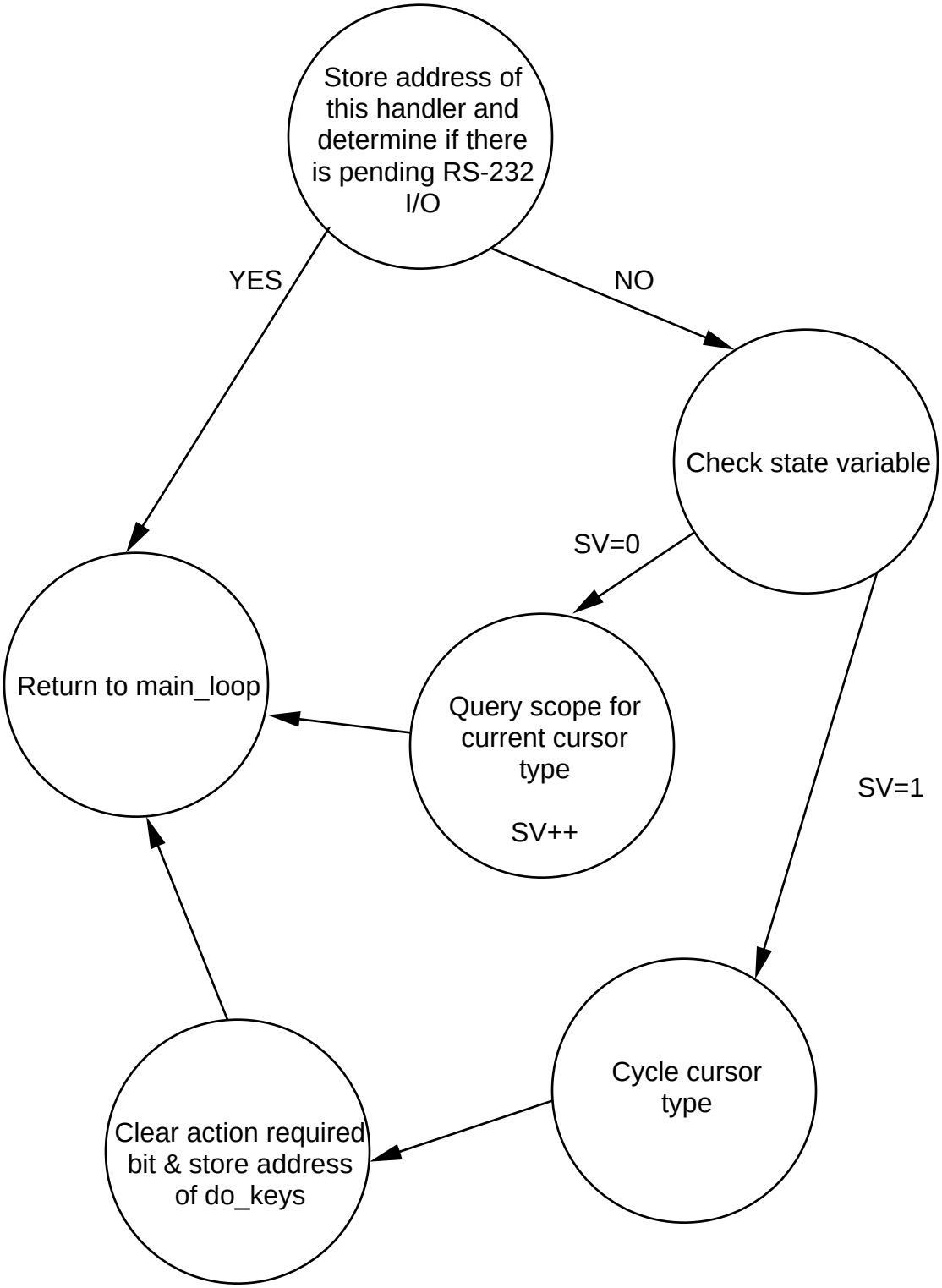
State transition diagram for the key_10 handler
(trigger slope button)



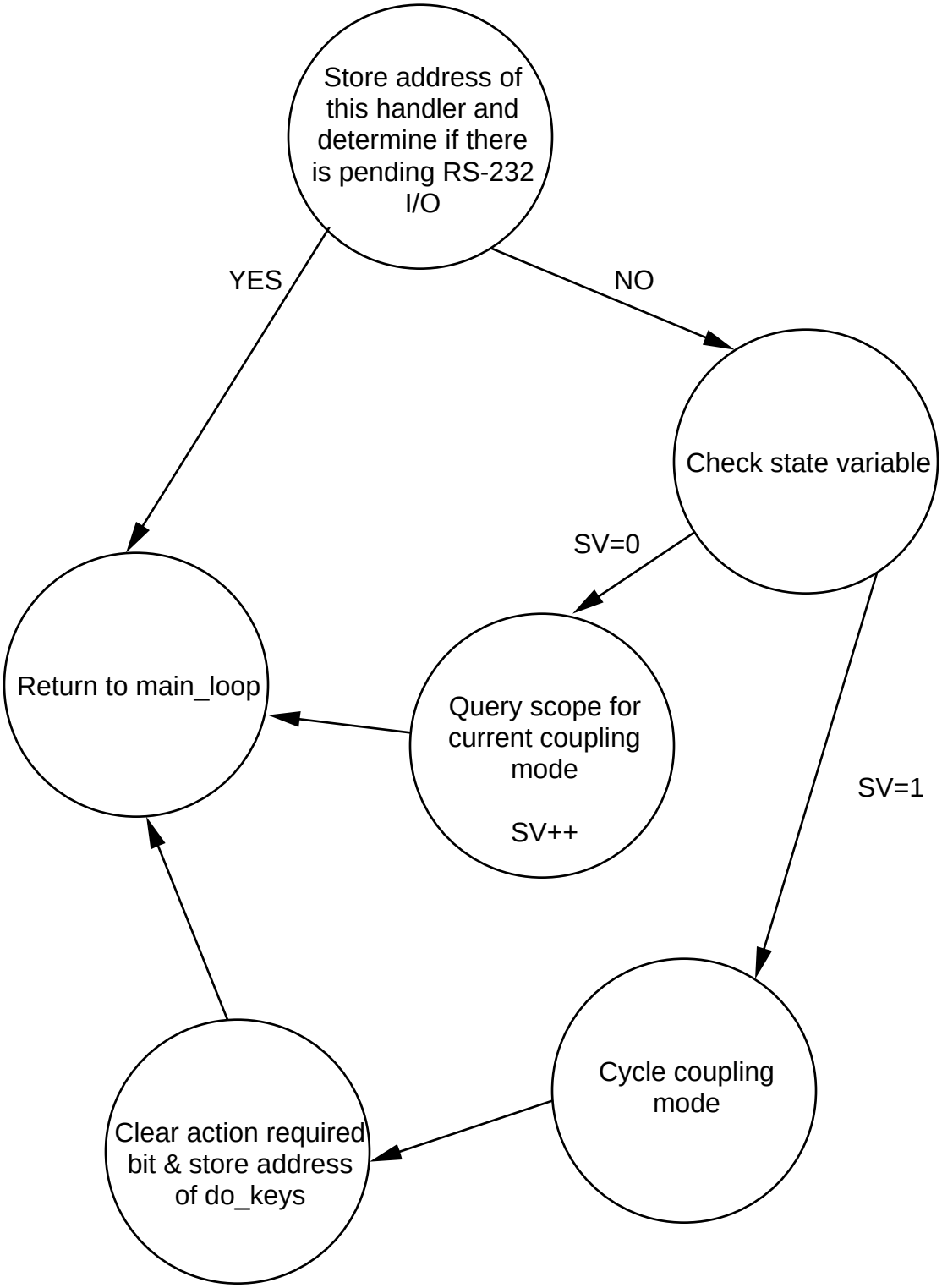
State transition diagram for the key_11 handler
(auto / normal trigger button)



State transition diagram for the key_14 handler
(cursor type select button)



State transition diagram for the key_15 handler
(channel 2 coupling button)



sCo-Pilot
Human Interface Enhancement

Project Timeline & Miscellaneous Notes

The sCo-Pilot was first envisioned on Friday, March 19, 1999 when I was using the THS series handheld scope in a surface mount application. I was holding two probes "chopstick" style on some SMD IC pins while having to carefully navigate the hierarchical menus of the THS scope. To my frustration, every time I would move my eyes from the circuit to the scope to read the menus, I would accidentally move the probes! I was using infinite persistence mode to capture time varying pulse widths and discovered that up to six keystrokes are required to reset the infinite persistence! Thus the sCo-Pilot idea was born.

Being a personal project, the only development time available was during nights and weekends. Since the deadline was only two months away, the sCo-Pilot had to be a personal "fast track" project. The timeline on the following page illustrates the development of the sCo-Pilot from start to finish.

The following "one liners" contain miscellaneous project notes and thoughts worthy of mention.

- The sCo-Pilot is a personal (home) project. Hence, many design parameters were dictated by home shop capabilities and budget. In other words, a custom molded plastic case or professional machining was out of the question. A prototype PCB was unlikely, but still may occur in the future.

- Considerable effort was expended in locating the "perfect" enclosure for the sCo-Pilot. Several sources were consulted. An off-the-shelf enclosure from Radio Shack ended-up to be the perfect match.

- The silkscreening of the front panel was professionally done. An expedite fee was paid to reduce the turn-time from two weeks to one week.

- Almost every new instruction on the CPU08 was used somewhere in the sCo-Pilot software.

- A prototype (surrogate) front panel was mocked-up to use for development purposes while the "real" front panel was out for silkscreening. See photo 3. This surrogate front panel was also used to verify ergonomics and component fit.

- A ZIF socket was installed on the sCo-Pilot prototype perfboard so that the 'GP20 could be easily shuttled between the programmer and the sCo-Pilot. See photo 1. The ZIF socket was removed prior to installation in the final enclosure.

- The sCo-Pilot's mounting brackets are custom made from aluminum sheet metal.

- Notable Reference material for this project:

  THS710A, THS720A, THS730A & THS720P TekScope Programmer Manual
  Tektronix P/N 070-9751-01

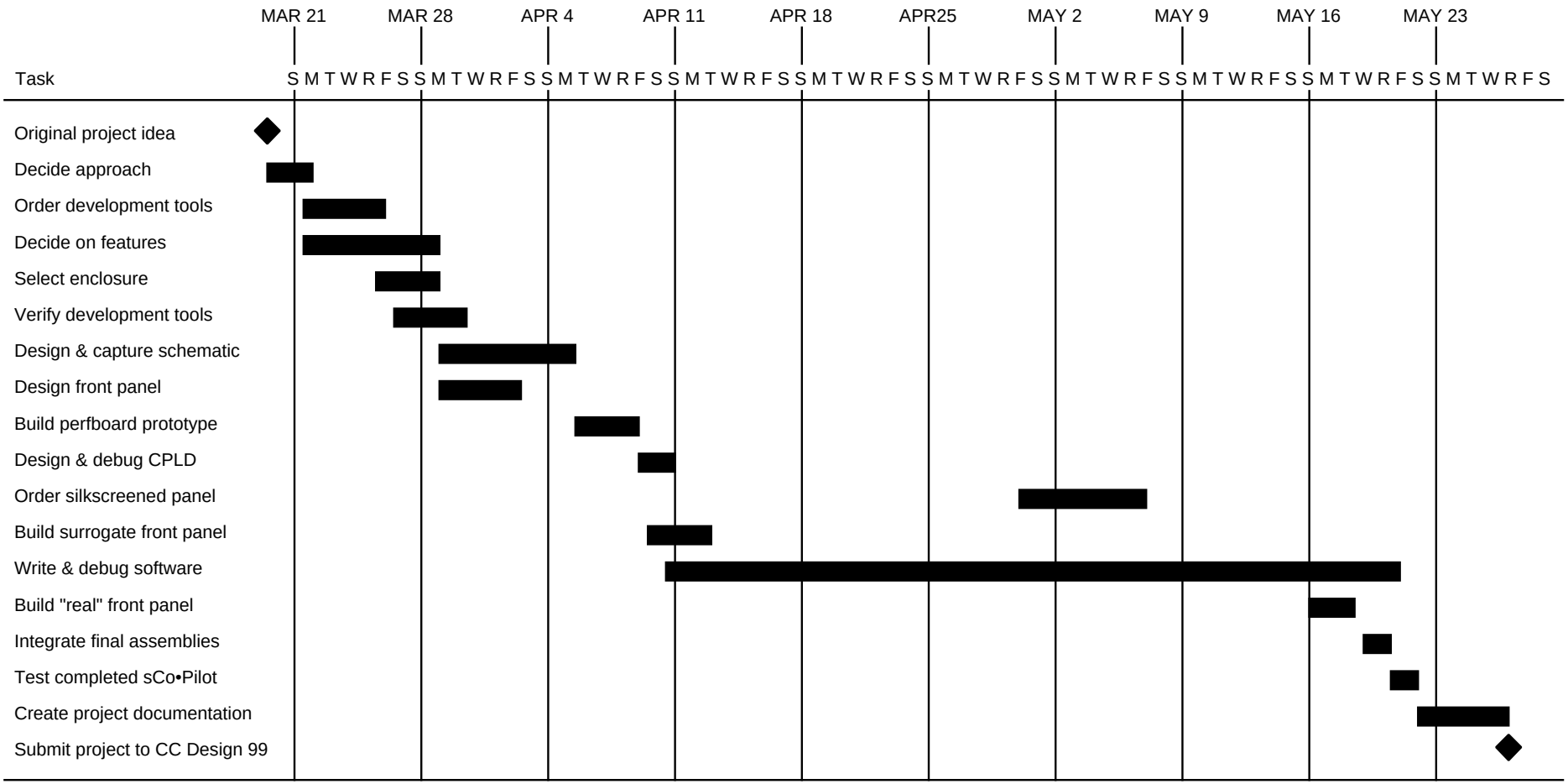
  CPU08 Central Processor Unit Reference Manual
  Motorola P/N CPU08RM/AD Rev. 1

  MC68HC908GP20 HCMOS Microcontroller Unit Advance Information
  Motorola P/N MC68HC908GP20/D Rev. 2.0

- The completed sCo-Pilot works as well as originally envisioned. No further enhancements are planned at this time.

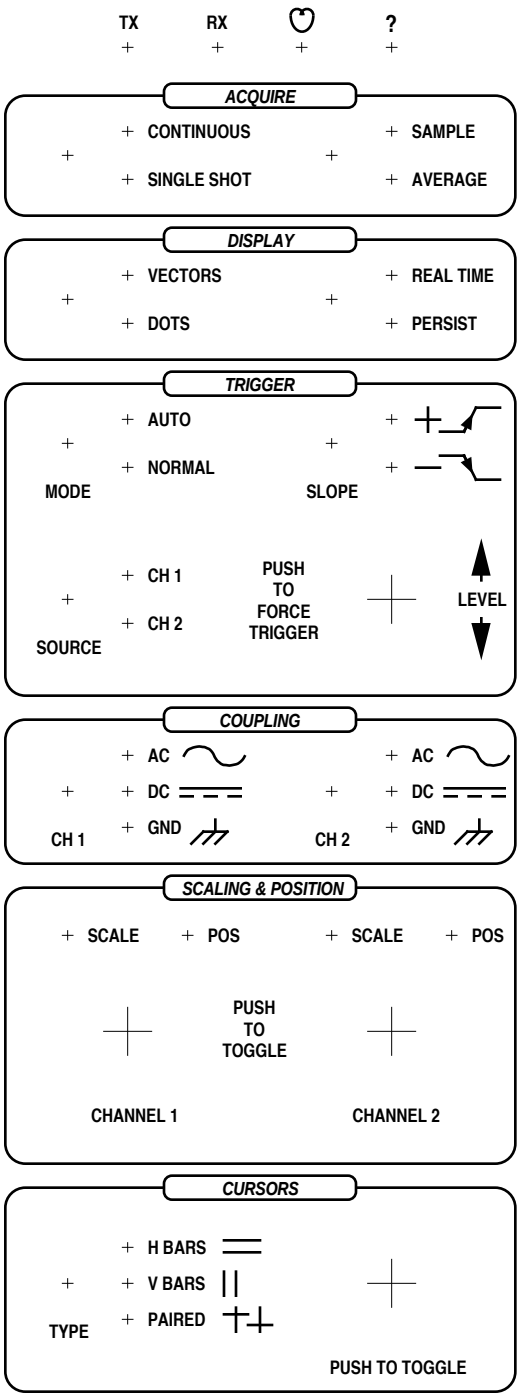
sCo•Pilot
Human Interface Enhancement

Project Timeline



FILM REQUIREMENTS:

100% SCALE
FILM POSITIVE
EMULSION UP




sCo Pilot
*Human Interface Enhancement*

| *sCo-Pilot silkscreen* |
| --- |
| *Page 1 of 1*<br>*5/2/99* |

sCo Pilot
Human Interface Enhancement

Project Photographs

**Photo 1**

The sCo-Pilot perfboard prototype.

Note ZIF socket for ‘GP20 and ground plane on perfboard.

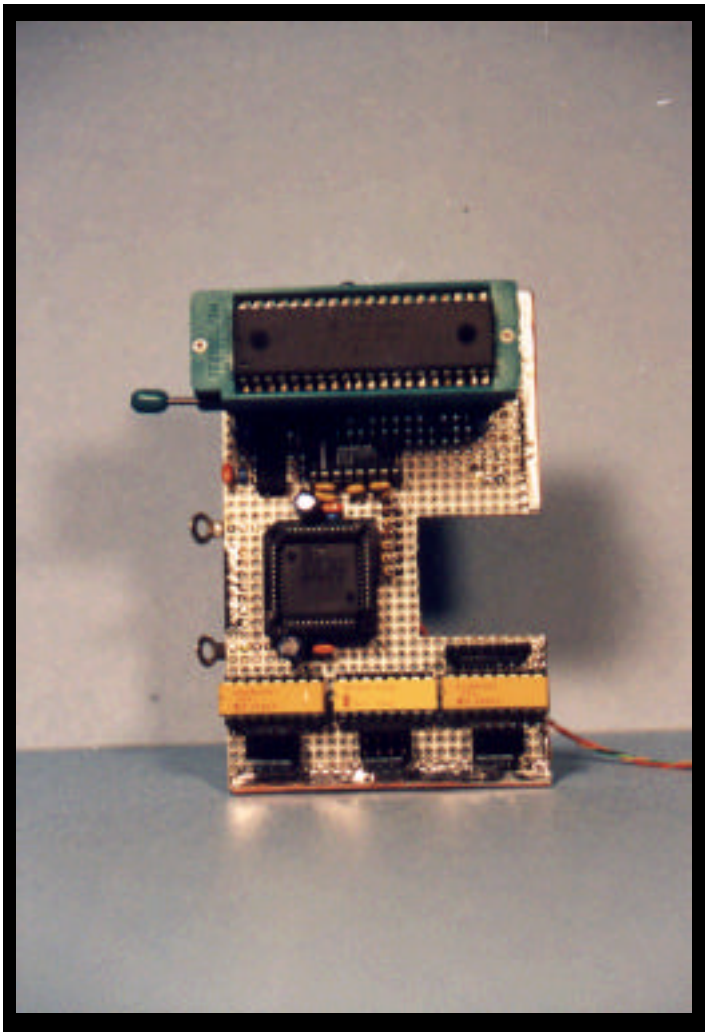


**Photo 2**

The “solder side” of the perfboard prototype.

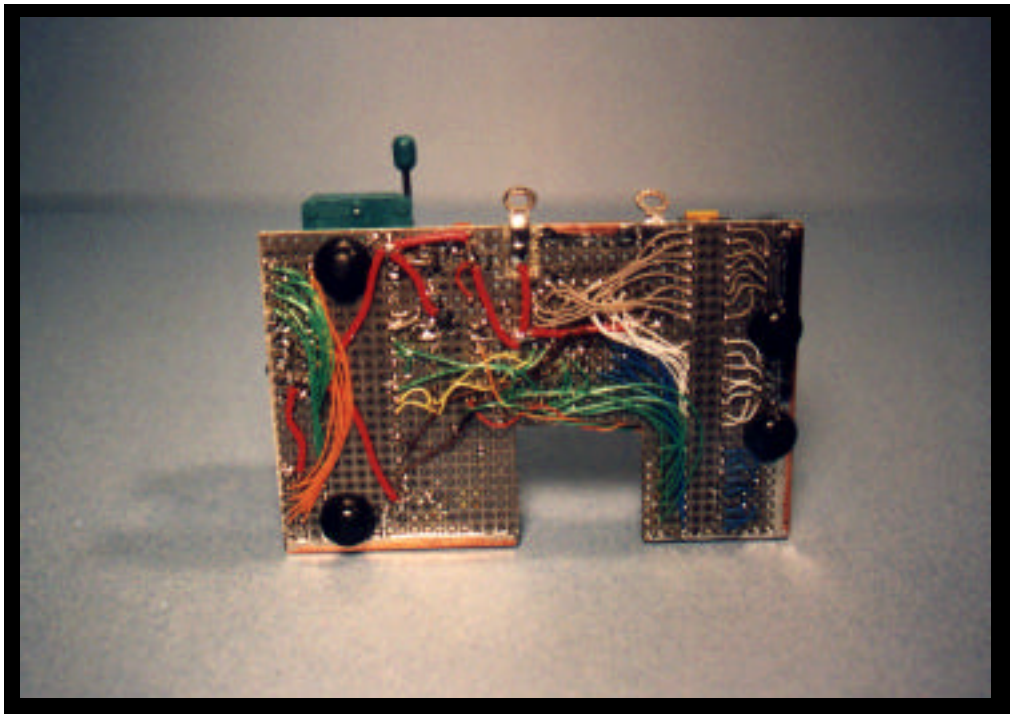


**Photo 3**

The sCo-Pilot development front panel, the prototype perfboard, and the THS720 scope.

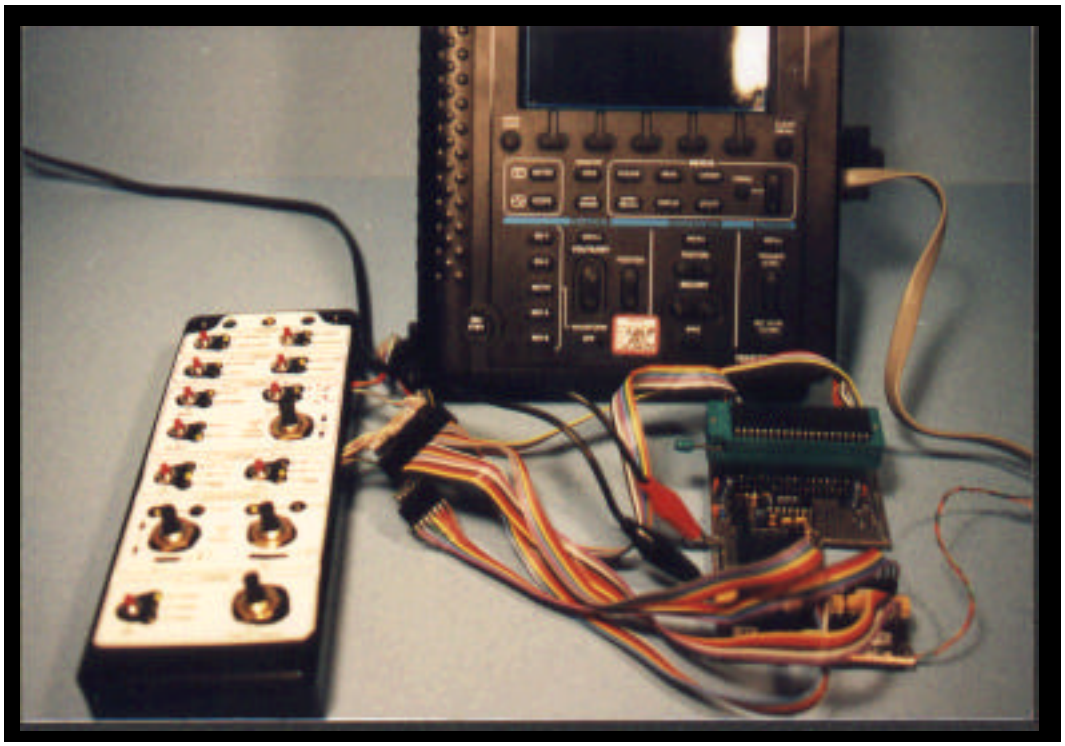


**Photo 4**

The sCo-Pilot prototype perfboard installed in the plastic chassis. Ready to accept the front panel.

Note voltage regulator & heatsink in lower left corner.

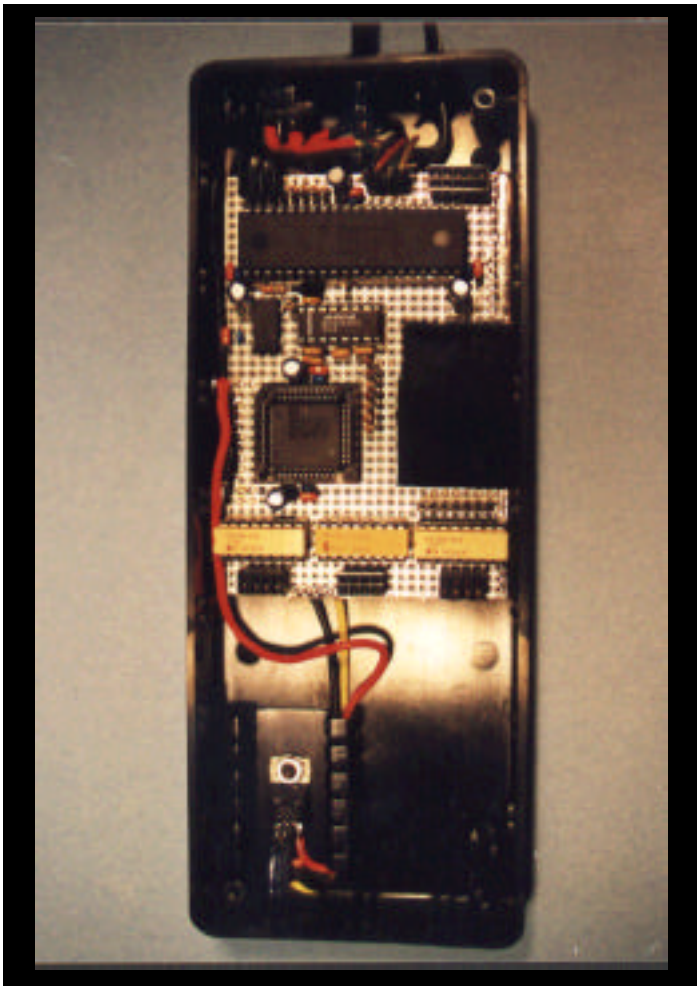


**Photo 5**

The rear view of the prototype front panel.

Note optical encoders.

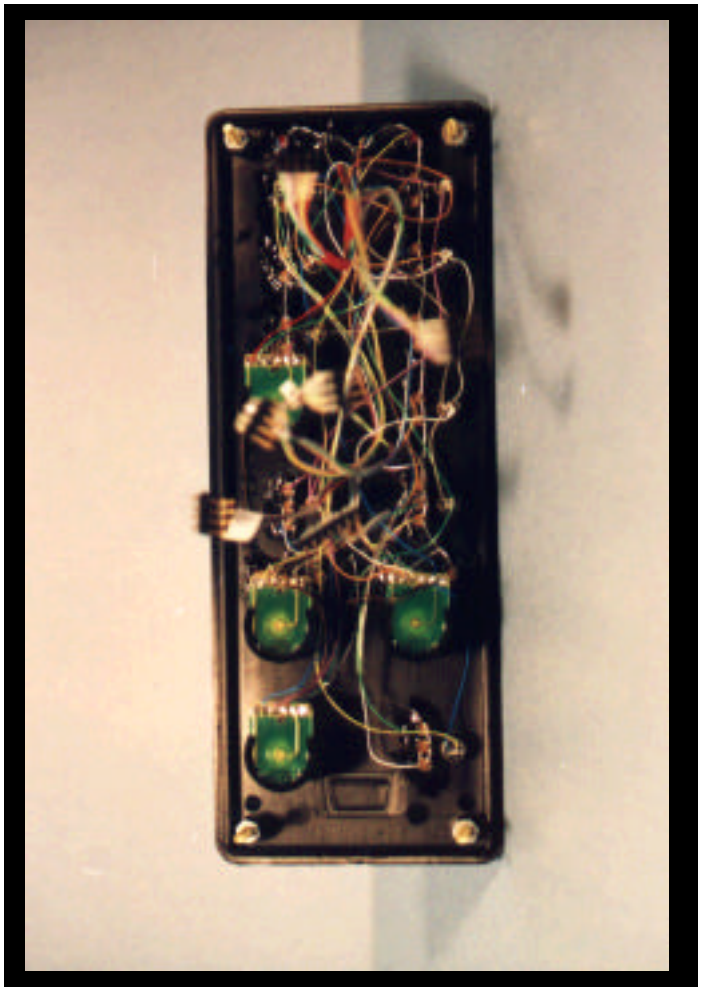



**Photo 6**

The assembled sCo-Pilot (finally!)

Note RF chokes on power and data cables and mounting brackets on the right side.

**Photo 7**

The completed sCo-Pilot.



**Photo 8**

A side view of the completed sCo-Pilot highlighting the mounting brackets.


**Photo 9**

The completed and working sCo-Pilot installed on the scope.


Software listing

The ‘GP20 software for the sCo-Pilot is listed in this section. The listing has been formatted for hardcopy presentation by adjusting tabs, spaces, and page breaks. In formatting the listing, no modifications to compileable source code have been made. Although the contents of the hardcopy listing will differ from that which is stored on the floppy disk, both sources will produce exactly the same compiled output.

```
;    Title: sCo-Pilot controller software
;   Author: Derek Matsunaga
;   Target: 68HC908GP20 (DIP40)
; Rev date: May 21, 1999
;Schematic: sCo-Pilot 5/21/99

$Include 'gp20regs.inc'                     ;  Include the predefined register macros.

;-------------------------------------------------------------------------------------------
;'GP20 Location definitions.
RAM_Start         EQU  $0040                ;  The start of RAM.
RAM_End           EQU  $0240                ;  The end of RAM.
ROM_Start         EQU  $B000                ;  The beginning of ROM (FLASH).
Vectortable       EQU  $FFDC                ;  Vector table.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;RAM variable definitions.
                  org     RAM_Start         ;  RAM variables.

timecount         RMB      !2               ;  A 16 bit location for keeping time.
LED_bank_A        RMB      !1               ;  Stores LED bank A status.
LED_bank_B        RMB      !1               ;  Stores LED bank B status.
LED_bank_C        RMB      !1               ;  Stores LED bank C status.
LED_bank_D        RMB      !1               ;  Stores LED bank D status.
keystate          RMB      !16              ;* Storage for key status.
knobstate         RMB      !8               ;* Storage for knob status.
row_count         RMB      !1               ;  A temporary variable for counting key rows.
col_count         RMB      !1               ;  A temporary variable for counting key columns.
key_image         RMB      !1               ;  A temporary variable for storing an image of the keyboard.
keytime           RMB      !1               ;  Stores the low byte of timecount for key debouncing.
knob_store        RMB      !1               ;  Stores the previously read knob state.
knob_now          RMB      !1               ;  Stores the current knob position.
knob_temp         RMB      !1               ;  A temporary location for quadrature processing.
knob_count        RMB      !1               ;  A temporary location for quadrature processing.
out_data_ptr      RMB      !2               ;  Stores the address of the 232 data to be sent.
inbuf_ptr         RMB      !1               ;  SCI input buffer pointer.
inbuf             RMB      !20              ;  The SCI input buffer.
waiting           RMB      !1               ;  A flag used to indicate that the sCo-Pilot is waiting for data.
temp_state        RMB      !1               ;  A temporary state variable used for key processing.
key_vector        RMB      !2               ;  Stores the address of the currently operating key service routine.
state_inq         RMB      !1               ;  A state variable set to request getting scope state.
fixed_ptr         RMB      !2               ;  A pointer to fixed point data.
BCD_ones          RMB      !1               ;* A temporary storage location for BCD tens & ones digits.
BCD_tenths        RMB      !1               ;* A temporary storage location for BCD tenths & hundredths digits.
BCD_source        RMB      !9               ;* A storage location for converted fixed-point BCD values.
BCD_inc           RMB      !9               ;  A storage location for the BCD increment value.
BCD_dest          RMB      !9               ;  A storage location for the result of fixed-point BCD math.
BCD_exp           RMB      !1               ;  A temporary storage location for BCD exponent conversions.
BCD_DP            RMB      !1               ;  A temporary pointer to the decimal point.
nibble_count      RMB      !1               ;  A counter used for BCD nibble shifting.
shift_count       RMB      !1               ;  A counter used to count nibble shift operations.
borrow            RMB      !1               ;  A storage place for the carry bit.
nib_source        RMB      !1               ;  A storage location used to pass nibbles to the subtract routine.
nib_sub           RMB      !1               ;  A storage location used to pass nibbles to the subtract routine.
curs_temp         RMB      !1               ;  A temporary storage location for cursor information.
curs_chan         RMB      !1               ;  A temporary storage location for the cursor channel.
curs_active       RMB      !1               ;  Stores the active cursor.
knob_inc_ptr      RMB      !8               ;  Points to an acceleration value in a lookup table.
knob_inc_count    RMB      !8               ;  The number of times a knob function is to executed based on acceleration.
knob_accel        RMB      !8               ;  Used to store the time of initial acceleration.
accel_time        RMB      !1               ;  Stores the knob acceleration time.
accel_temp        RMB      !1               ;  A temporary location to store a copy of acceleration data.
idle_count        RMB      !2               ;  A 16 bit value to count inactivity time.
idle_time         RMB      !1               ;  Stores the low timer byte when update_idle is called.
wait_time         RMB      !1               ;  Stores the low timer byte when check_waiting is called.
wait_count        RMB      !2               ;  A 16 bit value to count the time spent waiting for data.

*RAM variables highlighted with (*) must retain the same relative positions.  All others can move.
```

```
                org     $0100           ;  The upper 256 bytes of RAM.
float_buf       RMB     !20             ;  A storage location for ASCII floating point data to scope.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;Program start out of reset.
                org     ROM_Start       ;  The beginning of ROM (FLASH).
reset:          ldhx    #RAM_start      ;  Point H:X at the beginning of RAM.
clear_RAM:      clr     ,x              ;  Clear RAM location H:X.
                aix     #$0001          ;  Increment H:X.
                cphx    #RAM_End        ;  Test for end of RAM.
                bne     clear_RAM       ;  Loop until all RAM has been zeroed.

                rsp                     ;  Reset stack pointer to top of RAM ($FF).
                mov     #$01,CONFIG1    ;  Disable COP.
                sei                     ;  Disable interrupts.
                jsr     init_SCI        ;  Initialize the SCI for scope communication.
                jsr     init_SPI        ;  Initialize the SPI for LED I/O expander (PLD).
                jsr     init_TIM        ;  Initialize the TIM for timekeeping.
                cli                     ;  Enable interrupts.

                mov     #$ff,DDRC       ;  Configure all of PORTC to be outputs.
                mov     #$00,DDRA       ;  Put PORTA into high Z mode (inputs).
                mov     #$00,PTA        ;  Initialize all of PORTA low.
                mov     #$00,PTB        ;  Make all PORTB pins inputs (for knobs).
                mov     PTB,knob_store  ;  Initialize knob_store with current knob settings.
                ldhx    #do_process     ;  Load the address of the key processing routine.
                sthx    key_vector      ;  Store it in key_vector.
                clrh                    ;  Clear H for IX1 mode.

                ldhx    #msg_clearbuf   ;  Point H:X at the clear buffer message.
                sthx    out_data_ptr    ;  Queue it to be sent to the scope.

                jsr     poweron_lights1 ;  Go through the power-on light show to verify LEDs.
                jsr     poweron_lights1 ;  Go through the power-on light show to verify LEDs.
                jsr     poweron_lights2 ;  Go through the power-on light show to verify LEDs.
                jsr     poweron_lights2 ;  Go through the power-on light show to verify LEDs.

                mov     #$FF,LED_bank_A ;  Turn-off all bank_A LEDs.
                mov     #$FF,LED_bank_B ;  Turn-off all bank_B LEDs.
                mov     #$F5,LED_bank_C ;  Initialize bank_C LEDs.
                mov     #$FF,LED_bank_D ;  Turn-off all bank_D LEDs.

                jsr     request_status  ;  Request initial status so that LEDs may be set to indicate scope status.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The main_loop operates continuously.  The nominal loop time is about 270us, which enables the
;incoming 9600BPS serial data to be polled rather than interrupt driven.  Because each main_loop
;subroutine is a "state machine", the calling order is irrelevant and it is not possible
;for the CPU to get "locked-up" in a given state machine.

main_loop:      bclr    1,PTC           ;  Set loop time pin.
                bset    1,PTC           ;  Clear loop time pin.
                jsr     update_LEDs     ;  Send the LED data to the IO expander.
                jsr     read_keys       ;  Read the key matrix.
                jsr     debounce_keys   ;  Debounce any pressed keys.
                jsr     do_keys         ;  Perform key and knob actions if necessary.
                jsr     read_knobs      ;  Read the knobs.
                jsr     heartbeat       ;  Update the heartbeat.
                jsr     out232          ;  Send pending RS-232 output (if any).
                jsr     in232           ;  See if there is RS-232 input available.
                jsr     update_accel    ;  Update the knob acceleration values.
                jsr     update_idle     ;  If sCo-Pilot is idle for a certain time, update status.
                jsr     check_waiting   ;  Determine if the sCo-Pilot has been waiting for data for too long.
                bra     main_loop       ;  Continue to loop.
;-------------------------------------------------------------------------------------------
```

```
msg_clearbuf:     db       !10,!10,!10,!10,!10,0      ;  Line feeds flush the scope's input buffer.


;-------------------------------------------------------------------------------------------
;The poweron_lights1 routine displays a "light show" at power-on to verify the functionality of
;all sCo-Pilot LEDs.  The LEDs are lit in the combinations & sequences determined by the lookup
;table light_order1.  In the case of light show 1, the LEDs are "circulated" around the sCo-Pilot
;in a counter clockwise order.  The state is advanced every 50ms.  This subroutine trashes A, X,
;and H.  Because the physical LED locations and the software LED locations are different, it is
;most convenient to sequence the light show via a lookup table.  The power-on light show is the
;last sCo-Pilot routine to be implemented.  There is still plenty of program space left.

poweron_lights1:  clrx                      ;  Point X to the first value in the LUT.
                  clrh                      ;  Clear H for IX1 addressing.
poweron_loop1:    lda      light_order1,x   ;  Get the LED bank A value from the LUT.
                  sta      LED_bank_A       ;  Store it in the LED_bank_A location.
                  lda      light_order1+1,x ;  Get the LED bank B value from the LUT.
                  sta      LED_bank_B       ;  Store it in the LED_bank_B location.
                  lda      light_order1+2,x ;  Get the LED bank C value from the LUT.
                  sta      LED_bank_C       ;  Store it in the LED_bank_C location.
                  lda      light_order1+3,x ;  Get the LED bank D value from the LUT.
                  sta      LED_bank_D       ;  Store it in the LED_bank_D location.
                  jsr      update_LEDs      ;  Write the new values to the LED banks.
                  jsr      light_delay      ;  Delay.

                  incx                      ;  Point X to the next position in the LUT.
                  incx                      ;  Point X to the next position in the LUT.
                  incx                      ;  Point X to the next position in the LUT.
                  incx                      ;  Point X to the next position in the LUT.

                  cpx      #light_lut1_end  ;  See if the show is over.
                  bne      poweron_loop1    ;  Loop again if not.
                  rts
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The poweron_lights2 routine displays a "light show" at power on to verify the functionality of
;all sCo-Pilot LEDs.  The LEDs are lit in the combinations & sequences determined by the lookup
;table light_order2.  In the case of light show 2, a horizontal bar of LEDs appears to travel
;down the sCo-Pilot and then back up.  The state is advanced every 50ms.  This subroutine
;trashes A, X, and H.  Because the physical LED locations and the software LED locations are different, it is
;most convenient to sequence the light show via a lookup table.  The power-on light show is the
;last sCo-Pilot routine to be implemented.  There is still plenty of program space left.

poweron_lights2:  clrx                      ;  Point X to the first value in the LUT.
                  clrh                      ;  Clear H for IX1 addressing.
poweron_loop2:    lda      light_order2,x   ;  Get the LED bank A value from the LUT.
                  sta      LED_bank_A       ;  Store it in the LED_bank_A location.
                  lda      light_order2+1,x ;  Get the LED bank B value from the LUT.
                  sta      LED_bank_B       ;  Store it in the LED_bank_B location.
                  lda      light_order2+2,x ;  Get the LED bank C value from the LUT.
                  sta      LED_bank_C       ;  Store it in the LED_bank_C location.
                  lda      light_order2+3,x ;  Get the LED bank D value from the LUT.
                  sta      LED_bank_D       ;  Store it in the LED_bank_D location.
                  jsr      update_LEDs      ;  Write the new values to the LED banks.
                  jsr      light_delay      ;  Delay.
                  incx                      ;  Point X to the next position in the LUT.
                  incx                      ;  Point X to the next position in the LUT.
                  incx                      ;  Point X to the next position in the LUT.
                  incx                      ;  Point X to the next position in the LUT.

                  cpx      #light_lut2_end  ;  See if the downward progression is over.
                  bne      poweron_loop2    ;  Loop again if not.
;Make the LEDs travel back up.
                  decx                      ;  Point X to the previous position in the LUT.
                  decx                      ;  Point X to the previous position in the LUT.
                  decx                      ;  Point X to the previous position in the LUT.
                  decx                      ;  Point X to the previous position in the LUT.
```

```
poweron_loop3:    lda     light_order2,x    ;  Get the LED bank A value from the LUT.
                  sta     LED_bank_A        ;  Store it in the LED_bank_A location.
                  lda     light_order2+1,x  ;  Get the LED bank B value from the LUT.
                  sta     LED_bank_B        ;  Store it in the LED_bank_B location.
                  lda     light_order2+2,x  ;  Get the LED bank C value from the LUT.
                  sta     LED_bank_C        ;  Store it in the LED_bank_C location.
                  lda     light_order2+3,x  ;  Get the LED bank D value from the LUT.
                  sta     LED_bank_D        ;  Store it in the LED_bank_D location.
                  jsr     update_LEDs       ;  Write the new values to the LED banks.
                  jsr     light_delay       ;  Delay.
                  decx                      ;  Point X to the previous position in the LUT.
                  decx                      ;  Point X to the previous position in the LUT.
                  decx                      ;  Point X to the previous position in the LUT.
                  decx                      ;  Point X to the previous position in the LUT.
                  bne     poweron_loop3     ;  Loop until the show is over.
                  rts
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;The following lookup tables sequence the LEDs into an aesthetically pleasing order for the
;power-on light show.

light_lut1_end    EQU     !108              ;  The number of bytes in the light show LUT.

light_order1:     db      $FE,$FF,$FF,$FF   ;  Continuous LED.
                  db      $FD,$FF,$FF,$FF   ;  Single shot LED.
                  db      $EF,$FF,$FF,$FF   ;  Vectors LED.
                  db      $DF,$FF,$FF,$FF   ;  Dots LED.
                  db      $FF,$F7,$FF,$FF   ;  Auto LED.
                  db      $FF,$FB,$FF,$FF   ;  Normal LED.
                  db      $FF,$EF,$FF,$FF   ;  Trigger CH1 LED.
                  db      $FF,$DF,$FF,$FF   ;  Trigger CH2 LED.
                  db      $FF,$BF,$FF,$FF   ;  CH1 AC LED.
                  db      $FF,$FF,$7F,$FF   ;  CH1 DC LED.
                  db      $FF,$FF,$DF,$FF   ;  CH1 GND LED.
                  db      $FF,$FF,$F7,$FF   ;  CH1 scale LED.
                  db      $FF,$FF,$FB,$FF   ;  CH1 position LED.
                  db      $FF,$FF,$FF,$FE   ;  Hbars LED.
                  db      $FF,$FF,$FF,$FD   ;  Vbars LED.
                  db      $FF,$FF,$FF,$FB   ;  Pbars LED.

                  db      $FF,$FF,$FE,$FF   ;  CH2 position LED.
                  db      $FF,$FF,$FD,$FF   ;  CH2 scale LED.
                  db      $FF,$FF,$EF,$FF   ;  CH2 GND LED.
                  db      $FF,$FF,$BF,$FF   ;  CH2 DC LED.
                  db      $FF,$7F,$FF,$FF   ;  CH2 AC LED.
                  db      $FF,$FE,$FF,$FF   ;  Slope - LED.
                  db      $FF,$FD,$FF,$FF   ;  Slope + LED.
                  db      $7F,$FF,$FF,$FF   ;  Persist LED.
                  db      $BF,$FF,$FF,$FF   ;  Real time LED.
                  db      $F7,$FF,$FF,$FF   ;  Average LED.
                  db      $FB,$FF,$FF,$FF   ;  Sample LED.

light_lut2_end    EQU     !60               ;  The number of bytes in the light show LUT.
light_order2:     db      $FA,$FF,$FF,$FF   ;  Continuous & sample LEDs.
                  db      $F5,$FF,$FF,$FF   ;  Singleshot & average LEDs.
                  db      $AF,$FF,$FF,$FF   ;  Vectors & real time LEDs.
                  db      $5F,$FF,$FF,$FF   ;  Dots & persist LEDs.
                  db      $FF,$F5,$FF,$FF   ;  Auto & slope + LEDs.
                  db      $FF,$FA,$FF,$FF   ;  Normal & slope - LEDs.
                  db      $FF,$EF,$FF,$FF   ;  Trigger CH1 LED.
                  db      $FF,$DF,$FF,$FF   ;  Trigger CH2 LED.
                  db      $FF,$3F,$FF,$FF   ;  CH1 AC & CH2 AC LEDs.
                  db      $FF,$FF,$3F,$FF   ;  CH1 DC & CH2 DC LEDs.
                  db      $FF,$FF,$CF,$FF   ;  CH1 GND & CH2 GND LEDs.
                  db      $FF,$FF,$F0,$FF   ;  CH1 scale & pos & CH2 scale & pos LEDs.
                  db      $FF,$FF,$FF,$FE   ;  Hbars LED.
                  db      $FF,$FF,$FF,$FD   ;  Vbars LED.
                  db      $FF,$FF,$FF,$FB   ;  Pbars LED.
```

```
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The light_delay routine is a simple delay during which nothing is done.  This delay is used
;during the power-up light show to provide delay for the LEDs.  A is destroyed by this
;routine.  The main_loop is not running when this routine is called.

light_delay:      lda     #!25             ;  Initialize A for 50ms countdown (every-other timecount).

light_delay_loop: brclr   0,timecount+1,$  ;  Loop until the low bit of timecount is 1.
                  dbnza   light_delay_cont ;  Continue to count if the countdown has not expired.
                  rts                      ;  Return to caller after 200ms.
light_delay_cont: brset   0,timecount+1,$  ;  Loop until the low bit of timecount is 0.
                  bra     light_delay_loop ;  Loop until the countdown is complete.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The check_waiting routine checks for 3 seconds of continuous wait time.  If the 3 second wait
;time is exceeded, the temp_state variable is reset to zero and the msg_clearbuf is sent to
;clear the scope's buffer.  Calling this routine will prevent the sCo-Pilot from getting
;"hung up" while waiting for data if the communication link becomes broken.

check_waiting:    brclr   0,waiting,nowait ;  Get-out if the scope is not waiting.
                  psha                     ;  Save A.
                  lda     wait_time        ;  Get the low byte of the idle_time.
                  eor     timecount+1      ;  See if it has changed since last time.
                  beq     wait_done        ;  Exit if 1ms has not elapsed since last time called.

                  lda     timecount+1      ;  Get the low byte of timecount.
                  sta     wait_time        ;  "Stamp" the entry into this routine.
                  pshx                     ;  Save X.
                  pshh                     ;  Save H.
                  ldhx    wait_count       ;  Get the 16 bit wait_count value.
                  cphx    #!3000           ;  See if 3 seconds (3000ms) have elapsed.
                  bne     wait_inc         ;  Increment the idle_count if not.
                  clr     wait_count+1     ;  Reset the low byte of wait_count.
                  clr     wait_count       ;  Reset the high byte of wait_count.
                  ldhx    #msg_clearbuf    ;  Point H:X to the clear buffer message.
                  sthx    out_data_ptr     ;  Put it into the output queue.
                  clr     waiting          ;  Reset the waiting flag so the state machine can re-execute.
                  clr     temp_state       ;  Reset the temporary state variable.
                  pulh                     ;  Restore H.
                  pulx                     ;  Restore X.
wait_done:        pula                     ;  Restore A.
                  rts                      ;  Return to caller.

wait_inc:         aix     #0001            ;  Add one to the 16 bit wait_count value.
                  sthx    wait_count       ;  Store it back to wait_count.
                  pulh                     ;  Restore H.
                  pulx                     ;  Restore X.
                  bra     wait_done        ;  Return to caller.

nowait:           clr     wait_count+1     ;  Reset the low byte of wait_count.
                  clr     wait_count       ;  Reset the high byte of wait_count.
                  rts                      ;  Return to caller.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The update_idle routine checks for 30 seconds of continuous idle time (i.e. no buttons pressed
;or knobs turned).  If the idle time is exceeded, the update_idle routine proceeds to request
;status information from the scope (re-synchronize).  The parameters requested by the sCo-pilot
;during update_idle are identical to those requested on power-up.

update_idle:      psha                     ;  Save A.
                  lda     idle_time        ;  Get the low byte of the idle_time.
                  eor     timecount+1      ;  See if it has changed since last time.
                  beq     idle_done        ;  Exit if 1ms has not elapsed since last time called.
```

```
                lda     timecount+1       ;  Get the low byte of timecount.
                sta     idle_time         ;  "Stamp" the entry into this routine.
                pshx                      ;  Save X.
                pshh                      ;  Save H.
                ldhx    idle_count        ;  Get the 16 bit idle_count value.
                cphx    #$7530            ;  See if 30 seconds (30,000ms) have elapsed.
                bne     idle_inc          ;  Increment the idle_count if not.
                clr     idle_count+1      ;  Reset the low byte of idle_count.
                clr     idle_count        ;  Reset the high byte of idle_count.
                bsr     request_status    ;  Make the update status request (same as on power-up).
                pulh                      ;  Restore H.
                pulx                      ;  Restore X.
idle_done:      pula                      ;  Restore A.
                rts                       ;  Return to caller.
idle_inc:       aix     #0001             ;  Add one to the 16 bit idle_count value.
                sthx    idle_count        ;  Store it back to idle_count.
                pulh                      ;  Restore H.
                pulx                      ;  Restore X.
                bra     idle_done         ;  Return to caller.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The request_status routine sets the action required bit for all key and knob handlers whose
;status is stored locally.  Then, state_inq is set to indicate to the key and knob handlers
;that it is a status request only.

request_status:   bset    5,keystate+0      ;  Request active cursor data.
                  bset    5,keystate+4      ;  Request real-time/persist data.
                  bset    5,keystate+5      ;  Request vectors/dots data.
                  bset    5,keystate+6      ;  Request sampling/average data.
                  bset    5,keystate+7      ;  Request continuous/single shot data.
                  bset    5,keystate+8      ;  Request channel 1 coupling data.
                  bset    5,keystate+9      ;  Request trigger source data.
                  bset    5,keystate+!10    ;  Request trigger slope data.
                  bset    5,keystate+!11    ;  Request trigger mode data.
                  bset    5,keystate+!14    ;  Request cursor type data.
                  bset    5,keystate+!15    ;  Request channel 2 coupling data.
                  mov     #$ff,state_inq    ;  Indicate to the state machines that this is inquiry only.
                  rts                       ;  Return to caller.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The update_accel routine decrements every element in the knob_accel array until each element is
;zero.  The decrement occurs every millisecond.

update_accel:     psha                      ;  Save A.
                  lda     accel_time        ;  Get the initial acceleration time.
                  eor     timecount+1       ;  See if 1ms has elapsed since time stamp.
                  beq     accel_done        ;  Exit if 1ms has not elapsed since last time called.

                  pshx                      ;  Save X.
                  pshh                      ;  Save H.
                  clrh                      ;  Clear H for IX1 addressing.

                  ldx     #$09              ;  Initialize X for 8 acceleration values.
accel_loop:       lda     knob_accel-1,x    ;  Get the acceleration time for this knob.
                  beq     next_accel        ;  Move-on if this value is already zero.
                  dec     knob_accel-1,x    ;  Decrement the acceleration time.
next_accel:       dbnzx   accel_loop        ;  Loop until entire array has been updated.

                  mov     timecount+1,accel_time     ;  Update the call time.
                  pulh                      ;  Restore H.
                  pulx                      ;  Restore X.
accel_done:       pula                      ;  Restore A.
                  rts                       ;  Return to main_loop.
;-------------------------------------------------------------------------------------------
```

```
;-------------------------------------------------------------------------------------------
;The read_knobs routine looks for changes at PORTB.  Each of the four knob outputs a two
;bit quadrature pattern.  This pattern is stored in knob_state for comparison purposes.
;If a change in knob_state is detected, the appropriate action required bit in key_state
;will be set for the handlers to process.  The quadrature decode is done by means of a
;16 element lookup table.  The address of the lookup table is formed by concatenating
;the previous two quadrature bits and the present two quadrature bits.  A zero in the
;table represents a clockwise turn and a one represents a counterclockwise turn. An
;example of the method is illustrated below:

;    previous position   current position    rotation     table address        data
;           00         +       01              CCW      0001 (binary)  $01     $01
;           10         +       00              CCW      1000 (binary)  $08     $01
;           11         +       10              CCW      1110 (binary)  $0e     $01
;           01         +       11               CW      0111 (binary)  $07     $01
;           00         +       10               CW      0010 (binary)  $02     $00
;           10         +       11               CW      1011 (binary)  $0b     $00
;           11         +       01               CW      1101 (binary)  $0d     $00
;           01         +       00               CW      0100 (binary)  $04     $00

;Note that some elements in the table will never occur and are therefore never used.
;The unused elements in the LUT have data $FF for clarity.

read_knobs:       psha                      ;  Save A.
                  lda     PTB               ;  Get the current knob values.
                  cmp     knob_store        ;  See if they are the same as last time.
                  bne     knob_change       ;  If something has changed, start checking the knobs.
                  pula                      ;  Restore A.
                  rts                       ;  Return to main_loop if knobs have not changed.

knob_change:      clr     idle_count+1      ;  Reset the low byte of idle_count since a knob has been turned.
                  clr     idle_count        ;  Reset the high byte of idle_count since a knob has been turned.
                  mov     PTB,knob_now      ;  Copy the current knob state into memory.
                  pshx                      ;  Save X.
                  pshh                      ;  Save H.
                  clrh                      ;  Clear H for IX1 addressing.
                  mov     #$03,knob_count   ;  Initialize a loop counter for 4 knobs.
knob_loop:        bsr     check_knob        ;  Check the current knob for change.
                  lsr     knob_now          ;  Position the next knob in the lower 2 bits.
                  lsr     knob_now          ;  Position the next knob in the lower 2 bits.
                  lsr     knob_store        ;  Position the next knob in the lower 2 bits.
                  lsr     knob_store        ;  Position the next knob in the lower 2 bits.
                  dec     knob_count        ;  Decrement the loop counter.
                  bpl     knob_loop         ;  Continues to loop until all knobs have been checked.
                  pulh                      ;  Restore H.
                  pulx                      ;  Restore X.
                  pula                      ;  Restore A.
                  mov     PTB,knob_store    ;  Store the current knob state.
                  rts                       ;  Return to main_loop.

check_knob:       lda     knob_now          ;  Get the current knob value.
                  eor     knob_store        ;  See what changed.
                  and     #$03              ;  Remove all but the low bits.
                  beq     knob_same         ;  Check the next knob if this one didn't change.
                  lda     knob_store        ;  Get the previous knob value.
                  and     #$03              ;  Remove all but low bits.
                  lsla                      ;  Move the data into bits 2 and 3.
                  lsla
                  sta     knob_temp         ;  Store it.
                  lda     knob_now          ;  Get the current knob value.
                  and     #$03              ;  Remove all but knob 0.
                  ora     knob_temp         ;  Combine to form LUT address.
                  tax                       ;  Put the LUT address in X.
                  lda     knob_LUT,x        ;  Get the LUT value.
                  beq     knob_CW           ;  The knob was turned clockwise if LUT value is zero.
knob_CCW:         ldx     knob_count        ;  Get the loop count number.
                  lslx                      ;  Multiply it by two.
                  lda     #$20              ;  Load the action required mask (bit 5).
                  sta     knobstate+1,x     ;  Set the action required bit for this knob's CCW state machine.
```

```
                lda     knob_accel+1,x    ;  See if the knob needs to be accelerated.
                bne     knob_CCW_ac       ;  Branch to accelerate routine if the timeout has not happened.
                clr     knob_inc_ptr+1,x  ;  Reset the knob_inc_ptr to no acceleration.
                pshx                      ;  Save X.
                ldx     knob_inc_ptr+1,x  ;  Get the current acceleration count.
                lda     knob_inc_LUT,x    ;  Use it to get the acceleration (repetition) value.
                pulx                      ;  Restore X.
                sta     knob_inc_count+1,x        ;  Store the repetition value for this knob.
                lda     #!100             ;  Initialize the acceleration countdown.
                sta     knob_accel+1,x    ;  Store the acceleration countdown value.
                rts                       ;  Return to main_loop.

knob_CCW_ac:    inc     knob_inc_ptr+1,x  ;  Point to the next acceleration (repetition) value).
                pshx                      ;  Save X.
                ldx     knob_inc_ptr+1,x  ;  Get the current acceleration count.
                lda     knob_inc_LUT,x    ;  Use it to get the acceleration (repetition) value.
                pulx                      ;  Restore X.
                sta     knob_inc_count+1,x        ;  Store the repetition value for this knob.
                lda     #!100             ;  Reset the acceleration countdown.
                sta     knob_accel+1,x    ;  Store the countdown value in the knob_accel array.
                rts                       ;  Return to caller.

knob_CW: ldx    knob_count                ;  Get the loop count number.
                lslx                      ;  Multiply it by two.
                lda     #$20              ;  Load the action required mask (bit 5).
                sta     knobstate,x       ;  Set the action required bit for this knob's CCW state machine.
                lda     knob_accel,x      ;  See if the knob needs to be accelerated.
                bne     knob_CW_ac        ;  Branch to accelerate routine if so.
                clr     knob_inc_ptr,x    ;  Reset the knob_inc_ptr to no acceleration.
                pshx                      ;  Save X.
                ldx     knob_inc_ptr,x    ;  Get the current acceleration count.
                lda     knob_inc_LUT,x    ;  Use it to get the acceleration (repetition) value.
                pulx                      ;  Restore X.
                sta     knob_inc_count,x  ;  Store the repetition value for this knob.
                lda     #!100             ;  Initialize the acceleration countdown.
                sta     knob_accel,x      ;  Store the acceleration countdown value.
                rts                       ;  Return to main_loop.

knob_CW_ac:     inc     knob_inc_ptr,x    ;  Point to the next acceleration (repetition) value).
                pshx                      ;  Save X.
                ldx     knob_inc_ptr,x    ;  Get the current acceleration count.
                lda     knob_inc_LUT,x    ;  Use it to get the acceleration (repetition) value.
                pulx                      ;  Restore X.
                sta     knob_inc_count,x  ;  Store the repetition value for this knob.
                lda     #!100             ;  Reset the acceleration countdown.
                sta     knob_accel,x      ;  Store the countdown value in the knob_accel array.
knob_same:      rts                       ;  Return to caller.

knob_LUT:       db      $ff               ;  Table address 0 (unused).
                db      $01               ;  Table address 1 (CCW).
                db      $00               ;  Table address 2 (CW).
                db      $ff               ;  Table address 3 (unused).
                db      $00               ;  Table address 4 (CW).
                db      $ff               ;  Table address 5 (unused).
                db      $ff               ;  Table address 6 (unused).
                db      $01               ;  Table address 7 (CCW).
                db      $01               ;  Table address 8 (CCW).
                db      $ff               ;  Table address 9 (unused).
                db      $ff               ;  Table address a (unused).
                db      $00               ;  Table address b (CW).
                db      $ff               ;  Table address c (unused).
                db      $00               ;  Table address d (CW).
                db      $01               ;  Table address e (CCW).
                db      $ff               ;  Table address f (unused).

knob_inc_LUT:   db      !1                ;  The acceleration table.
                db      !2
                db      !4
                db      !8
                db      !12
```

```
                db      !16
                db      !20
                db      !25
                db      !30
                db      !35
                db      !40
                db      !45
                db      !50
                db      !60
                db      !70
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;In order for the sCo-Pilot to manipulate the cursors, waveform offsets (H and V positions),
;and trigger position, a small collection of rudimentary floating point operations is
;required.

;Position data from the scope is returned in Tek's <NR3> format, which is simply a floating
;point value with an exponent (see THS programming manual).  Although the data length of a
;query may vary depending on mode and value, the scope will accept a command of fixed
;data length.  A few examples of the differences are given below:

;    Query format (from scope)     Command format (to scope)
;          -2.0E-6                       -02.00E-6
;          1.2E-3                        +01.20E-3
;          -4.23E1                       -04.23E+1

;The sCo-Pilot will accept a response in the query format and translate it into a pure BCD
;format. Due to the range of the numbers expected from the scope, it is necessary to reserve
;six digits (3 bytes) for numbers to the left of the decimal point and 10 digits (5 bytes)
;for numbers to the right of the decimal point.  Including a byte for the sign, this format
;requires 9 bytes per value.

;The ascii2fixed routine converts the floating point data format from the scope to the
;sCo-Pilot's working fixed point BCD format (described above).  The source data is contained
;in inbuf and the destination data is stored in BCD_source.  This routine
;is called only when there is floating point data to process.
;Note: the scope will never return exponents greater than 10.

ascii2fixed:    pshh                    ;  Save H.
                pshx                    ;  Save X.
                psha                    ;  Save A

                clrh                    ;  Clear H.
                ldx     #!09            ;  Initialize X for loop counting.
clear_source:   clr     BCD_source-1,x  ;  Clear the source numbers.
                dbnzx   clear_source    ;  Loop until all clear.


find_end:       lda     inbuf,x         ;  Load the current character of the input buffer.
                cmp     #!10            ;  Look for the termination character.
                beq     found_end       ;  Continue to process once the end of inbuf is found.
                incx                    ;  Point to the next character of inbuf.
                bra     find_end        ;  Loop until the termination character is located.

found_end:      decx                    ;  Point X to the last digit of the exponent.
                lda     inbuf,x         ;  Get the last digit of the exponent.
                sub     #'0'            ;  Convert it to BCD.
                sta     BCD_exp         ;  Temporarily store the exponent.

                decx                    ;  Point X to the previous character of inbuf.
                lda     inbuf,x         ;  Get the second to last exponent digit.
                cmp     #'-'            ;  See if it is a minus sign.
                beq     exponent_minus  ;  Insert a minus sign if so.
                cmp     #'E'            ;  See if it is the exponent character.
                beq     exponent_done   ;  No more exponent digits.

                mov     #!10,BCD_exp    ;  At this point, the exponent must be 10.
```

```
                lda     inbuf,x         ;  Get the previous character.
                cmp     #'-'            ;  See if it is a minus sign for the exponent.
                beq     exponent_minus  ;  Insert a minus sign if so.
                bra     exponent_done   ;  Continue to process numbers.

exponent_minus: com     BCD_exp         ;  Convert sign of exponent.
                inc     BCD_exp         ;  Increment it to produce the proper number of shifts.
exponent_done:  lda     inbuf           ;  Get the first character of inbuf.
                cmp     #'-'            ;  See if it is a minus sign
                bne     get_mant        ;  Get the mantissa if not.
                bset    0,BCD_source+8  ;  Set the minus sign bit.

get_mant:       clrx                    ;  Clear X.
                clrh                    ;  Clear H for IX1 addressing.
find_DP: lda    inbuf,x                 ;  Get the next character from inbuf.
                cmp     #'.'            ;  See if it is a decimal point.
                beq     found_DP        ;  Continue to process if decimal point is found.
                incx                    ;  Point X to the next character.
                bra     find_DP         ;  Loop until the decimal point is found.
found_DP:       stx     BCD_DP          ;  Store the location of the DP for future use.
                incx                    ;  Point X to the character to the right of the DP.
                lda     inbuf,x         ;  Get the tenths digit.
                sub     #'0'            ;  Convert it to BCD.
                nsa                     ;  Put the BCD value into the high nibble.
                sta     BCD_tenths      ;  Store the tenths digit.
                incx                    ;  Point X to the next character in inbuf.
                lda     inbuf,x         ;  Get the next character.
                cmp     #'E'            ;  See if it is the exponent character.
                beq     rightDP_done    ;  If so, everything to the right of the DP is done.
                sub     #'0'            ;  Convert it to BCD.
                ora     BCD_tenths      ;  Combine the high nibble and the low nibble.
                sta     BCD_tenths      ;  Store the tenths and hundredths BCD values.
rightDP_done:   ldx     BCD_DP          ;  Get the position of the decimal point.
                beq     fixed_done      ;  Skip if there are no characters to the left of the DP.
                decx                    ;  Point X to the next character to the left of the DP.
                lda     inbuf,x         ;  Get the ones digit.
                sub     #'0'            ;  Convert it to BCD.
                sta     BCD_ones        ;  Store the ones digit.
                cpx     #$00            ;  See if X points to the beginning.
                beq     fixed_done      ;  If so, the conversion is done.
                decx                    ;  Decrement X to point to the second character to the left of DP.
                lda     inbuf,x         ;  Get the next character.
                cmp     #'-'            ;  See if it is a minus sign.
                beq     fixed_done      ;  If so, the conversion is done.  Otherwise process tens digit.
                sub     #'0'            ;  Convert it to BCD.
                nsa                     ;  Put the tens digit into the upper nibble.
                ora     BCD_ones        ;  Combine the high and low nibbles.
                sta     BCD_ones        ;  Store the tens and ones BCD values.

fixed_done:     lda     #$08            ;  Load the shift offset.
                sub     BCD_exp         ;  Subtract the signed exponent.  A now contains the number of required nibble
shifts.
                sta     nibble_count    ;  Store the number of required nibble shifts.

nibble_shift:   mov     #4,shift_count  ;  Initialize the number of bit shifts.
nibble_loop:    clc                     ;  Initialize the carry bit.
                clrx                    ;  Clear X.
                lda     #!10            ;  Initialize the number of bytes to shift through.
shift:          ror     BCD_ones,x      ;  Shift-out the next bit into C.
                incx                    ;  Advance the address pointer to the next byte.
                dbnza   shift           ;  Loop until all ten locations have been shifted.
                dbnz    shift_count,nibble_loop    ;  Loop until a full nibble has been shifted.
                dbnz    nibble_count,nibble_shift  ;  Loop until the required number of nibbles have been shifted.

                pula                    ;  Restore A.
                pulx                    ;  Restore X.
                pulh                    ;  Restore H.
                rts                     ;  Return to the caller.
;-----------------------------------------------------------------------------------------
```

```
;-------------------------------------------------------------------------------------------
;The BCD2float routine converts the sCo-Pilot's fixed point BCD storage format (described above)
;to ASCII floating point for transmission to the scope.  The completed ASCII conversion is
;stored in RAM location float_buf for transmission to the scope.  The source data is located
;at BCD_dest.  The termination character is appended to float_buf.

default_buf:      db       '+00.00E+0',0

BCD2float:        pshh                     ;  Save H.
                  pshx                     ;  Save X.
                  psha                     ;  Save A
                  clrh                     ;  Clear H for IX1 mode.
                  clrx                     ;  Initialize X to point to the beginning of default_buf.
load_default:     lda      default_buf,x   ;  Get the next character of the default buffer.
                  beq      mant_sign       ;  When the default buffer is loaded, get the mantissa sign.
                  sta      float_buf,x     ;  Store it in the output buffer.
                  incx                     ;  Point X to the next character.
                  bra      load_default    ;  Loop until the default buffer is loaded.

mant_sign:        lda      BCD_dest+8      ;  Get the sign of the mantissa.
                  beq      positive        ;  Skip if positive.
                  lda      #'-'            ;  Load the ASCII value of a minus sign.
                  sta      float_buf       ;  Overwrite the previously stored plus sign.

positive:         clrx

zero_loop:        lda      BCD_dest,x      ;  Get the next character in the BCD number.
                  bne      non_zero        ;  Find the first non-zero byte.
                  incx                     ;  Point X to the next character.
                  cpx      #$8             ;  See if the end of BCD_dest has been reached.
                  beq      BCD2float_done  ;  Leave the default buffer if mantissa is zero.
                  bra      zero_loop       ;  Loop until the first non-zero character is found.

non_zero:         nsa                      ;  Put the high nibble into the low.
                  and      #$0f            ;  Remove the high nibble.
                  add      #'0'            ;  Convert to ASCII.
                  sta      float_buf+1     ;  Store it as the first digit.
                  lda      BCD_dest,x      ;  Get the first non-zero nibble.
                  and      #$0f            ;  Remove the high nibble.
                  add      #'0'            ;  Convert to ASCII.
                  sta      float_buf+2     ;  Store it as the second digit.
                  lda      #'.'            ;  Load the ASCII value of a decimal point.
                  sta      float_buf+3     ;  Store it as the third digit.
                  incx                     ;  Point X to the next byte in BCD_dest.
                  lda      BCD_dest,x      ;  Get the next digit.
                  nsa                      ;  Put the high nibble into the low.
                  and      #$0f            ;  Remove the high nibble.
                  add      #'0'            ;  Convert to ASCII.
                  sta      float_buf+4     ;  Store it as the fourth digit.
                  lda      BCD_dest,x      ;  Get the next non-zero nibble.
                  and      #$0f            ;  Remove the high nibble.
                  add      #'0'            ;  Convert to ASCII.
                  sta      float_buf+5     ;  Store it as the fifth digit.
                  lda      #'E'            ;  Load the ASCII value of the exponent character.
                  sta      float_buf+6     ;  Store it as the sixth digit.
                  decx                     ;  Decrement X for lookup table use.
                  lslx                     ;  Multiply X by two.
                  lda      exp_LUT,x       ;  Get the sign of the exponent.
                  sta      float_buf+7     ;  Store it in the output buffer.
                  lda      exp_LUT+1,x     ;  Get the exponent value.
                  sta      float_buf+8     ;  Store it in the output buffer.
                  lda      #$0A            ;  Load the scope's termination character.
                  sta      float_buf+9     ;  Store it in the output buffer.
                  lda      #$00            ;  Load the sCo-Pilot's termination character.
                  sta      float_buf+!10   ;  Store it in the output buffer.
BCD2float_done:   pula                     ;  Restore A.
                  pulx                     ;  Restore X.
                  pulh                     ;  Restore H.
                  rts                      ;  Return to the caller.
```

```
exp_LUT: db       '+4'                      ;  Exponent lookup table.
                  db       '+2'
                  db       '+0'
                  db       '-2'
                  db       '-4'
                  db       '-6'
                  db       '-8'
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The get_Y_inc routine uses the floating point vertical scale number located in inbuf
;to look-up the appropriate Y increment for trigger level, vertical offset level, and vertical
;cursors.  The Y increment is placed in inbuf.  The scale to increment conversion is performed
;according to the following static relationship:

;      Y scale (in inbuf from previous poll)           Y increment
;                  1.0E(+/-)X                          .02E(+/-)X
;                  2.0E(+/-)X                          .04E(+/-)X
;                  5.0E(+/-)X                          0.1E(+/-)X

;So, according to this relationship, if the first digit is 1, then the increment is .02.  Similarly,
;if the first digit is 2, then the increment is .04 and so on.  Note that the exponent and its
;sign remain the same as the Y scale value.  As such, only the first three characters of inbuf
;need to be changed as a result of the lookup.

get_Y_inc:        psha                      ;  Save A.
                  lda      inbuf            ;  Get the first character of inbuf.
                  cmp      #'1'             ;  See if it is a 1.
                  beq      Yinc_02          ;  Load the increment with .02 if so.
                  cmp      #'2'             ;  See if it is a 1.
                  beq      Yinc_04          ;  Load the increment with .04 if so.
Yinc_01: mov      #'0',inbuf                ;  Load the first character of inbuf with the increment.
                  mov      #'.',inbuf+1     ;  Load the second character of inbuf with the increment.
                  mov      #'1',inbuf+2     ;  Load the third character of inbuf with the increment.
                  pula                      ;  Restore A.
                  rts                       ;  Return to caller with the increment in inbuf.
Yinc_02: mov      #'.',inbuf                ;  Load the first character of inbuf with the increment.
                  mov      #'0',inbuf+1     ;  Load the second character of inbuf with the increment.
                  mov      #'2',inbuf+2     ;  Load the third character of inbuf with the increment.
                  pula                      ;  Restore A.
                  rts                       ;  Return to caller with the increment in inbuf.
Yinc_04: mov      #'.',inbuf                ;  Load the first character of inbuf with the increment.
                  mov      #'0',inbuf+1     ;  Load the second character of inbuf with the increment.
                  mov      #'4',inbuf+2     ;  Load the third character of inbuf with the increment.
                  pula                      ;  Restore A.
                  rts                       ;  Return to caller with the increment in inbuf.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The mult_inc_by_10 routine multiplies the contents of BCD_inc by 10.  This is accomplished
;simply by shifting all 16 BCD nibbles one position to the left.  The most significant nibble
;is lost.  This routine is used to condition the X increment when the scope is not in magnify
;mode.

mult_inc_by_10:   psha                      ;  Save A.
                  pshx                      ;  Save X.
                  pshh                      ;  Save H.
                  clrh                      ;  Clear H for IX1 addressing.
                  clrx                      ;  Point X to the beginning of BCD_inc.
                  clr      BCD_inc+8        ;  Make sure the increment sign byte is clear.

mult_loop:        lda      BCD_inc+1,x      ;  Get the next nibble.
                  rola                      ;  Put the next MSB into C.
                  rol      BCD_inc,x        ;  Put C into the next nibble.
                  rola                      ;  Put the next MSB into C.
                  rol      BCD_inc,x        ;  Put C into the next nibble.
                  rola                      ;  Put the next MSB into C.
```

```
                rol     BCD_inc,x       ;  Put C into the next nibble.
                rola                    ;  Put the next MSB into C.
                rol     BCD_inc,x       ;  Put C into the next nibble.

                incx                    ;  Increment X to point to the next byte.
                cpx     #$07            ;  See if the multiply is done.
                bne     mult_loop       ;  Continue to shift nibbles until done.

                pulh                    ;  Restore H.
                pulx                    ;  Restore X.
                pula                    ;  Restore A.
                rts                     ;  Return to caller.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The in_232 routine receives one character per call and places it in the input buffer, inbuf.
;When the termination character from the scope is received, the waiting flag is cleared to allow
;the received data to be processed.

in232:          brclr   5,SCS1,in232_ret  ;  Get-out if there is no 232 received data.
                pshx                    ;  Save X.
                psha                    ;  Save A.
                pshh                    ;  Save H for zero page IX1 mode.
                clrh                    ;  Point H at page zero for IX1 mode.
                ldx     inbuf_ptr       ;  Load the current data pointer.
                lda     SCDR            ;  Get the new character.
                sta     inbuf,x         ;  Store it in inbuf.
                cmp     #!10            ;  Check for termination.
                beq     receive_done    ;
                inc     inbuf_ptr       ;  Increment the pointer for next character.
in232_done:     pulh                    ;  Restore H.
                pula                    ;  Restore A.
                pulx                    ;  Restore X.
in232_ret:      rts                     ;  Return to main_loop.

receive_done:   clr     waiting         ;  Clear the waiting flag.
                clr     inbuf_ptr       ;  Reset inbuf_ptr.
                bra     in232_done
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The out232 routine transmits one character per call if the transmitter is available (not busy).
;The data to be sent is pointed to by out_data_ptr.  If the high byte of out_data_ptr is non-zero,
;there is pending message to be sent.  The data is terminated by the nul character ($00).  The
;high byte of out_data_ptr is cleared when the transmission is complete.  Other routines can
;poll the high byte of out_data_ptr to determine if there is a transmission in progress.

out232:         psha                    ;  Save A.
                lda     out_data_ptr    ;  Check for data to send.
                beq     out232_ret      ;  No data to transmit if out_data_ptr is zero.
                brclr   6,SCS1,out232_ret ;  SCI Transmission in progress (busy).

                pshx                    ;  Save X.
                pshh                    ;  Save H.

                ldhx    out_data_ptr    ;  Point H:X to the output data.
                lda     ,x              ;  Get the next byte to be sent.
                beq     string_done     ;  If it is zero (null character), then the string has been sent.
                sta     SCDR            ;  Send the byte.
                aix     #$0001          ;  Increment the output data pointer (16 bits).
                sthx    out_data_ptr    ;  Store it (16 bits).

out232_done:    pulh                    ;  Restore H.
                pulx                    ;  Restore X.
out232_ret:     pula                    ;  Restore A.
                rts                     ;  Return to main_loop.
string_done:    clr     out_data_ptr    ;  Clear the high byte of out_data_ptr to indicate transmission done.
                bra     out232_done
```

```
;--------------------------------------------------------------------------------------------


;--------------------------------------------------------------------------------------------
;The debounce_keys routine examines the state of each key.  If the key is pressed, a timestamp
;is assigned to that key.  The key is debounced when 30ms of consecutive closure is detected.
;Upon successful debounce, the debounced bit and the action required bit are set in the keystate
;byte.  The debounce_keys routine cannot clear the action required bit - only the action
;service routine can clear this bit.
;The following statements summarize the debouncing process:
;   If the key is open (not pressed), clear the debounced bit and the closure time.
;   If the debounced bit is set, ignore the key because it is still pressed.
;   If the key is pressed, increment the closure time.
;   If the closure time has reached 30ms, set the debounced bit and the action required bit.
;Note: the keystate array is offset by -1 to allow efficient use of the DBNZX instruction.

debounce_keys:    pshh                      ;  Save H (for IX1 mode).
                  psha                      ;  Save A.
                  pshx                      ;  Save X.

                  clrh                      ;  Point H to zero page for IX1 mode.
                  mov     timecount+1,keytime       ;  'Stamp' the entry time into this routine.
                  ldx     #$10              ;  Point X to the last element in the keystate array.

check_key:        lda     #$80              ;  Put the pressed mask (bit 7) into A.
                  bit     keystate-1,x      ;  See if the key is pressed.
                  beq     key_open          ;  Clear the debounced bit and closure time if the key is open.

                  clr     idle_count+1      ;  Reset the low byte of idle_count since a key is pressed.
                  clr     idle_count        ;  Reset the high byte of idle_count since a key is pressed.

                  lda     #$40              ;  Put the debounced mask (bit 6) into A.
                  bit     keystate-1,x      ;  See if the key is already debounced.
                  bne     next_key          ;  Do nothing if the key is already debounced.

                  lda     keystate-1,x      ;  Put the key state in A for time processing.
                  and     #$1f              ;  Remove everything but the closure time.
                  cmp     #!30              ;  See if 30ms have elapsed on this key.
                  beq     key_debounced     ;  Set the debounced & action bits if 30ms of closure has elapsed.

                  eor     keytime           ;  See if a timecount has elapsed since last debounce time.
                  and     #$01              ;  A timecount has elapsed if the LSB is set.
                  beq     next_key          ;  Do nothing if a timecount has not elapsed.

                  inc     keystate-1,x      ;  Increment the closure time for this key.
                  bra     next_key          ;  Check the next key.

key_open:         lda     #$20              ;  Mask to clear debounce bit and closure time, but not action bit.
                  and     keystate-1,x      ;  Clear the bits.
                  sta     keystate-1,x      ;  Store the new value in the array.
next_key:         dbnzx   check_key         ;  Check the next key in the array.

debounce_done:    pulx                      ;  Restore X.
                  pula                      ;  Restore A.
                  pulh                      ;  Restore H.
                  rts                       ;  Return to main_loop.

key_debounced:    lda     #$60              ;  OR mask for setting the debounced and action required bits.
                  ora     keystate-1,x      ;  Set the bits.
                  sta     keystate-1,x      ;  Store the result in the keystate array.
                  bra     next_key          ;  Move to the next key.
;--------------------------------------------------------------------------------------------


;--------------------------------------------------------------------------------------------
;The heartbeat routine provides a visual indication that the CPU is running.  The LED on
;PORTC1 gets toggled 512 milliseconds.  The free running counter, timecount, is the basis
;for heartbeat.

heartbeat:        brset   1,timecount,heart_on      ;  If bit 9 of timecount is set, turn on heartbeat LED (2^9=512ms).
```

```
heart_off:      bset    0,PTC           ;  Extinguish the heartbeat LED.
                rts                     ;  Return to main_loop.
heart_on:       bclr    0,PTC           ;  Light the heartbeat LED.
                rts                     ;  Return to main_loop.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The read_keys routine reads the status of each of the 16 keys and sets the corresponding
;bit in the keystate array.  A set bit 7 indicates a currently pressed key.  Although this method
;allows simultaneous key closures, they are meaningless in this application.  The keys are in
;a 4x4 matrix on PORTA. See schematic.
;Each byte in the keystate array represents the state of each key according to the following
;bit definitions:
;-------------------------------------------------------------------------------
;|  bit 7  |  bit 6  |  bit 5  |  bit 4  |  bit 3  |  bit 2  |  bit 1  |  bit 0  |
;|         |         |         |                                                   |
;| pressed |debounced| action  |       time of consecutive closure (ms)          |
;|         |         | required|                                                   |
;-------------------------------------------------------------------------------

read_keys:      psha                    ;  Save the accumulator.
                pshx                    ;  Save X.
                pshh                    ;  Save H for IX1 mode.
                clrh                    ;  Point H to page zero for IX1 mode.

                clrx                    ;  Initialize X to point to the first element in the keystate array.
                mov     #$f0,PTA        ;  Turn-on the columns.
                mov     #$04,col_count  ;  Initialize the column counter.
                mov     #$08,DDRA       ;  Initialize DDRA to be one shift more than the first value.

kcol:           lsl     DDRA            ;  Turn-off the current column and turn-on the next column.
                mov     #$04,row_count  ;  Initialize the row counter for 4 rows.
                mov     PTA,key_image   ;  Make a copy of the currently pressed keys.

krow:           asr     key_image       ;  Put the LSB into C.
                bcs     pressed
not_pressed:    lda     #$7f            ;  Load A with a mask to clear bit 7 of keystate.
                and     keystate,x      ;  Clear the 'pressed' bit.
                bra     store_key

pressed: lda    #$80                    ;  Load A with a mask to set bit 7 of keystate.
                ora     keystate,x      ;  Set the 'pressed' bit.

store_key:      sta     keystate,x      ;  Store it back in the keystate array.
                incx                    ;  Point X to the next element in the keystate array.
                dbnz    row_count,krow  ;  Continue to read the four rows.
                dbnz    col_count,kcol  ;  Continue to read the four columns.

                mov     #$00,DDRA       ;  Turn-off all columns.

                pulh                    ;  Restore H.
                pulx                    ;  Restore X.
                pula                    ;  Restore the accumulator.
                rts                     ;  Return to main_loop.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The init_SCI subroutine sets-up the serial communications interface for 9600BPS, 8 data bits,
;no parity, and one stop bit.  The transmitter and receiver are enabled with SCI interrupts
;disabled.

init_SCI:       mov     #$05,SCBR       ;  9600BPS with 19.661Mhz osc.
                mov     #$40,SCC1       ;  8 data bits, no parity, enable SCI.
                mov     #$0C,SCC2       ;  No SCI interrupts, enable transmitter & receiver.
                rts                     ;  Return to main_loop.
;-------------------------------------------------------------------------------------------
```

```
;--------------------------------------------------------------------------------------------
;The init_SPI routine configures the serial peripheral interface to operate at full speed with
;no interrupts.  The SPI IO expander CPLD is the only SPI device on the bus.

init_SPI:         mov      #$3a,SPCR         ;  CPHA=1, CPOL=1, no interrupts, master mode, enable SPI.
                  mov      #$00,SPSCR        ;  No error interrupts, run at full speed.
                  mov      #$ff,DDRD         ;  Configure PORTD0 for output (MISO, SCK, \CS).
                  bset     0,PTD             ;  Nullify \CS.
                  rts                        ;  Return to main_loop.
;--------------------------------------------------------------------------------------------


;--------------------------------------------------------------------------------------------
;The Timer module is configured to produce a timer interrupt every 1ms.

init_TIM:         mov      #$50,T1SC         ;  Enable TOF interrupt, reset & start timer.
                  mov      #$13,T1MODH       ;  Set the high byte of timer 1 modulo (4878 for 1ms).
                  mov      #$0e,T1MODL       ;  Set the low byte of timer 1 modulo (4878 for 1ms).
                  rts                        ;  Return to main_loop.
;--------------------------------------------------------------------------------------------


;--------------------------------------------------------------------------------------------
;The update_LEDs routine sends the four bytes from RAM locations LED_bank_A through LED_bank_D
;to the SPI IO expander CPLD over the SPI.

update_LEDs:      bclr     0,PTD             ;  Assert \CS for the LED driver.
                  brclr    3,SPSCR,$         ;  Loop until TX empty.
                  mov      LED_BANK_D,SPDR   ;  Send bank D data (upper 5 bits will be lost).
                  brclr    3,SPSCR,$         ;  Loop until TX empty.
                  mov      LED_BANK_C,SPDR   ;  Send bank C data.
                  brclr    3,SPSCR,$         ;  Loop until TX empty.
                  mov      LED_BANK_B,SPDR   ;  Send bank B data.
                  brclr    3,SPSCR,$         ;  Loop until TX empty.
                  mov      LED_BANK_A,SPDR   ;  Send bank A data.
                  lda      #!6               ;  3.6621us delay for last byte to finish.
                  dbnza    $                 ;3 Delay for last SPI transmission.
                  bset     0,PTD             ;  Clear \CS for the LED driver.
                  brclr    0,waiting,led_wait         ;  Extinguish waiting LED if waiting bit is clear.
                  bclr     4,PTC             ;  Light waiting LED if necessary.
                  rts                        ;  Return to main_loop.
led_wait:         bset     4,PTC             ;  Extinguish waiting LED.
update_done:      rts                        ;  Return to main_loop.
;--------------------------------------------------------------------------------------------


;--------------------------------------------------------------------------------------------
;The keep_time ISR is executed when Timer 1 overflows.  The modulus of T1, combined with
;the ISR time, increments the 16 bit time counter (timecount) every 1ms.  The timecount
;value is free-running and is allowed to overflow.
keep_time:        pshh                       ;2  Save the high byte of X (the low byte is pushed by interrupt).
                  ldhx     timecount         ;4  Get the existing 16 bit time value.
                  aix      #$0001            ;2  Increment the time value.
                  sthx     timecount         ;4  Store the new time value.
                  lda      T1SC              ;3  Read T1SC to clear TOF.
                  bclr     7,T1SC            ;4  Clear TOF.
                  pulh                       ;2  Restore the high byte of X.
                  rti                        ;7
;--------------------------------------------------------------------------------------------


;--------------------------------------------------------------------------------------------
;The fixed_add routine sums the fixed point values found in BCD_source and BCD_inc.  BCD_inc,
;the addend, will always be positive.  If the BCD_source is negative, the subtract routine will
;be called instead.  The result is stored in BCD_dest.  A, X, and H are destroyed by this routine.

fixed_add:        brset    0,BCD_source+8,sub_unsigned         ;  Subtract if the source is negative.
add_unsigned:     ldx      #$08              ;  Initialize X for the number of BYTES to add.
                  clrh                       ;  Clear H for IX1 addressing.
```

```
                clc                     ;  Initialize the carry bit.
add_loop:       lda     BCD_source-1,x  ;  Get the source byte.
                adc     BCD_inc-1,x     ;  Add the increment byte.
                daa                     ;  Adjust BCD coding.
                sta     BCD_dest-1,x    ;  Store the result in BCD_dest.
                dbnzx   add_loop        ;  Loop until all digits have been added.
                lda     BCD_source+8    ;  Get the sign of the source.
                sta     BCD_dest+8      ;  Transfer it to the sign of the destination.
                rts                     ;  Return to caller with A, X, and H destroyed.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The fixed_sub routine subtracts the value in BCD_inc from BCD_source.  The subtrahend, BCD_inc,
;will always be positive.  If the source is negative, the fixed_add routine is called instead.
;The result is stored in BCD_dest.  A, X, and H are destroyed by this routine.

fixed_sub:      brset   0,BCD_source+8,add_unsigned       ;  Add if the source is negative.
sub_unsigned:   ldx     #$08            ;  Initialize X with the number of BYTES to subtract.
                clrh                    ;  Clear H for IX1 addressing.
zero_test:      lda     BCD_source-1,x  ;  Get a source digit.
                bne     sub_loop_init   ;  Prepare to subtract if it is not zero.
                dbnzx   zero_test       ;  Loop until all digits have been checked.
                bset    0,BCD_source+8  ;  Make the value source value negative.
                bra     add_unsigned    ;  Add the subtrahend to the negative zero value.
sub_loop_init:  ldx     #$08
                clr     borrow          ;  Initialize the borrow status.
sub_loop:       lda     BCD_source-1,x  ;  Load the next source value.
                and     #$0f            ;  Remove the high nibble.
                sta     nib_source      ;  Store it as the source nibble.
                lda     BCD_inc-1,x     ;  Load the next increment nibble.
                and     #$0f            ;  Remove the high nibble.
                sta     nib_sub         ;  Store it as the subtrahend.
                bsr     nibble_sub      ;  Perform the subtraction.
                sta     BCD_dest-1,x    ;  Store the low nibble of the subtraction.

                lda     BCD_source-1,x  ;  Load the next source value.
                nsa                     ;  Put the high nibble into the low nibble.
                and     #$0f            ;  Remove the high nibble.
                sta     nib_source      ;  Store it as the source nibble.
                lda     BCD_inc-1,x     ;  Load the next increment nibble.
                nsa                     ;  Put the high nibble into the low nibble.
                and     #$0f            ;  Remove the high nibble.
                sta     nib_sub         ;  Store it as the subtrahend.
                bsr     nibble_sub      ;  Perform the subtraction.
                nsa                     ;  Put the result into the high nibble
                ora     BCD_dest-1,x    ;  Combine the low & high nibbles.
                sta     BCD_dest-1,x    ;  Store the result.
                dbnzx   sub_loop        ;  Repeat until all digits have been subtracted.
                lda     BCD_source+8    ;  Get the sign of the source.
                sta     BCD_dest+8      ;  Transfer it to the sign of the destination.
                rts                     ;  Return to caller.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;The nibble_sub subroutine subtracts one BCD nibble from another.  Bit zero of the borrow
;location will be set if a borrow occurred, cleared otherwise.  The source nibble should be
;in nib_source and the subtrahend should be in nib_sub.  The result of the subtract
;is returned in the accumulator.

nibble_sub:     brset   0,borrow,prev_borrow      ;  Deal with a previous borrow operation.
subtract:       lda     nib_source      ;  Load the source nibble.
                sub     nib_sub         ;  Subtract the subtrahend.
                bmi     borrow_now      ;  If the result is negative, a borrow is needed.
                bclr    0,borrow        ;  Clear the borrow bit.
                rts                     ;  Return to caller.
borrow_now:     lda     nib_source      ;  Load the source nibble.
                add     #!10            ;  Add ten.
                sub     nib_sub         ;  Subtract the subtrahend.
```

```
                bset    0,borrow        ;  Indicate that a borrow occurred.
                rts                     ;  Return to caller.
prev_borrow:    dec     nib_source      ;  Perform the borrow on the source nibble.
                bpl     subtract        ;  Continue the subtract if another borrow is not required.
                inc     nib_source      ;  Re-increment the source nibble.
                lda     nib_source      ;  Get the original value.
                add     #!09            ;  Add nine to account for the borrow.
                sub     nib_sub         ;  Subtract the subtrahend.
                bset    0,borrow        ;  Indicate that a borrow occurred.
                rts                     ;  Return to caller.
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;The do_keys routine inspects the keystate & knobstate array to determine if any keys have
;been debounced or knobs turned and require action.  If action is in progress, key_vector
;will contain the address of the currently running key service routine.  If there is no action
;in progress, key_vector will contain the address of the do_process label.  This prevents
;the simultaneous execution of a key service routine.

do_keys: pshx                           ;  Save X.
                pshh                    ;  Save H.
                psha                    ;  Save A.

                ldhx    key_vector      ;  Point H:X to the address of the currently running key service routine.
                jmp     ,x              ;  Run that key service routine (or do_process).
do_process:     clrh                    ;  Point H to page zero for IX1 mode.

                lda     #$20            ;  The mask for the action required bit.
                ldx     #!24            ;  Prepare to scan the 24 elements of the keystate & knobstate array.
do_key_loop:    bit     keystate-1,x    ;  Check the action required bit in the keystate array.
                bne     make_jump       ;  Create the jump address if action is required.
                dbnzx   do_key_loop     ;  Progress through the keystate array

                clr     state_inq       ;  When all of the action required bits are clear, the scope state inquiry is
complete.
do_key_return:  pula                    ;  Restore A.  At this point, no keys require action.
                pulh                    ;  Restore H.
                pulx                    ;  Restore X.
                rts                     ;  Return to main_loop.

make_jump:      decx                    ;  Subtract 1 from X to go from 0 to 23.
                aslx                    ;  Multiply X by two.
                aslx                    ;  Multiply X by two for a total of four.
                jmp     jump_table,x    ;  Go to the appropriate key handler routine.

;Because each key provides a different function with different responses and protocol, it is
;convenient to write a separate handler for each key rather than to attempt a general handler
;for all keys.  Such a general handler, in the sCo-Pilot's case, would require more
;exceptions than rules.  So, the unique response to each key is implemented via a jump table.
;The jump_table contains JMP instructions to branch the program to the appropriate key handler
;routine.  The table is padded with NOPs to avoid multiplying X (the table offset pointer) by 3.
;A multiply by 4 only requires two shifts (see make_jump).

jump_table:     jmp     key_0           ;  Key 0 handler (cursor knob toggle).
                nop                     ;  Pad one byte.
                jmp     key_1           ;  Key 1 handler (horizontal knob toggle).
                nop                     ;  Pad one byte.
                jmp     key_2           ;  Key 2 handler (vertical knob toggle).
                nop                     ;  Pad one byte.
                jmp     key_3           ;  Key 3 handler (trigger knob toggle).
                nop                     ;  Pad one byte.
                jmp     key_4           ;  Key 4 handler (realtime/persist button).
                nop                     ;  Pad one byte.
                jmp     key_5           ;  Key 5 handler (vectors/dots button).
                nop                     ;  Pad one byte.
                jmp     key_6           ;  Key 6 handler (sample/average button).
                nop                     ;  Pad one byte.
                jmp     key_7           ;  Key 7 handler (continuous/singleshot button).
                nop                     ;  Pad one byte.
```

```
                jmp     key_8           ;  Key 8 handler (channel 1 coupling button).
                nop                     ;  Pad one byte.
                jmp     key_9           ;  Key 9 handler (trigger source button).
                nop                     ;  Pad one byte.
                jmp     key_10          ;  Key 10 handler (trigger slope button).
                nop                     ;  Pad one byte.
                jmp     key_11          ;  Key 11 handler (auto/normal trigger button).
                nop                     ;  Pad one byte.
                jmp     key_12          ;  Key 12 handler (unused).
                nop                     ;  Pad one byte.
                jmp     key_13          ;  Key 13 handler (unused).
                nop                     ;  Pad one byte.
                jmp     key_14          ;  Key 14 handler (cursor type select button).
                nop                     ;  Pad one byte.
                jmp     key_15          ;  Key 15 handler (channel 2 coupling button).
                nop                     ;  Pad one byte.
                jmp     knob_3CW        ;  Knob 3 clockwise handler (trigger level increase).
                nop                     ;  Pad one byte.
                jmp     knob_3CCW       ;  Knob 3 counterclockwise handler (trigger level decrease).
                nop                     ;  Pad one byte.
                jmp     knob_2CW        ;  Knob 2 clockwise handler (vertical increase).
                nop                     ;  Pad one byte.
                jmp     knob_2CCW       ;  Knob 2 counterclockwise handler (vertical decrease).
                nop                     ;  Pad one byte.
                jmp     knob_1CW        ;  Knob 1 clockwise handler (horizontal increase).
                nop                     ;  Pad one byte.
                jmp     knob_1CCW       ;  Knob 1 counterclockwise handler (horizontal decrease).
                nop                     ;  Pad one byte.
                jmp     knob_0CW        ;  Knob 0 clockwise handler (cursor increase).
                nop                     ;  Pad one byte.
                jmp     knob_0CCW       ;  Knob 0 counterclockwise handler (cursor decrease).
;-----------------------------------------------------------------------------------------------



;-----------------------------------------------------------------------------------------------
;Cursor knob toggle key state machine
key_0:          lda     LED_bank_D      ;  Get the status of the cursor LEDs.
                coma                    ;  Invert A.
                and     #$07            ;  Remove insignificant bits.
                beq     key0_done       ;  Get-out if cursors are not active.

                ldhx    #key_0          ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key0_next       ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key0_next       ;  Skip until finished waiting for data.

do_key_0:       lda     temp_state      ;  Check the temporary state variable.
                bne     key0_st1        ;  If it is non-zero, the inquiry command is done.

;State 0: Query selected cursor based on cursor mode (hbar, vbar, paired).
key0_st0:       inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit.
                brclr   0,LED_bank_D,inq_hbar    ;  Inquire active HBAR cursor if HBARs are active.
                brclr   1,LED_bank_D,inq_vbar    ;  Inquire active VBAR cursor if VBARs are active.
inq_Pbar:       ldhx    #msg_actP_inq   ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key0_next       ;  Return to main_loop until next time through.
inq_Hbar:       ldhx    #msg_actH_inq   ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key0_next       ;  Return to main_loop until next time through.
inq_Vbar:       ldhx    #msg_actV_inq   ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key0_next       ;  Return to main_loop until next time through.

;State 1: Get the currently selected cursor and select the other one.
key0_st1:       lda     state_inq       ;  Load the inquiry request.
                beq     curs_toggle     ;  Process the key if this is not an inquiry.
```

```
                lda     inbuf+6          ;  Load the 7th character of inbuf (CURSORX).
                sta     curs_active      ;  Store it as the active cursor.
                bra     key0_done        ;  Done with the inquiry.

curs_toggle:    lda     inbuf+6          ;  Load the 7th character of inbuf (CURSORX).
                cmp     #'2'             ;  See if the active cursor is 2.
                beq     curs2_toggle     ;  Toggle to cursor 1 if so.

curs1_toggle:   lda     #'2'             ;  Load the active cursor number.
                sta     curs_active      ;  Store it as the active cursor.
                brclr   0,LED_bank_D,curs1H_tog    ;  Toggle to HBAR2.
                brclr   1,LED_bank_D,curs1V_tog    ;  Toggle to VBAR2.
curs1P_tog:     ldhx    #msg_curs1P_tog  ;  Point H:X at the set cursor message.
                sthx    out_data_ptr     ;  Store it in the output data pointer.
                bra     key0_done        ;  Return to main_loop until next time through.
curs1H_tog:     ldhx    #msg_curs1H_tog  ;  Point H:X at the set cursor message.
                sthx    out_data_ptr     ;  Store it in the output data pointer.
                bra     key0_done        ;  Return to main_loop until next time through.
curs1V_tog:     ldhx    #msg_curs1V_tog  ;  Point H:X at the set cursor message.
                sthx    out_data_ptr     ;  Store it in the output data pointer.
                bra     key0_done        ;  Return to main_loop until next time through.

curs2_toggle:   lda     #'1'             ;  Load the active cursor number.
                sta     curs_active      ;  Store it as the active cursor.
                brclr   0,LED_bank_D,curs2H_tog    ;  Toggle to HBAR1.
                brclr   1,LED_bank_D,curs2V_tog    ;  Toggle to VBAR1.
curs2P_tog:     ldhx    #msg_curs2P_tog  ;  Point H:X at the set cursor message.
                sthx    out_data_ptr     ;  Store it in the output data pointer.
                bra     key0_done        ;  Return to main_loop until next time through.
curs2H_tog:     ldhx    #msg_curs2H_tog  ;  Point H:X at the set cursor message.
                sthx    out_data_ptr     ;  Store it in the output data pointer.
                bra     key0_done        ;  Return to main_loop until next time through.
curs2V_tog:     ldhx    #msg_curs2V_tog  ;  Point H:X at the set cursor message.
                sthx    out_data_ptr     ;  Store it in the output data pointer.

key0_done:      clr     waiting          ;  Clear the waiting bit.
                clr     temp_state       ;  Reset the temporary state variable.
                bclr    5,keystate+!0    ;  Clear the action required bit for the key.
                ldhx    #do_process      ;  Get the address of the key processing routine.
                sthx    key_vector       ;  Make sure it runs next time since this one is complete.
key0_next:      jmp     do_key_return    ;  Restore registers and return to main_loop.

msg_actH_inq:   db      'CURS:HBA:SEL?',!10,0
msg_actV_inq:   db      'CURS:VBA:SEL?',!10,0
msg_actP_inq:   db      'CURS:PAI:SEL?',!10,0

msg_curs1H_tog: db      'CURS:HBA:SEL CURSOR2',!10,0
msg_curs1V_tog: db      'CURS:VBA:SEL CURSOR2',!10,0
msg_curs1P_tog: db      'CURS:PAI:SEL CURSOR2',!10,0

msg_curs2H_tog: db      'CURS:HBA:SEL CURSOR1',!10,0
msg_curs2V_tog: db      'CURS:VBA:SEL CURSOR1',!10,0
msg_curs2P_tog: db      'CURS:PAI:SEL CURSOR1',!10,0
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Horizontal knob toggle (not a state machine because no data is exchanged with the scope)

key_1:          lda     LED_bank_C       ;  Load the horizontal knob LEDs.
                eor     #$03             ;  Toggle the LEDs.
                sta     LED_bank_C       ;  Update the LED byte.

key1_done:      bclr    5,keystate+!1    ;  Clear the action required bit for the key.
                ldhx    #do_process      ;  Get the address of the key processing routine.
                sthx    key_vector       ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return    ;  Restore registers and return to main_loop.
;-----------------------------------------------------------------------------------------------
```

```
;-------------------------------------------------------------------------------------------
;Vertical knob toggle (not a state machine because no data is exchanged with the scope)

key_2:          lda     LED_bank_C      ;  Load the vertical knob LEDs.
                eor     #$0c            ;  Toggle the LEDs.
                sta     LED_bank_C      ;  Update the LED byte.

key2_done:      bclr    5,keystate+!2   ;  Clear the action required bit for the key.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return   ;  Restore registers and return to main_loop.
;-------------------------------------------------------------------------------------------

;-------------------------------------------------------------------------------------------
;Trigger knob (force trigger) state machine

key_3:          ldhx    #key_3          ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key3_next       ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key3_next       ;  Skip until finished waiting for data.

dokey_3: ldhx   #msg_force_trig         ;  Point H:X at the force trigger message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.

key3_done:      bclr    5,keystate+!3   ;  Clear the action required bit for the key.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
key3_next:      jmp     do_key_return   ;  Restore registers and return to main_loop.

msg_force_trig: db      'TRIG FORC',!10,0
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;Realtime/persist (accumulate) state machine

key_4:          ldhx    #key_4          ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key4_next       ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key4_next       ;  Skip until finished waiting for data.

dokey_4: lda    temp_state              ;  Check the temporary state variable.
                bne     key4_st1        ;  If it is non-zero, the inquiry command is done.

;State 0: Query display mode
key4_st0:       ldhx    #msg_disp_inq   ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit
                bra     key4_next       ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key4_st1:       lda     state_inq       ;  Load the inquiry request.
                beq     accum_set       ;  Process the key if this is not an inquiry.
                lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'A'            ;  See if scope is in accumulate mode.
                beq     accum_LED       ;  If so, set the accumulate LED.
realtime_LED:   bset    7,LED_bank_A    ;  Extinguish the ACCUMULATE LED.
                bclr    6,LED_bank_A    ;  Light the REAL TIME LED.
                bra     key4_done       ;  Done with the inquiry.

accum_LED:      bclr    7,LED_bank_A    ;  Light the ACCUMULATE LED.
                bset    6,LED_bank_A    ;  Extinguish the REAL TIME LED.
                bra     key4_done       ;  Done with the inquiry.

;Set the display mode.
```

```
accum_set:      lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'A'            ;  See if scope is already in accumulate mode.
                beq     accum_off       ;  If so, toggle into the non-accumulate mode.
                cmp     #'V'            ;  See if scope is in non-accumulate vectors mode.
                beq     accum_on_v      ;  If so, toggle into accumulate vectors mode.

accum_on_d:     bset    6,LED_bank_A    ;  Extinguish 'REAL TIME' LED.
                bclr    7,LED_bank_A    ;  Light 'PERSIST' LED.
                ldhx    #msg_a_dots     ;  Point H:X to the set dots (with accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key4_done       ;  The response to key 4 is complete.

accum_on_v:     bset    6,LED_bank_A    ;  Extinguish 'REAL TIME' LED.
                bclr    7,LED_bank_A    ;  Light 'PERSIST' LED.
                ldhx    #msg_a_vectors  ;  Point H:X to the set vectors (with accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key4_done       ;  The response to key 4 is complete.

accum_off:      lda     inbuf+5         ;  Get the ACCUMV or ACCUMD character.
                cmp     #'V'            ;  See if scope is in accumulate vectors mode.
                beq     accum_off_v     ;  If so, toggle into real time vectors mode.  Else real time dots.

accum_off_d:    bset    7,LED_bank_A    ;  Extinguish 'PERSIST' LED.
                bclr    6,LED_bank_A    ;  Light 'REAL TIME' LED.
                ldhx    #msg_dots       ;  Point H:X to the set dots (no accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key4_done       ;  The response to key 4 is complete.

accum_off_v:    bset    7,LED_bank_A    ;  Extinguish 'PERSIST' LED.
                bclr    6,LED_bank_A    ;  Light 'REAL TIME' LED.
                ldhx    #msg_vectors    ;  Point H:X to the set vectors (no accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.

key4_done:      clr     waiting         ;  Clear the waiting bit.
                clr     temp_state      ;  Reset the temporary state variable.
                bclr    5,keystate+!4   ;  Clear the action required bit for the key.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
key4_next:      jmp     do_key_return   ;  Restore registers and return to main_loop.

;Key 4 command strings are shared with those of key 5.
;-------------------------------------------------------------------------------------------



;-------------------------------------------------------------------------------------------
;Vectors/dots button state machine

key_5:          ldhx    #key_5          ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key5_next       ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key5_next       ;  Skip until finished waiting for data.

do_key_5:       lda     temp_state      ;  Check the temporary state variable.
                bne     key5_st1        ;  If it is non-zero, the inquiry command is done.

;State 0: Query display mode.
key5_st0:       ldhx    #msg_disp_inq   ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit
                bra     key5_next       ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key5_st1:       lda     state_inq       ;  Load the inquiry request.
                beq     disp_set        ;  Process the key if this is not an inquiry.
                lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'V'            ;  See if scope is in vectors mode.
```

```
                beq     vectors_LED     ;  If so, set the vectors LED.
                lda     inbuf+5         ;  Load the ACCUMV or ACCUMD character.
                cmp     #'V'            ;  See if the scope is in accumulate vectors mode.
                beq     vectors_LED     ;  If so, set the vectors LED, else set the dots LED.

dots_LED:       bset    4,LED_bank_A    ;  Extinguish the VECTORS LED.
                bclr    5,LED_bank_A    ;  Light the DOTS LED.
                bra     key5_done       ;  Done with the inquiry.

vectors_LED:    bclr    4,LED_bank_A    ;  Light the VECTORS LED.
                bset    5,LED_bank_A    ;  Extinguish the DOTS LED.
                bra     key5_done       ;  Done with the inquiry.

disp_set:       lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'A'            ;  See if scope is in accumulate mode.
                beq     disp_accum      ;  If so, the command will need to include the accumulate command.
                cmp     #'V'            ;  See is scope is in vectors mode.
                beq     disp_dots       ;  Toggle into dots mode if so.

disp_vectors:   bset    5,LED_bank_A    ;  Extinguish 'DOTS' LED.
                bclr    4,LED_bank_A    ;  Light 'VECTORS' LED.
                ldhx    #msg_vectors    ;  Point H:X to the set vectors (no accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key5_done       ;  The response to key 5 is complete.

disp_dots:      bset    4,LED_bank_A    ;  Extinguish 'VECTORS' LED.
                bclr    5,LED_bank_A    ;  Light 'DOTS' LED.
                ldhx    #msg_dots       ;  Point H:X to the set dots (no accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key5_done       ;  The response to key 5 is complete.

disp_accum:     lda     inbuf+5         ;  Get the ACCUMV or ACCUMD character.
                cmp     #'V'            ;  See if scope is in accumulate vectors mode.
                beq     disp_a_dots     ;  If so, toggle into accumulate dots mode.  Else accumulate vectors.

disp_a_vectors: bset    5,LED_bank_A    ;  Extinguish 'DOTS' LED.
                bclr    4,LED_bank_A    ;  Light 'VECTORS' LED.
                ldhx    #msg_a_vectors  ;  Point H:X to the set vectors (with accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key5_done       ;  The response to key 5 is complete.

disp_a_dots:    bset    4,LED_bank_A    ;  Extinguish 'VECTORS' LED.
                bclr    5,LED_bank_A    ;  Light 'DOTS' LED.
                ldhx    #msg_a_dots     ;  Point H:X to the set dots (with accumulate) command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.

key5_done:      clr     waiting         ;  Clear the waiting bit.
                clr     temp_state      ;  Reset the temporary state variable.
                bclr    5,keystate+!5   ;  Clear the action required bit for the key.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
key5_next:      jmp     do_key_return   ;  Restore registers and return to main_loop.

msg_vectors:    db      'DIS:STY VEC',!10,0
msg_dots:       db      'DIS:STY DOT',!10,0
msg_a_vectors:  db      'DIS:STY ACCUMV',!10,0
msg_a_dots:     db      'DIS:STY ACCUMD',!10,0
msg_disp_inq:   db      'DIS:STY?',!10,0
;-----------------------------------------------------------------------------------------------



;-----------------------------------------------------------------------------------------------
;Sample/average button state machine

key_6:          ldhx    #key_6          ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key6_next       ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
```

```
                bne     key6_next       ;  Skip until finished waiting for data.

do_key_6:       lda     temp_state      ;  Check the temporary state variable.
                bne     key6_st1        ;  If it is non-zero, the inquiry command is done.

;State 0: Query sample mode.
key6_st0:       ldhx    #msg_mode_inq   ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit
                bra     key6_next       ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key6_st1:       lda     state_inq       ;  Load the inquiry request.
                beq     mode_set        ;  Process the key if this is not an inquiry.
                lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'A'            ;  See if scope is in average mode.
                beq     average_LED     ;  If so, set the average LED.

sample_LED:     bset    3,LED_bank_A    ;  Extinguish the AVERAGE LED.
                bclr    2,LED_bank_A    ;  Light the SAMPLE LED.
                bra     key6_done       ;  Done with the inquiry.

average_LED:    bclr    3,LED_bank_A    ;  Light the AVERAGE LED.
                bset    4,LED_bank_A    ;  Extinguish the SAMPLE LED.
                bra     key6_done       ;  Done with the inquiry.

mode_set:       lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'S'            ;  See if scope is in sample mode.
                beq     mode_average    ;  Toggle into average if so, else go into sample mode.

mode_sample:    bset    3,LED_bank_A    ;  Extinguish 'AVERAGE' LED.
                bclr    2,LED_bank_A    ;  Light 'SAMPLE' LED.
                ldhx    #msg_sample     ;  Point H:X to the set sample mode command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key6_done       ;  The response to key 9 is complete.

mode_average:   bset    2,LED_bank_A    ;  Extinguish 'SAMPLE' LED.
                bclr    3,LED_bank_A    ;  Light 'AVERAGE' LED.
                ldhx    #msg_average    ;  Point H:X to the set average mode command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.

key6_done:      clr     waiting         ;  Clear the waiting bit.
                clr     temp_state      ;  Reset the temporary state variable.
                bclr    5,keystate+!6   ;  Clear the action required bit for the key.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
key6_next:      jmp     do_key_return   ;  Restore registers and return to main_loop.

msg_sample:     db      'ACQ:MOD SAM',!10,0
msg_average:    db      'ACQ:MOD AVE',!10,0
msg_mode_inq:   db      'ACQ:MOD?',!10,0
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Continuous/singleshot button state machine

key_7:          ldhx    #key_7          ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key7_next       ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key7_next       ;  Skip until finished waiting for data.


do_key_7:       lda     temp_state      ;  Check the temporary state variable.
                bne     key7_st1        ;  If it is non-zero, the inquiry command is done.

;State 0: Query sample mode.
```

```
key7_st0:         ldhx     #msg_stop_inq     ;  Point H:X at the inquiry message.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bset     0,waiting         ;  Set the waiting bit
                  bra      key7_next         ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key7_st1:         lda      state_inq         ;  Load the inquiry request.
                  beq      single_set        ;  Process the key if this is not an inquiry.
                  lda      inbuf             ;  Load the first character of inbuf.
                  cmp      #'S'              ;  See if scope is in single shot mode.
                  beq      average_LED       ;  If so, set the average LED.

cont_LED:         bset     1,LED_bank_A      ;  Extinguish the SINGLE SHOT LED.
                  bclr     0,LED_bank_A      ;  Light the CONTINUOUS LED.
                  bra      key7_done         ;  Done with the inquiry.

single_LED:       bclr     1,LED_bank_A      ;  Light the SINGLE SHOT LED.
                  bset     0,LED_bank_A      ;  Extinguish the CONTINUOUS LED.
                  bra      key7_done         ;  Done with the inquiry.

single_set:       lda      inbuf             ;  Load the first character of inbuf.
                  cmp      #'R'              ;  See if scope is in continuous mode.
                  beq      single_cmd        ;  Toggle singleshot if so.

continuous_cmd:   bset     1,LED_bank_A      ;  Extinguish 'SINGLE SHOT' LED.
                  bclr     0,LED_bank_A      ;  Light 'CONTINUOUS' LED.
                  ldhx     #msg_continuous   ;  Point H:X to the set continuous command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  bra      key7_done         ;  The response to key 7 is complete.

single_cmd:       bset     0,LED_bank_A      ;  Extinguish 'CONTINUOUS' LED.
                  bclr     1,LED_bank_A      ;  Light 'SINGLE SHOT' LED.
                  ldhx     #msg_singleshot   ;  Point H:X to the set singleshot command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.

key7_done:        clr      waiting           ;  Clear the waiting bit.
                  clr      temp_state        ;  Reset the temporary state variable.
                  bclr     5,keystate+!7     ;  Clear the action required bit for the key.
                  ldhx     #do_process       ;  Get the address of the key processing routine.
                  sthx     key_vector        ;  Make sure it runs next time since this one is complete.
key7_next:        jmp      do_key_return     ;  Restore registers and return to main_loop.

msg_continuous:   db       'ACQ:STOPA RUNST',!10,0
msg_singleshot:   db       'ACQ:STOPA SEQ',!10,0
msg_stop_inq:     db       'ACQ:STOPA?',!10,0
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Channel 1 coupling button state machine

key_8:            ldhx     #key_8            ;  Get the address of this routine.
                  sthx     key_vector        ;  Indicate that this routine is currently running.
                  lda      out_data_ptr      ;  See if there is RS-232 output data pending.
                  bne      key8_next         ;  Skip until 232 output data is done.
                  lda      waiting           ;  See if we are waiting for reply data.
                  bne      key8_next         ;  Skip until finished waiting for data.


do_key_8:         lda      temp_state        ;  Check the temporary state variable.
                  bne      key8_st1          ;  If it is non-zero, the inquiry command is done.

;State 0: Query channel 1 coupling mode.
key8_st0:         ldhx     #msg_coup1_inq    ;  Point H:X at the inquiry message.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bset     0,waiting         ;  Set the waiting bit
                  bra      key8_next         ;  Return to main_loop until next time through.
```

```
;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key8_st1:         lda      state_inq         ;  Load the inquiry request.
                  beq      coup1_set         ;  Process the key if this is not an inquiry.
                  lda      inbuf             ;  Load the first character of inbuf.
                  cmp      #'A'              ;  See if channel is AC coupled.
                  beq      AC1_LED           ;  If so, set the AC LED.
                  cmp      #'D'              ;  See if channel is DC coupled.
                  beq      DC1_LED           ;  If so, set the DC LED, else set GND LED.

GND1_LED:         bclr     5,LED_bank_C      ;  Light the CH1 GND LED.
                  bset     7,LED_bank_C      ;  Extinguish the CH1 DC LED.
                  bset     6,LED_bank_B      ;  Extinguish the CH1 AC LED.
                  bra      key8_done         ;  Done with the inquiry.

AC1_LED: bset     5,LED_bank_C              ;  Extinguish the CH1 GND LED.
                  bset     7,LED_bank_C      ;  Extinguish the CH1 DC LED.
                  bclr     6,LED_bank_B      ;  Light the CH1 AC LED.
                  bra      key8_done         ;  Done with the inquiry.

DC1_LED: bset     5,LED_bank_C              ;  Extinguish the CH1 GND LED.
                  bclr     7,LED_bank_C      ;  Light the CH1 DC LED.
                  bset     6,LED_bank_B      ;  Extinguish the CH1 AC LED.
                  bra      key8_done         ;  Done with the inquiry.

coup1_set:        lda      inbuf             ;  Load the first character of inbuf.
                  cmp      #'A'              ;  See if coupling is AC now.
                  beq      coup1_set_DC      ;  Toggle to DC is so.
                  cmp      #'D'              ;  See if coupling if DC.
                  beq      coup1_set_GND     ;  Toggle to GND if so.

coup1_set_AC:     bclr     6,LED_bank_B      ;  Light 'AC' LED.
                  bset     7,LED_bank_C      ;  Extinguish 'DC' LED.
                  bset     5,LED_bank_C      ;  Extinguish 'GND' LED.
                  ldhx     #msg_coup1_AC     ;  Point H:X to the set AC coupling command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  bra      key8_done         ;  The response to key 8 is complete.

coup1_set_DC:     bset     6,LED_bank_B      ;  Extinguish 'AC' LED.
                  bclr     7,LED_bank_C      ;  Light 'DC' LED.
                  bset     5,LED_bank_C      ;  Extinguish 'GND' LED.
                  ldhx     #msg_coup1_DC     ;  Point H:X to the set AC coupling command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  bra      key8_done         ;  The response to key 8 is complete.

coup1_set_GND:    bset     6,LED_bank_B      ;  Extinguish 'AC' LED.
                  bset     7,LED_bank_C      ;  Extinguish 'DC' LED.
                  bclr     5,LED_bank_C      ;  Light 'GND' LED.
                  ldhx     #msg_coup1_GND    ;  Point H:X to the set AC coupling command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.

key8_done:        clr      waiting           ;  Clear the waiting bit.
                  clr      temp_state        ;  Reset the temporary state variable.
                  bclr     5,keystate+!8     ;  Clear the action required bit for the key.
                  ldhx     #do_process       ;  Get the address of the key processing routine.
                  sthx     key_vector        ;  Make sure it runs next time since this one is complete.
key8_next:        jmp      do_key_return     ;  Restore registers and return to main_loop.

msg_coup1_inq:    db       'CH1:COUP?',!10,0
msg_coup1_AC:     db       'CH1:COUPL AC',!10,0
msg_coup1_DC:     db       'CH1:COUPL DC',!10,0
msg_coup1_GND:    db       'CH1:COUPL GND',!10,0
;-----------------------------------------------------------------------------------------------



;-----------------------------------------------------------------------------------------------
;Trigger source button state machine

key_9:            ldhx     #key_9            ;  Get the address of this routine.
                  sthx     key_vector        ;  Indicate that this routine is currently running.
```

```
                lda     out_data_ptr     ;  See if there is RS-232 output data pending.
                bne     key9_next        ;  Skip until 232 output data is done.
                lda     waiting          ;  See if we are waiting for reply data.
                bne     key9_next        ;  Skip until finished waiting for data.

do_key_9:       lda     temp_state       ;  Check the temporary state variable.
                bne     key9_st1         ;  If it is non-zero, the inquiry command is done.

;State 0: Query trigger source.
key9_st0:       ldhx    #msg_source_inq  ;  Point H:X at the inquiry message.
                sthx    out_data_ptr     ;  Store it in the output data pointer.
                inc     temp_state       ;  Advance the state variable.
                bset    0,waiting        ;  Set the waiting bit
                bra     key9_next        ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key9_st1:       lda     state_inq        ;  Load the inquiry request.
                beq     source_set       ;  Process the key if this is not an inquiry.
                lda     inbuf+2          ;  Load the third character of inbuf.
                cmp     #'2'             ;  See if channel 2 is the source.
                beq     source_2_LED     ;  If so, set the CHANNEL 2 LED.
                cmp     #'1'             ;  See if channel 1 is the source.
                beq     source_1_LED     ;  If so, set the CHANNEL 1 LED, else indicate DMM mode.

not_1or2_LED:   bset    5,LED_bank_B     ;  Extinguish the CH2 SOURCE LED.
                bset    4,LED_bank_B     ;  Extinguish the CH1 SOURCE LED.
                bra     key9_done        ;  Done with the inquiry.

source_1_LED:   bset    5,LED_bank_B     ;  Extinguish the CH2 SOURCE LED.
                bclr    4,LED_bank_B     ;  Light the CH1 SOURCE LED.
                bra     key9_done        ;  Done with the inquiry.

source_2_LED:   bclr    5,LED_bank_B     ;  Light the CH2 SOURCE LED.
                bset    4,LED_bank_B     ;  Extinguish the CH1 SOURCE LED.
                bra     key9_done        ;  Done with the inquiry.

source_set:     lda     inbuf+2          ;  Load the third character of inbuf.
                cmp     #'1'             ;  See if the current source is channel 1.
                beq     source_2         ;  Toggle to channel 2 if so.
source_1:       bset    5,LED_bank_B     ;  Extinguish 'CH2' LED.
                bclr    4,LED_bank_B     ;  Light 'CH1' LED.
                ldhx    #msg_source_1    ;  Point H:X to the set channel 1 command.
                sthx    out_data_ptr     ;  Store it in the output data pointer.
                bra     key9_done        ;  The response to key 9 is complete.

source_2:       bset    4,LED_bank_B     ;  Extinguish 'CH1' LED.
                bclr    5,LED_bank_B     ;  Light 'CH2' LED.
                ldhx    #msg_source_2    ;  Point H:X to the set channel 2 command.
                sthx    out_data_ptr     ;  Store it in the output data pointer.

key9_done:      clr     waiting          ;  Clear the waiting bit.
                clr     temp_state       ;  Reset the temporary state variable.
                bclr    5,keystate+!9    ;  Clear the action required bit for the key.
                ldhx    #do_process      ;  Get the address of the key processing routine.
                sthx    key_vector       ;  Make sure it runs next time since this one is complete.
key9_next:      jmp     do_key_return    ;  Restore registers and return to main_loop.

msg_source_1:   db      'TRIG:MAI:EDGE:SOU CH1',!10,0
msg_source_2:   db      'TRIG:MAI:EDGE:SOU CH2',!10,0
msg_source_inq: db      'TRIG:MAI:EDGE:SOU?',!10,0
;-----------------------------------------------------------------------------------------------



;-----------------------------------------------------------------------------------------------
;Trigger slope button state machine

key_10:         ldhx    #key_10          ;  Get the address of this routine.
                sthx    key_vector       ;  Indicate that this routine is currently running.
                lda     out_data_ptr     ;  See if there is RS-232 output data pending.
```

```
                bne     key10_next      ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key10_next      ;  Skip until finished waiting for data.

do_key_10:      lda     temp_state      ;  Check the temporary state variable.
                bne     key10_st1       ;  If it is non-zero, the inquiry command is done.

;State 0: Query trigger slope.
key10_st0:      ldhx    #msg_slope_inq  ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit
                bra     key10_next      ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key10_st1:      lda     state_inq       ;  Load the inquiry request.
                beq     slope_set       ;  Process the key if this is not an inquiry.
                lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'R'            ;  See if scope is in rising edge trigger mode.
                beq     rising_LED      ;  If so, set the rising LED.

falling_LED:    bset    1,LED_bank_B    ;  Extinguish the RISING edge LED.
                bclr    0,LED_bank_B    ;  Light the FALLING edge LED.
                bra     key10_done      ;  Done with the inquiry.

rising_LED:     bclr    1,LED_bank_B    ;  Light the RISING edge LED.
                bset    0,LED_bank_B    ;  Extinguish the FALLING edge LED.
                bra     key10_done      ;  Done with the inquiry.

slope_set:      lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'F'            ;  See if the current state is 'FALL'.
                beq     slope_rise      ;  Toggle to 'RISE' if so.
slope_fall:     bset    1,LED_bank_B    ;  Extinguish 'RISE' LED.
                bclr    0,LED_bank_B    ;  Light 'FALL' LED.
                ldhx    #msg_slope_neg  ;  Point H:X to the set falling edge command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     key10_done      ;  The response to key 10 is complete.

slope_rise:     bset    0,LED_bank_B    ;  Extinguish 'FALL' LED.
                bclr    1,LED_bank_B    ;  Light 'RISE' LED.
                ldhx    #msg_slope_pos  ;  Point H:X to the set rising edge command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.

key10_done:     clr     waiting         ;  Clear the waiting bit.
                clr     temp_state      ;  Reset the temporary state variable.
                bclr    5,keystate+!10  ;  Clear the action required bit for the key.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
key10_next:     jmp     do_key_return   ;  Restore registers and return to main_loop.

msg_slope_pos:  db      'TRIG:MAI:EDGE:SLO RIS',!10,0
msg_slope_neg:  db      'TRIG:MAI:EDGE:SLO FALL',!10,0
msg_slope_inq:  db      'TRIG:MAI:EDGE:SLO?',!10,0
;-----------------------------------------------------------------------------------------------



;-----------------------------------------------------------------------------------------------
;Auto/normal trigger button state machine

key_11:
                ldhx    #key_11         ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key11_next      ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key11_next      ;  Skip until finished waiting for data.

do_key_11:      lda     temp_state      ;  Check the temporary state variable.
                bne     key11_st1       ;  If it is non-zero, the inquiry command is done.
```

```
;State 0: Query trigger mode.
key11_st0:        ldhx     #msg_trig_inq     ;  Point H:X at the inquiry message.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  mov      #$01,temp_state   ;  Advance the state variable.
                  bset     0,waiting         ;  Set the waiting bit
                  bra      key11_next        ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key11_st1:        lda      state_inq         ;  Load the inquiry request.
                  beq      trig_set          ;  Process the key if this is not an inquiry.
                  lda      inbuf             ;  Load the first character of inbuf.
                  cmp      #'N'              ;  See if scope is normal trigger mode.
                  beq      normal_LED        ;  If so, set the normal LED.

auto_LED:         bclr     3,LED_bank_B      ;  Light the AUTO LED.
                  bset     2,LED_bank_B      ;  Extinguish the NORMAL LED.
                  bra      key11_done        ;  Done with the inquiry.

normal_LED:       bset     3,LED_bank_B      ;  Extinguish the AUTO LED.
                  bclr     2,LED_bank_B      ;  Light the NORMAL LED.
                  bra      key11_done        ;  Done with the inquiry.

trig_set:         lda      inbuf             ;  Load the first character of inbuf.
                  cmp      #'N'              ;  See if the current state is 'NORM'.
                  beq      trig_auto         ;  Toggle to 'AUTO' if so.
trig_norm:        bset     3,LED_bank_B      ;  Extinguish 'AUTO' LED.
                  bclr     2,LED_bank_B      ;  Light 'NORM' LED.
                  ldhx     #msg_norm         ;  Point H:X to the set normal command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  bra      key11_done        ;  The response to key 11 is complete.

trig_auto:        bset     2,LED_bank_B      ;  Extinguish 'NORM' LED.
                  bclr     3,LED_bank_B      ;  Light 'AUTO' LED.
                  ldhx     #msg_auto         ;  Point H:X to the set auto command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.

key11_done:       clr      waiting           ;  Clear the waiting bit.
                  clr      temp_state        ;  Reset the temporary state variable.
                  bclr     5,keystate+!11    ;  Clear the action required bit for the key.
                  ldhx     #do_process       ;  Get the address of the key processing routine.
                  sthx     key_vector        ;  Make sure it runs next time since this one is complete.
key11_next:       jmp      do_key_return     ;  Restore registers and return to main_loop.

msg_auto:         db       'TRIG:MAI:MOD AUTO',!10,0
msg_norm:         db       'TRIG:MAI:MOD NORM',!10,0
msg_trig_inq:     db       'TRIG:MAI:MOD?',!10,0
;---------------------------------------------------------------------------------------------


;---------------------------------------------------------------------------------------------
;Unused
key_12:           jmp      do_key_return     ;  Restore registers and return to main_loop.
;---------------------------------------------------------------------------------------------


;---------------------------------------------------------------------------------------------
;Unused
key_13:           jmp      do_key_return     ;  Restore registers and return to main_loop.
;---------------------------------------------------------------------------------------------


;---------------------------------------------------------------------------------------------
;Cursor type select button state machine
key_14:           ldhx     #key_14           ;  Get the address of this routine.
                  sthx     key_vector        ;  Indicate that this routine is currently running.
                  lda      out_data_ptr      ;  See if there is RS-232 output data pending.
                  bne      key14_next        ;  Skip until 232 output data is done.
                  lda      waiting           ;  See if we are waiting for reply data.
                  bne      key14_next        ;  Skip until finished waiting for data.
```

```
do_key_14:        lda     temp_state        ;  Check the temporary state variable.
                  bne     key14_st1         ;  If it is non-zero, the inquiry command is done.

;State 0: Query trigger mode.
key14_st0:        ldhx    #msg_curs_inq     ;  Point H:X at the inquiry message.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  inc     temp_state        ;  Advance the state variable.
                  bset    0,waiting         ;  Set the waiting bit
                  bra     key14_next        ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key14_st1:        lda     state_inq         ;  Load the inquiry request.
                  beq     curs_set          ;  Process the key if this is not an inquiry.
                  lda     inbuf             ;  Load the first character of inbuf.
                  cmp     #'H'              ;  See if HBARS are active.
                  beq     HBAR_LED          ;  If so, set the HBARS LED.
                  cmp     #'V'              ;  See if VBARS are active.
                  beq     VBAR_LED          ;  If so, set the VBARS LED.
                  cmp     #'P'              ;  See if PAIRED cursors are active.
                  beq     PBAR_LED          ;  If so, set the HBARS LED, else set no LEDs.

no_curs_LED:      bset    0,LED_bank_D      ;  Extinguish HBARS LED.
                  bset    1,LED_bank_D      ;  Extinguish VBARS LED.
                  bset    2,LED_bank_D      ;  Extinguish PAIRED BARS LED.
                  bra     key14_done        ;  Done with the inquiry.

HBAR_LED:         bclr    0,LED_bank_D      ;  Light HBARS LED.
                  bset    1,LED_bank_D      ;  Extinguish VBARS LED.
                  bset    2,LED_bank_D      ;  Extinguish PAIRED BARS LED.
                  bra     key14_done        ;  Done with the inquiry.

key14_next:       jmp     do_key_return     ;  Restore registers and return to main_loop.

VBAR_LED:         bset    0,LED_bank_D      ;  Extinguish HBARS LED.
                  bclr    1,LED_bank_D      ;  Light VBARS LED.
                  bset    2,LED_bank_D      ;  Extinguish PAIRED BARS LED.
                  bra     key14_done        ;  Done with the inquiry.

PBAR_LED:         bset    0,LED_bank_D      ;  Extinguish HBARS LED.
                  bset    1,LED_bank_D      ;  Extinguish VBARS LED.
                  bclr    2,LED_bank_D      ;  Light PAIRED BARS LED.
                  bra     key14_done        ;  Done with the inquiry.

curs_set:         lda     inbuf             ;  Load the first character of inbuf.
                  cmp     #'O'              ;  See if the cursors are OFF.
                  beq     curs_set_H        ;  Activate HBARS if so.
                  cmp     #'H'              ;  See if HBARS are on.
                  beq     curs_set_V        ;  Activate VBARS if so.
                  cmp     #'V'              ;  See if VABRS are on.
                  beq     curs_set_P        ;  Activate PAIRED cursors if so.
curs_set_off:     bset    0,LED_bank_D      ;  Extinguish 'HBARS' LED.
                  bset    1,LED_bank_D      ;  Extinguish 'VBARS' LED.
                  bset    2,LED_bank_D      ;  Extinguish 'PAIRED' LED.
                  ldhx    #msg_curs_off     ;  Point H:X to the set cursors off command.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  bra     key14_done        ;  The response to key 14 is complete.

curs_set_H:       bclr    0,LED_bank_D      ;  Light 'HBARS' LED.
                  bset    1,LED_bank_D      ;  Extinguish 'VBARS' LED.
                  bset    2,LED_bank_D      ;  Extinguish 'PAIRED' LED.
                  ldhx    #msg_curs_hbar    ;  Point H:X to the set cursors HBARS command.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  bra     key14_done        ;  The response to key 14 is complete.

curs_set_V:       bset    0,LED_bank_D      ;  Extinguish 'HBARS' LED.
                  bclr    1,LED_bank_D      ;  Light 'VBARS' LED.
                  bset    2,LED_bank_D      ;  Extinguish 'PAIRED' LED.
                  ldhx    #msg_curs_vbar    ;  Point H:X to the set cursors VBARS command.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
```

```
                bra     key14_done      ;  The response to key 14 is complete.

curs_set_P:     bset    0,LED_bank_D    ;  Extinguish 'HBARS' LED.
                bset    1,LED_bank_D    ;  Extinguish 'VBARS' LED.
                bclr    2,LED_bank_D    ;  Light 'PAIRED' LED.
                ldhx    #msg_curs_pair  ;  Point H:X to the set cursors PAIRED command.
                sthx    out_data_ptr    ;  Store it in the output data pointer.

key14_done:     clr     waiting         ;  Clear the waiting bit.
                clr     temp_state      ;  Reset the temporary state variable.
                bclr    5,keystate+!14  ;  Clear the action required bit for the key.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return   ;  Restore registers and return to main_loop.

msg_curs_off:   db      'CURS:FUNC OFF',!10,0
msg_curs_hbar:  db      'CURS:FUNC HBA',!10,0
msg_curs_vbar:  db      'CURS:FUNC VBA',!10,0
msg_curs_pair:  db      'CURS:FUNC PAI',!10,0
msg_curs_inq:   db      'CURS:FUNC?',!10,0
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;Chanel 2 coupling button state machine
key_15:         ldhx    #key_15         ;  Get the address of this routine.
                sthx    key_vector      ;  Indicate that this routine is currently running.
                lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     key15_next      ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     key15_next      ;  Skip until finished waiting for data.

do_key_15:      lda     temp_state      ;  Check the temporary state variable.
                bne     key15_st1       ;  If it is non-zero, the inquiry command is done.

;State 0: Query channel 2 coupling mode.
key15_st0:      ldhx    #msg_coup2_inq  ;  Point H:X at the inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit
                bra     key15_next      ;  Return to main_loop until next time through.

;State 1: Determine if this is a master inquiry to set the LEDs or an actual set command.
key15_st1:      lda     state_inq       ;  Load the inquiry request.
                beq     coup2_set       ;  Process the key if this is not an inquiry.
                lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'A'            ;  See if channel is AC coupled.
                beq     AC2_LED         ;  If so, set the AC LED.
                cmp     #'D'            ;  See if channel is DC coupled.
                beq     DC2_LED         ;  If so, set the DC LED, else set GND LED.

GND2_LED:       bclr    4,LED_bank_C    ;  Light the CH2 GND LED.
                bset    6,LED_bank_C    ;  Extinguish the CH2 DC LED.
                bset    7,LED_bank_B    ;  Extinguish the CH2 AC LED.
                bra     key15_done      ;  Done with the inquiry.

AC2_LED: bset   4,LED_bank_C            ;  Extinguish the CH2 GND LED.
                bset    6,LED_bank_C    ;  Extinguish the CH2 DC LED.
                bclr    7,LED_bank_B    ;  Light the CH2 AC LED.
                bra     key15_done      ;  Done with the inquiry.

DC2_LED: bset   4,LED_bank_C            ;  Extinguish the CH2 GND LED.
                bclr    6,LED_bank_C    ;  Light the CH2 DC LED.
                bset    7,LED_bank_B    ;  Extinguish the CH2 AC LED.
                bra     key15_done      ;  Done with the inquiry.

coup2_set:      lda     inbuf           ;  Load the first character of inbuf.
                cmp     #'A'            ;  See if coupling is AC now.
                beq     coup2_set_DC    ;  Toggle to DC is so.
                cmp     #'D'            ;  See if coupling if DC.
```

```
                beq     coup2_set_GND     ;  Toggle to GND if so.

coup2_set_AC:   bclr    7,LED_bank_B      ;  Light 'AC' LED.
                bset    6,LED_bank_C      ;  Extinguish 'DC' LED.
                bset    4,LED_bank_C      ;  Extinguish 'GND' LED.
                ldhx    #msg_coup2_AC     ;  Point H:X to the set AC coupling command.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     key15_done        ;  The response to key 15 is complete.

coup2_set_DC:   bset    7,LED_bank_B      ;  Extinguish 'AC' LED.
                bclr    6,LED_bank_C      ;  Light 'DC' LED.
                bset    4,LED_bank_C      ;  Extinguish 'GND' LED.
                ldhx    #msg_coup2_DC     ;  Point H:X to the set AC coupling command.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     key15_done        ;  The response to key 15 is complete.

coup2_set_GND:  bset    7,LED_bank_B      ;  Extinguish 'AC' LED.
                bset    6,LED_bank_C      ;  Extinguish 'DC' LED.
                bclr    4,LED_bank_C      ;  Light 'GND' LED.
                ldhx    #msg_coup2_GND    ;  Point H:X to the set AC coupling command.
                sthx    out_data_ptr      ;  Store it in the output data pointer.

key15_done:     clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,keystate+!15    ;  Clear the action required bit for the key.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
key15_next:     jmp     do_key_return     ;  Restore registers and return to main_loop.

msg_coup2_inq:  db      'CH2:COUP?',!10,0
msg_coup2_AC:   db      'CH2:COUPL AC',!10,0
msg_coup2_DC:   db      'CH2:COUPL DC',!10,0
msg_coup2_GND:  db      'CH2:COUPL GND',!10,0
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Knob 3 clockwise state machine (increase trigger level)

knob_3CW:       ldhx    #knob_3CW         ;  Get the address of this routine.
                sthx    key_vector        ;  Indicate that this routine is currently running.
                lda     out_data_ptr      ;  See if there is RS-232 output data pending.
                bne     knob3CW_next      ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob3CW_next      ;  Skip until finished waiting for data.

do_knob3CW:     lda     temp_state        ;  Check the temporary state variable.
                beq     knob3CW_st0       ;  If it is non-zero, the inquiry command is done.
                cmp     #$01              ;  See if it is 1.
                beq     knob3CW_st1       ;  Go to state 1 (ask scope current scale).
                cmp     #$02              ;  See if it is 2.
                beq     knob3CW_st2       ;  Go to state 2 (ask scope current trigger level).
                bra     knob3CW_st3       ;  It must be state 3 (set new trigger level).

;State 0: Query trigger source channel vertical scale.
knob3CW_st0:    brclr   4,LED_bank_B,knob3CW_CH1   ;  Trigger source is channel 1 if LED is on.
                ldhx    #msg_sca2_inq     ;  Point H:X at the inquiry message.
                bra     knob3CW_inq       ;  Branch to send the inquiry message.
knob3CW_CH1:    ldhx    #msg_sca1_inq     ;  Point H:X at the inquiry message.
knob3CW_inq:    sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob3CW_next      ;  Return to main_loop until next time through.

;State 1: Convert scale data (in inbuf) to increment data and query present trigger position.
knob3CW_st1:    jsr     get_Y_inc         ;  Convert the floating point scale value to increment value.
                jsr     ascii2fixed       ;  Convert the increment value to a fixed point value.
                clrh                      ;  Clear H for IX1 addressing.
                ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob3CW_move:   lda     BCD_source-1,x    ;  Load the source data.
```

```
                sta     BCD_inc-1,x       ;  Store it in BCD_inc.
                dbnzx   knob3CW_move      ;  Loop until done.

                ldhx    #msg_lev_inq      ;  Point H:X at the inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob3CW_next      ;  Return to main_loop until next time through.

;State 2: Add the increment to the present level and send the new level to the scope.
knob3CW_st2:    jsr     ascii2fixed       ;  Convert the trigger level to fixed point BCD.
                lda     knob_inc_count+0  ;  Get the number of times to increment for this knob.
                sta     accel_temp        ;  Make a copy of it.

knob3CW_accel:  jsr     fixed_add         ;  Add the increment.
                ldx     #$09              ;  Initialize X to move 8 bytes.
accel3CW_loop:  lda     BCD_dest-1,x      ;  Load the result from the previous math operation.
                sta     BCD_source-1,x    ;  Store it back as the source.
                dbnzx   accel3CW_loop     ;  Loop until all BCD numbers have been moved.

                dec     accel_temp        ;  Decrement the temporary acceleration counter.
                bne     knob3CW_accel     ;  Continue to add the increment until done.

                jsr     BCD2float         ;  Convert the result to floating point ASCII.
                ldhx    #msg_lev_set      ;  Point H:X at the trigger level set command.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bra     knob3CW_next      ;  Return to main_loop until next time through.

;State 3: Send the new trigger value to the scope.
knob3CW_st3:    ldhx    #float_buf        ;  Point H:X at the floating point level value.
                sthx    out_data_ptr      ;  Store it in the output data pointer.

knob3CW_done:   clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+0     ;  Clear the action required bit for the key.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.

knob3CW_next:   jmp     do_key_return     ;  Restore registers and return to main_loop.

msg_lev_inq:    db      'TRIG:MAI:LEV?',!10,0
msg_lev_set:    db      'TRIG:MAI:LEV ',0
;Note: msg_sca1_inq and msg_sca2_inq are shared with knobs 1 and 2.
;-----------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------
;Knob 3 counterclockwise state machine (decrease trigger level)

knob_3CCW:      ldhx    #knob_3CCW        ;  Get the address of this routine.
                sthx    key_vector        ;  Indicate that this routine is currently running.
                lda     out_data_ptr      ;  See if there is RS-232 output data pending.
                bne     knob3CCW_next     ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob3CCW_next     ;  Skip until finished waiting for data.

do_knob3CCW:    lda     temp_state        ;  Check the temporary state variable.
                beq     knob3CCW_st0      ;  If it is non-zero, the inquiry command is done.
                cmp     #$01              ;  See if it is 1.
                beq     knob3CCW_st1      ;  Go to state 1 (ask scope current scale).
                cmp     #$02              ;  See if it is 2.
                beq     knob3CCW_st2      ;  Go to state 2 (ask scope current trigger level).
                bra     knob3CCW_st3      ;  It must be state 3 (set new trigger level).

;State 0: Query trigger source channel vertical scale.
knob3CCW_st0:   brclr   4,LED_bank_B,knob3CCW_CH1  ;  Trigger source is channel 1 if LED is on.
                ldhx    #msg_sca2_inq     ;  Point H:X at the inquiry message.
                bra     knob3CCW_inq      ;  Branch to send the inquiry message.
knob3CCW_CH1:   ldhx    #msg_sca1_inq     ;  Point H:X at the inquiry message.
```

```
knob3CCW_inq:     sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bset     0,waiting         ;  Set the waiting bit
                  bra      knob3CCW_next     ;  Return to main_loop until next time through.

;State 1: Convert scale data (in inbuf) to increment data and query present trigger position.
knob3CCW_st1:     jsr      get_Y_inc         ;  Convert the floating point scale value to increment value.
                  jsr      ascii2fixed       ;  Convert the increment value to a fixed point value.
                  clrh                       ;  Clear H for IX1 addressing.
                  ldx      #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob3CCW_move:    lda      BCD_source-1,x    ;  Load the source data.
                  sta      BCD_inc-1,x       ;  Store it in BCD_inc.
                  dbnzx    knob3CCW_move     ;  Loop until done.

                  ldhx     #msg_lev_inq      ;  Point H:X at the inquiry message.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bset     0,waiting         ;  Set the waiting bit
                  bra      knob3CCW_next     ;  Return to main_loop until next time through.

;State 2: Subtract the increment to the present level and send the new level to the scope.
knob3CCW_st2:     jsr      ascii2fixed       ;  Convert the trigger level to fixed point BCD.

                  lda      knob_inc_count+1  ;  Get the number of times to decrement for this knob.
                  sta      accel_temp        ;  Make a copy of it.

knob3CCW_accel:   jsr      fixed_sub         ;  Subtract the increment.
                  ldx      #$09              ;  Initialize X to move 8 bytes.
accel3CCW_loop:   lda      BCD_dest-1,x      ;  Load the result from the previous math operation.
                  sta      BCD_source-1,x    ;  Store it back as the source.
                  dbnzx    accel3CCW_loop    ;  Loop until all BCD numbers have been moved.

                  dec      accel_temp        ;  Decrement the temporary acceleration counter.
                  bne      knob3CCW_accel    ;  Continue to subtract the increment until done.

                  jsr      BCD2float         ;  Convert the result to floating point ASCII.
                  ldhx     #msg_lev_set      ;  Point H:X at the trigger level set command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bra      knob3CCW_next     ;  Return to main_loop until next time through.

;State 3: Send the new trigger value to the scope.
knob3CCW_st3:     ldhx     #float_buf        ;  Point H:X at the floating point level value.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.

knob3CCW_done:    clr      waiting           ;  Clear the waiting bit.
                  clr      temp_state        ;  Reset the temporary state variable.
                  bclr     5,knobstate+1     ;  Clear the action required bit for the key.
                  ldhx     #do_process       ;  Get the address of the key processing routine.
                  sthx     key_vector        ;  Make sure it runs next time since this one is complete.
knob3CCW_next:    jmp      do_key_return     ;  Restore registers and return to main_loop.

;Note: msg_lev_inq and msg_lev_set are shared with knob 3CW.
;Note: msg_sca1_inq and msg_sca2_inq are shared with knobs 1 and 2.
;-----------------------------------------------------------------------------------------


knob_2CW:         brclr    3,LED_bank_C,knob2_CWgain  ;  If the scale LED is lit, adjust the scale.
                  jmp      knob2_CWpos       ;  Else adjust the position.

;-----------------------------------------------------------------------------------------
;Knob 2 clockwise state machine for decreasing channel 1 vertical gain

knob2_CWgain:     ldhx     #knob2_CWgain     ;  Get the address of this routine.
                  sthx     key_vector        ;  Indicate that this routine is currently running.
                  lda      out_data_ptr      ;  See if there is RS-232 output data pending.
                  bne      knob2CWg_next     ;  Skip until 232 output data is done.
                  lda      waiting           ;  See if we are waiting for reply data.
                  bne      knob2CWg_next     ;  Skip until finished waiting for data.
```

```
do_knob2CWg:      lda     temp_state        ;  Check the temporary state variable.
                  beq     knob2CWg_st0      ;  If it is non-zero, the inquiry command is done.
                  cmp     #$01              ;  See if it is state 1.
                  beq     knob2CWg_st1      ;  Go to state 1 (the processing state).
                  bra     knob2CWg_st2      ;  It must be state 2 (the send data state).

;State 0: Query channel 1 vertical scale.
knob2CWg_st0:     ldhx    #msg_sca1_inq     ;  Point H:X at the inquiry message.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  inc     temp_state        ;  Advance the state variable.
                  bset    0,waiting         ;  Set the waiting bit
                  bra     knob2CWg_next     ;  Return to main_loop until next time through.

;State 1: Process the data and send the command string.
knob2CWg_st1:     inc     temp_state        ;  Advance the state variable.
                  lda     inbuf             ;  Get the first number in inbuf.
                  cmp     #'5'              ;  See if it is 5.
                  beq     knob2CWg_5        ;  Go to the next increment above 5.
                  cmp     #'2'              ;  See if it is 2.
                  beq     knob2CWg_2        ;  Go to the next increment above 2.
                  cmp     #'1'              ;  See if it is 1.
                  beq     knob2CWg_1        ;  Go to the next increment above 1.
                  bra     knob2CWg_done     ;  Do nothing if none of these numbers appear.
knob2CWg_5:       mov     #' ',inbuf        ;  Put a blank space as first character.
                  mov     #'1',inbuf+1      ;  Make 10 the mantissa.
                  mov     #'0',inbuf+2      ;  Make 10 the mantissa.  Inbuf now contains ' 10E+/-XX'.
                  bra     send_knob2CWg     ;  Send the set command string.
knob2CWg_2:       mov     #'5',inbuf        ;  Make 5 the first character.
                  bra     send_knob2CWg     ;  Send the set command string.  Inbuf now contains '5.0E+/-XX'.
knob2CWg_1:       mov     #'2',inbuf        ;  Make 2 the first character.  Inbuf now contains '2.0E+/-XX'.
send_knob2CWg:    ldhx    #msg_sca1_set     ;  Point H:X at the set message.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  bra     knob2CWg_next     ;  Return to main loop until next time through.

;State 2: Send the value string from inbuf after the command string is finished.
knob2CWg_st2:     clrh                      ;  Clear H for IX1 addressing.
                  ldx     #$06              ;  Initialize X with the number of characters to move from inbuf to float_buf.
knob2CWg_loop:    lda     inbuf-1,x         ;  Get the character from inbuf.
                  sta     float_buf-1,x     ;  Store it in float_buf.
                  dbnzx   knob2CWg_loop     ;  Loop until all relevant inbuf characters have been moved.
                  lda     float_buf+4       ;  Load the 5th character of inbuf.
                  cmp     #'-'              ;  See if it is a minus sign for the exponent.
                  beq     knob2CWg_term     ;  Terminate the string if so.
                  lda     #' '              ;  Load a space.
                  sta     float_buf+5       ;  Store it after the positive exponent.
knob2CWg_term:    lda     #!10              ;  Load the scope's termination character
                  sta     float_buf+6       ;  Store it at the end of the data string.
                  lda     #$00              ;  Load the null character.
                  sta     float_buf+7       ;  Terminate the string with the null character.
                  ldhx    #float_buf        ;  Point H:X at the new scale value.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.

knob2CWg_done:    clr     waiting           ;  Clear the waiting bit.
                  clr     temp_state        ;  Reset the temporary state variable.
                  bclr    5,knobstate+!2    ;  Clear the action required bit for the key.
                  ldhx    #do_process       ;  Get the address of the key processing routine.
                  sthx    key_vector        ;  Make sure it runs next time since this one is complete.
knob2CWg_next:    jmp     do_key_return     ;  Restore registers and return to main_loop.

msg_sca1_inq:     db      'CH1:SCA?',!10,0
msg_sca1_set:     db      'CH1:SCA ',0
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Knob 2 clockwise state machine for increasing channel 1 position

knob2_CWpos:      ldhx    #knob2_CWpos      ;  Get the address of this routine.
                  sthx    key_vector        ;  Indicate that this routine is currently running.
                  lda     out_data_ptr      ;  See if there is RS-232 output data pending.
```

```
                bne     knob2CWp_next     ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob2CWp_next     ;  Skip until finished waiting for data.

do_knob2CWp:    lda     temp_state        ;  Check the temporary state variable.
                beq     knob2CWp_st0      ;  If it is non-zero, the inquiry command is done.
                cmp     #$01              ;  See if it is 1.
                beq     knob2CWp_st1      ;  Go to state 1 (add increment and send the set command).
                bra     knob2CWp_st2      ;  It must be state 2 (set new position level).

;State 0: Query channel 1 vertical position.
knob2CWp_st0:   ldhx    #msg_pos1_inq     ;  Point H:X at the inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob2CWp_next     ;  Return to main_loop until next time through.

;State 1: Add .02 divisions times the acceleration value and send the set message.
knob2CWp_st1:   clrh                      ;  Clear H for IX1 addressing.
                ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob2CWp_clr:   clr     BCD_inc-1,x       ;  Clear the BCD_inc value.
                dbnzx   knob2CWp_clr      ;  Loop until done.  BCD_inc now contains zero.

                mov     #$02,BCD_inc+3    ;  Load the .02 division increment value into BCD_inc.

                jsr     ascii2fixed       ;  Convert the current position to fixed point BCD.
                lda     knob_inc_count+2  ;  Get the number of times to increment for this knob.
                sta     accel_temp        ;  Make a copy of it.

knob2CWp_accel: jsr     fixed_add         ;  Add the increment.
                ldx     #$09              ;  Initialize X to move 8 bytes.
accel2CWp_loop: lda     BCD_dest-1,x      ;  Load the result from the previous math operation.
                sta     BCD_source-1,x    ;  Store it back as the source.
                dbnzx   accel2CWp_loop    ;  Loop until all BCD numbers have been moved.

                dec     accel_temp        ;  Decrement the temporary acceleration counter.
                bne     knob2CWp_accel    ;  Continue to add the increment until done.

                jsr     BCD2float         ;  Convert the result to floating point ASCII.
                ldhx    #msg_pos1_set     ;  Point H:X at the channel level set command.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bra     knob2CWp_next     ;  Return to main_loop until next time through.

;State 2: Send the new channel 1 position value to the scope.
knob2CWp_st2:   ldhx    #float_buf        ;  Point H:X at the floating point level value.
                sthx    out_data_ptr      ;  Store it in the output data pointer.

knob2CWp_done:  clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+!2    ;  Clear the action required bit for the key.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.

knob2CWp_next:  jmp     do_key_return     ;  Restore registers and return to main_loop.

msg_pos1_inq:   db      'CH1:POS?',!10,0
msg_pos1_set:   db      'CH1:POS ',0
;-----------------------------------------------------------------------------------------------


knob_2CCW:      brclr   3,LED_bank_C,knob_2CCWgain ;  If the scale LED is lit, adjust the scale.
                jmp     knob2_CCWpos      ;  Else adjust the position.

;-----------------------------------------------------------------------------------------------

;Knob 2 counterclockwise state machine (increase channel 1 vertical gain)
knob_2CCWgain:  ldhx    #knob_2CCWgain    ;  Get the address of this routine.
                sthx    key_vector        ;  Indicate that this routine is currently running.
                lda     out_data_ptr      ;  See if there is RS-232 output data pending.
```

```
                bne     knob2CCWg_next    ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob2CCWg_next    ;  Skip until finished waiting for data.

do_knob2CCWg:   lda     temp_state        ;  Check the temporary state variable.
                beq     knob2CCWg_st0     ;  If it is non-zero, the inquiry command is done.
                cmp     #$01              ;  See if it is state 1.
                beq     knob2CCWg_st1     ;  Go to state 1 (the processing state).
                bra     knob2CCWg_st2     ;  It must be state 2 (the send data state).

;State 0: Query channel 1 vertical scale.
knob2CCWg_st0:  ldhx    #msg_sca1_inq     ;  Point H:X at the inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob2CCWg_next    ;  Return to main_loop until next time through.

;State 1: Process the data and send the command string.
knob2CCWg_st1:  inc     temp_state        ;  Advance the state variable.
                lda     inbuf             ;  Get the first number in inbuf.
                cmp     #'5'              ;  See if it is 5.
                beq     knob2CCW_5g       ;  Go to the next increment below 5.
                cmp     #'2'              ;  See if it is 2.
                beq     knob2CCW_2g       ;  Go to the next increment below 2.
                cmp     #'1'              ;  See if it is 1.
                beq     knob2CCW_1g       ;  Go to the next increment below 1.
                bra     knob2CCWg_done    ;  Do nothing if none of these numbers appear in inbuf.
knob2CCW_1g:    mov     #'0',inbuf        ;  Put a blank space as first character.
                mov     #'.',inbuf+1      ;  Make 0.5 the mantissa.
                mov     #'5',inbuf+2      ;  Make 0.5 the mantissa.  Inbuf now contains '0.5E+/-XX'.
                bra     send_knob2CCWg    ;  Send the set command string.
knob2CCW_2g:    mov     #'1',inbuf        ;  Make 1 the first character.
                bra     send_knob2CCWg    ;  Send the set command string.  Inbuf now contains '1.0E+/-XX'.
knob2CCW_5g:    mov     #'2',inbuf        ;  Make 2 the first character.  Inbuf now contains '2.0E+/-XX'.
send_knob2CCWg: ldhx    #msg_sca1_set     ;  Point H:X at the set message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob2CCWg_next    ;  Return to main loop until next time through.

;State 2: Send the value string from inbuf after the command string is finished.
knob2CCWg_st2:  clrh                      ;  Clear H for IX1 addressing.
                ldx     #$06              ;  Initialize X with the number of characters to move from inbuf to float_buf.
knob2CCWg_loop: lda     inbuf-1,x         ;  Get the character from inbuf.
                sta     float_buf-1,x     ;  Store it in float_buf.
                dbnzx   knob2CCWg_loop    ;  Loop until all relevant inbuf characters have been moved.
                lda     float_buf+4       ;  Load the 5th character of inbuf.
                cmp     #'-'              ;  See if it is a minus sign for the exponent.
                beq     knob2CCWg_term    ;  Terminate the string if so.
                lda     #' '              ;  Load a space.
                sta     float_buf+5       ;  Store it after the positive exponent.
knob2CCWg_term: lda     #!10              ;  Load the scope's termination character
                sta     float_buf+6       ;  Store it at the end of the data string.
                lda     #$00              ;  Load the null character.
                sta     float_buf+7       ;  Terminate the string with the null character.
                ldhx    #float_buf        ;  Point H:X at the new scale value.
                sthx    out_data_ptr      ;  Store it in the output data pointer.

knob2CCWg_done: clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+3     ;  Clear the action required bit for the key.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
knob2CCWg_next: jmp     do_key_return     ;  Restore registers and return to main_loop.

;Note:  knob2_CCW message strings are shared with those of knob2_CW.
;-----------------------------------------------------------------------------------------



;-----------------------------------------------------------------------------------------
;Knob 2 counterclockwise state machine for decreasing channel 1 position
```

```
knob2_CCWpos:     ldhx     #knob2_CCWpos     ;  Get the address of this routine.
                  sthx     key_vector        ;  Indicate that this routine is currently running.
                  lda      out_data_ptr      ;  See if there is RS-232 output data pending.
                  bne      knob2CCWp_next    ;  Skip until 232 output data is done.
                  lda      waiting           ;  See if we are waiting for reply data.
                  bne      knob2CCWp_next    ;  Skip until finished waiting for data.

do_knob2CCWp:     lda      temp_state        ;  Check the temporary state variable.
                  beq      knob2CCWp_st0     ;  If it is non-zero, the inquiry command is done.
                  cmp      #$01              ;  See if it is 1.
                  beq      knob2CCWp_st1     ;  Go to state 1 (subtract increment and send the set command).
                  bra      knob2CCWp_st2     ;  It must be state 2 (send new position level).

;State 0: Query channel 1 vertical position.
knob2CCWp_st0:    ldhx     #msg_pos1_inq     ;  Point H:X at the inquiry message.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bset     0,waiting         ;  Set the waiting bit
                  bra      knob2CCWp_next    ;  Return to main_loop until next time through.

;State 1: Subtract 0.02 divisions times the acceleration value and send the set message.
knob2CCWp_st1:    clrh                       ;  Clear H for IX1 addressing.
                  ldx      #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob2CCWp_clr:    clr      BCD_inc-1,x       ;  Clear the BCD_inc value.
                  dbnzx    knob2CCWp_clr     ;  Loop until done.  BCD_inc now contains zero.
                  mov      #$02,BCD_inc+3    ;  Load the 0.02 division increment value into BCD_inc.
                  jsr      ascii2fixed       ;  Convert the current position to fixed point BCD.
                  lda      knob_inc_count+3  ;  Get the number of times to increment for this knob.
                  sta      accel_temp        ;  Make a copy of it.

knob2CCWp_accel:  jsr      fixed_sub         ;  Subtract the increment.
                  ldx      #$09              ;  Initialize X to move 8 bytes.
accel2CCWp_loop:  lda      BCD_dest-1,x      ;  Load the result from the previous math operation.
                  sta      BCD_source-1,x    ;  Store it back as the source.
                  dbnzx    accel2CCWp_loop   ;  Loop until all BCD numbers have been moved.
                  dec      accel_temp        ;  Decrement the temporary acceleration counter.
                  bne      knob2CCWp_accel   ;  Continue to subtract the increment until done.

                  jsr      BCD2float         ;  Convert the result to floating point ASCII.
                  ldhx     #msg_pos1_set     ;  Point H:X at the channel level set command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bra      knob2CCWp_next    ;  Return to main_loop until next time through.

;State 2: Send the new channel 1 position value to the scope.
knob2CCWp_st2:    ldhx     #float_buf        ;  Point H:X at the floating point level value.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.

knob2CCWp_done:   clr      waiting           ;  Clear the waiting bit.
                  clr      temp_state        ;  Reset the temporary state variable.
                  bclr     5,knobstate+!3    ;  Clear the action required bit for the key.
                  ldhx     #do_process       ;  Get the address of the key processing routine.
                  sthx     key_vector        ;  Make sure it runs next time since this one is complete.

knob2CCWp_next:   jmp      do_key_return     ;  Restore registers and return to main_loop.

;Note: msg_pos1_inq and msg_pos1_set are shared with those of knob2CWp.
;-----------------------------------------------------------------------------------------------


knob_1CW:         brclr    1,LED_bank_C,knob_1CWgain  ;  If the scale LED is lit, adjust the scale.
                  jmp      knob1_CWpos       ;  Else adjust the position.

;-----------------------------------------------------------------------------------------------
;Knob 1 clockwise state machine (decrease channel 2 vertical gain)

knob_1CWgain:     ldhx     #knob_1CWgain     ;  Get the address of this routine.
                  sthx     key_vector        ;  Indicate that this routine is currently running.
                  lda      out_data_ptr      ;  See if there is RS-232 output data pending.
```

```
                bne     knob1CWg_next     ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob1CWg_next     ;  Skip until finished waiting for data.

do_knob1CWg:    lda     temp_state        ;  Check the temporary state variable.
                beq     knob1CWg_st0      ;  If it is non-zero, the inquiry command is done.
                cmp     #$01              ;  See if it is state 1.
                beq     knob1CWg_st1      ;  Go to state 1 (the processing state).
                bra     knob1CWg_st2      ;  It must be state 2 (the send data state).

;State 0: Query channel 2 vertical scale.
knob1CWg_st0:   ldhx    #msg_sca2_inq     ;  Point H:X at the inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob1CWg_next     ;  Return to main_loop until next time through.

;State 1: Process the data and send the command string.
knob1CWg_st1:   inc     temp_state        ;  Advance the state variable.
                lda     inbuf             ;  Get the first number in inbuf.
                cmp     #'5'              ;  See if it is 5.
                beq     knob1CWg_5        ;  Go to the next increment above 5.
                cmp     #'2'              ;  See if it is 2.
                beq     knob1CWg_2        ;  Go to the next increment above 2.
                cmp     #'1'              ;  See if it is 1.
                beq     knob1CWg_1        ;  Go to the next increment above 1.
                bra     knob1CWg_done     ;  Do nothing if none of these numbers appear.
knob1CWg_5:     mov     #' ',inbuf        ;  Put a blank space as first character.
                mov     #'1',inbuf+1      ;  Make 10 the mantissa.
                mov     #'0',inbuf+2      ;  Make 10 the mantissa.  Inbuf now contains ' 10E+/-XX'.
                bra     send_knob1CWg     ;  Send the set command string.
knob1CWg_2:     mov     #'5',inbuf        ;  Make 5 the first character.
                bra     send_knob1CWg     ;  Send the set command string.  Inbuf now contains '5.0E+/-XX'.
knob1CWg_1:     mov     #'2',inbuf        ;  Make 2 the first character.  Inbuf now contains '2.0E+/-XX'.
send_knob1CWg:  ldhx    #msg_sca2_set     ;  Point H:X at the set message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob1CWg_next     ;  Return to main loop until next time through.

;State 2: Send the value string from inbuf after the command string is finished.
knob1CWg_st2:   clrh                      ;  Clear H for IX1 addressing.
                ldx     #$06              ;  Initialize X with the number of characters to move from inbuf to float_buf.
knob1CWg_loop:  lda     inbuf-1,x         ;  Get the character from inbuf.
                sta     float_buf-1,x     ;  Store it in float_buf.
                dbnzx   knob1CWg_loop     ;  Loop until all relevant inbuf characters have been moved.
                lda     float_buf+4       ;  Load the 5th character of inbuf.
                cmp     #'-'              ;  See if it is a minus sign for the exponent.
                beq     knob1CWg_term     ;  Terminate the string if so.
                lda     #' '              ;  Load a space.
                sta     float_buf+5       ;  Store it after the positive exponent.
knob1CWg_term:  lda     #!10              ;  Load the scope's termination character
                sta     float_buf+6       ;  Store it at the end of the data string.
                lda     #$00              ;  Load the null character.
                sta     float_buf+7       ;  Terminate the string with the null character.
                ldhx    #float_buf        ;  Point H:X at the new scale value.
                sthx    out_data_ptr      ;  Store it in the output data pointer.

knob1CWg_done:  clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+!4    ;  Clear the action required bit for the key.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
knob1CWg_next:  jmp     do_key_return     ;  Restore registers and return to main_loop.

msg_sca2_inq:   db      'CH2:SCA?',!10,0
msg_sca2_set:   db      'CH2:SCA ',0
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Knob 1 clockwise state machine for increasing channel 2 position
```

```
knob1_CWpos:      ldhx    #knob1_CWpos      ;  Get the address of this routine.
                  sthx    key_vector        ;  Indicate that this routine is currently running.
                  lda     out_data_ptr      ;  See if there is RS-232 output data pending.
                  bne     knob1CWp_next     ;  Skip until 232 output data is done.
                  lda     waiting           ;  See if we are waiting for reply data.
                  bne     knob1CWp_next     ;  Skip until finished waiting for data.

do_knob1CWp:      lda     temp_state        ;  Check the temporary state variable.
                  beq     knob1CWp_st0      ;  If it is non-zero, the inquiry command is done.
                  cmp     #$01              ;  See if it is 1.
                  beq     knob1CWp_st1      ;  Go to state 1 (add increment and send the set command).
                  bra     knob1CWp_st2      ;  It must be state 2 (set new position level).

;State 0: Query channel 2 vertical position.
knob1CWp_st0:     ldhx    #msg_pos2_inq     ;  Point H:X at the inquiry message.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  inc     temp_state        ;  Advance the state variable.
                  bset    0,waiting         ;  Set the waiting bit
                  bra     knob1CWp_next     ;  Return to main_loop until next time through.

;State 1: Add 0.02 divisions times the acceleration value and send the set message.
knob1CWp_st1:     clrh                      ;  Clear H for IX1 addressing.
                  ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob1CWp_clr:     clr     BCD_inc-1,x       ;  Clear the BCD_inc value.
                  dbnzx   knob1CWp_clr      ;  Loop until done.  BCD_inc now contains zero.
                  mov     #$02,BCD_inc+3    ;  Load the 0.02 division increment value into BCD_inc.
                  jsr     ascii2fixed       ;  Convert the current position to fixed point BCD.
                  lda     knob_inc_count+4  ;  Get the number of times to increment for this knob.
                  sta     accel_temp        ;  Make a copy of it.
knob1CWp_accel:   jsr     fixed_add         ;  Add the increment.
                  ldx     #$09              ;  Initialize X to move 8 bytes.
accel1CWp_loop:   lda     BCD_dest-1,x      ;  Load the result from the previous math operation.
                  sta     BCD_source-1,x    ;  Store it back as the source.
                  dbnzx   accel1CWp_loop    ;  Loop until all BCD numbers have been moved.
                  dec     accel_temp        ;  Decrement the temporary acceleration counter.
                  bne     knob1CWp_accel    ;  Continue to add the increment until done.

                  jsr     BCD2float         ;  Convert the result to floating point ASCII.
                  ldhx    #msg_pos2_set     ;  Point H:X at the channel level set command.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  inc     temp_state        ;  Advance the state variable.
                  bra     knob1CWp_next     ;  Return to main_loop until next time through.

;State 2: Send the new channel 2 position value to the scope.
knob1CWp_st2:     ldhx    #float_buf        ;  Point H:X at the floating point level value.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.

knob1CWp_done:    clr     waiting           ;  Clear the waiting bit.
                  clr     temp_state        ;  Reset the temporary state variable.
                  bclr    5,knobstate+!4    ;  Clear the action required bit for the key.
                  ldhx    #do_process       ;  Get the address of the key processing routine.
                  sthx    key_vector        ;  Make sure it runs next time since this one is complete.

knob1CWp_next:    jmp     do_key_return     ;  Restore registers and return to main_loop.

msg_pos2_inq:     db      'CH2:POS?',!10,0
msg_pos2_set:     db      'CH2:POS ',0
;-----------------------------------------------------------------------------------------------


knob_1CCW:        brclr   1,LED_bank_C,knob_1CCWgain ;  If the scale LED is lit, adjust the scale.
                  jmp     knob1_CCWpos      ;  Else adjust the position.

;-----------------------------------------------------------------------------------------------
;Knob 1 counterclockwise state machine (increase channel 1 vertical gain)

knob_1CCWgain:    ldhx    #knob_1CCWgain    ;  Get the address of this routine.
                  sthx    key_vector        ;  Indicate that this routine is currently running.
                  lda     out_data_ptr      ;  See if there is RS-232 output data pending.
```

```
                bne     knob1CCWg_next    ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob1CCWg_next    ;  Skip until finished waiting for data.

do_knob1CCWg:   lda     temp_state        ;  Check the temporary state variable.
                beq     knob1CCWg_st0     ;  If it is non-zero, the inquiry command is done.
                cmp     #$01              ;  See if it is state 1.
                beq     knob1CCWg_st1     ;  Go to state 1 (the processing state).
                bra     knob1CCWg_st2     ;  It must be state 2 (the send data state).

;State 0: Query channel 2 vertical scale.
knob1CCWg_st0:  ldhx    #msg_sca2_inq     ;  Point H:X at the inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob1CCWg_next    ;  Return to main_loop until next time through.

;State 1: Process the data and send the command string.
knob1CCWg_st1:  inc     temp_state        ;  Advance the state variable.
                lda     inbuf             ;  Get the first number in inbuf.
                cmp     #'5'              ;  See if it is 5.
                beq     knob1CCWg_5       ;  Go to the next increment below 5.
                cmp     #'2'              ;  See if it is 2.
                beq     knob1CCWg_2       ;  Go to the next increment below 2.
                cmp     #'1'              ;  See if it is 1.
                beq     knob1CCWg_1       ;  Go to the next increment below 1.
                bra     knob1CCWg_done    ;  Do nothing if none of these numbers appear in inbuf.
knob1CCWg_1:    mov     #'0',inbuf        ;  Put a blank space as first character.
                mov     #'.',inbuf+1      ;  Make 0.5 the mantissa.
                mov     #'5',inbuf+2      ;  Make 0.5 the mantissa.  Inbuf now contains '0.5E+/-XX'.
                bra     send_knob1CCWg    ;  Send the set command string.
knob1CCWg_2:    mov     #'1',inbuf        ;  Make 1 the first character.
                bra     send_knob1CCWg    ;  Send the set command string.  Inbuf now contains '1.0E+/-XX'.
knob1CCWg_5:    mov     #'2',inbuf        ;  Make 2 the first character.  Inbuf now contains '2.0E+/-XX'.
send_knob1CCWg: ldhx    #msg_sca2_set     ;  Point H:X at the set message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob1CCWg_next    ;  Return to main loop until next time through.

;State 2: Send the value string from inbuf after the command string is finished.
knob1CCWg_st2:  clrh                      ;  Clear H for IX1 addressing.
                ldx     #$06              ;  Initialize X with the number of characters to move from inbuf to float_buf.
knob1CCWg_loop: lda     inbuf-1,x         ;  Get the character from inbuf.
                sta     float_buf-1,x     ;  Store it in float_buf.
                dbnzx   knob1CCWg_loop    ;  Loop until all relevant inbuf characters have been moved.
                lda     float_buf+4       ;  Load the 5th character of inbuf.
                cmp     #'-'              ;  See if it is a minus sign for the exponent.
                beq     knob1CCWg_term    ;  Terminate the string if so.
                lda     #' '              ;  Load a space.
                sta     float_buf+5       ;  Store it after the positive exponent.
knob1CCWg_term: lda     #!10              ;  Load the scope's termination character
                sta     float_buf+6       ;  Store it at the end of the data string.
                lda     #$00              ;  Load the null character.
                sta     float_buf+7       ;  Terminate the string with the null character.
                ldhx    #float_buf        ;  Point H:X at the new scale value.
                sthx    out_data_ptr      ;  Store it in the output data pointer.

knob1CCWg_done: clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+!5    ;  Clear the action required bit for the key.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
knob1CCWg_next: jmp     do_key_return     ;  Restore registers and return to main_loop.

;Note:  knob1_CCW message strings are shared with those of knob1_CW.
;-----------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------
;Knob 1 counterclockwise state machine for decreasing channel 2 position
```

```
knob1_CCWpos:     ldhx     #knob1_CCWpos     ;  Get the address of this routine.
                  sthx     key_vector        ;  Indicate that this routine is currently running.
                  lda      out_data_ptr      ;  See if there is RS-232 output data pending.
                  bne      knob1CCWp_next    ;  Skip until 232 output data is done.
                  lda      waiting           ;  See if we are waiting for reply data.
                  bne      knob1CCWp_next    ;  Skip until finished waiting for data.

do_knob1CCWp:     lda      temp_state        ;  Check the temporary state variable.
                  beq      knob1CCWp_st0     ;  If it is non-zero, the inquiry command is done.
                  cmp      #$01              ;  See if it is 1.
                  beq      knob1CCWp_st1     ;  Go to state 1 (subtract increment and send the set command).
                  bra      knob1CCWp_st2     ;  It must be state 2 (send new position level).

;State 0: Query channel 1 vertical position.
knob1CCWp_st0:    ldhx     #msg_pos2_inq     ;  Point H:X at the inquiry message.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bset     0,waiting         ;  Set the waiting bit
                  bra      knob1CCWp_next    ;  Return to main_loop until next time through.

;State 1: Subtract 0.02 divisions times the acceleration value and send the set message.
knob1CCWp_st1:    clrh                       ;  Clear H for IX1 addressing.
                  ldx      #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob1CCWp_clr:    clr      BCD_inc-1,x       ;  Clear the BCD_inc value.
                  dbnzx    knob1CCWp_clr     ;  Loop until done.  BCD_inc now contains zero.
                  mov      #$02,BCD_inc+3    ;  Load the 0.02 division increment value into BCD_inc.
                  jsr      ascii2fixed       ;  Convert the current position to fixed point BCD.
                  lda      knob_inc_count+5  ;  Get the number of times to increment for this knob.
                  sta      accel_temp        ;  Make a copy of it.

knob1CCWp_accel:  jsr      fixed_sub         ;  Subtract the increment.
                  ldx      #$09              ;  Initialize X to move 8 bytes.
accel1CCWp_loop:  lda      BCD_dest-1,x      ;  Load the result from the previous math operation.
                  sta      BCD_source-1,x    ;  Store it back as the source.
                  dbnzx    accel1CCWp_loop   ;  Loop until all BCD numbers have been moved.
                  dec      accel_temp        ;  Decrement the temporary acceleration counter.
                  bne      knob1CCWp_accel   ;  Continue to subtract the increment until done.

                  jsr      BCD2float         ;  Convert the result to floating point ASCII.
                  ldhx     #msg_pos2_set     ;  Point H:X at the channel level set command.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.
                  inc      temp_state        ;  Advance the state variable.
                  bra      knob1CCWp_next    ;  Return to main_loop until next time through.

;State 2: Send the new channel 2 position value to the scope.
knob1CCWp_st2:    ldhx     #float_buf        ;  Point H:X at the floating point level value.
                  sthx     out_data_ptr      ;  Store it in the output data pointer.

knob1CCWp_done:   clr      waiting           ;  Clear the waiting bit.
                  clr      temp_state        ;  Reset the temporary state variable.
                  bclr     5,knobstate+!5    ;  Clear the action required bit for the key.
                  ldhx     #do_process       ;  Get the address of the key processing routine.
                  sthx     key_vector        ;  Make sure it runs next time since this one is complete.

knob1CCWp_next:   jmp      do_key_return     ;  Restore registers and return to main_loop.

;Note: msg_pos2_inq and msg_pos2_set are shared with those of knob1CWg.
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Knob 0 clockwise handler interface (increase cursor position)
;This handler interface hands-off control to the appropriate knob 0 clockwise state machine
;depending on the type of cursor selected.

knob_0CW:         brclr    0,LED_bank_D,knob0CW_H     ;  See if HBARs are active.
                  brclr    1,LED_bank_D,knob0CW_V     ;  See if VBARs are active.
                  brclr    2,LED_bank_D,knob0CW_V     ;  See if paired cursors are active (handled like VBARs).
knob0_noCurs:     clr      waiting           ;  Clear the waiting bit.
                  clr      temp_state        ;  Reset the temporary state variable.
```

```
                bclr    5,knobstate+!6    ;  Clear the action required bit for the knob.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return     ;  Restore registers and return to main_loop.

knob0CW_H:      ldhx    #knob0CW_Hbar     ;  Load H:X with the address of the hbar handler.
                sthx    key_vector        ;  Queue it to run next time.
                jmp     do_key_return     ;  Restore registers and return to main_loop.
knob0CW_V:      ldhx    #knob0CW_Vbar     ;  Load H:X with the address of the vbar handler.
                sthx    key_vector        ;  Queue it to run next time.
                jmp     do_key_return     ;  Restore registers and return to main_loop.
;-----------------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------------
;Knob 0 clockwise HBAR state machine.  On entry, it is known that HBARs are active and the
;selected HBAR is stored in curs_active.  The channel to which the cursors are attached needs
;to be determined so that the scale value can be found and added to the current position.

knob0CW_hbar:   lda     out_data_ptr      ;  See if there is RS-232 output data pending.
                bne     knob0CW_h_next    ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob0CW_h_next    ;  Skip until finished waiting for data.

                lda     temp_state        ;  Get the temporary state variable.
                beq     knob0CW_h_st0     ;  Go to state 0.
                cmp     #$01              ;  See if it is state 1.
                beq     knob0CW_h_st1     ;  Go to state 1 if so.
                cmp     #$02              ;  See if it is state 2.
                beq     knob0CW_h_st2     ;  Go to state 2 if so.
                cmp     #$03              ;  See if it is state 3.
                beq     knob0CW_h_st3     ;  Go to state 3 if so.
                bra     knob0CW_h_st4     ;  Else it must be state 4.

;State 0: Query the active channel (the channel to which the HBARs are attached).
knob0CW_h_st0:  ldhx    #msg_hchan_inq    ;  Point H:X at the active channel inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob0CW_h_next    ;  Return to main_loop until next time through.

;State 1: Get the active channel from inbuf (either CH1 or CH2) and query the scale.
knob0CW_h_st1:  bset    0,waiting         ;  Set the waiting bit
                inc     temp_state        ;  Advance the state variable.
                lda     inbuf+2           ;  Get the active channel number.
                sta     curs_chan         ;  Temporarily store it.
                cmp     #'2'              ;  See if it is channel 2.
                beq     knob0CW_h_ch2I    ;  Handle channel 2 if so, else handle channel 1.
knob0CW_h_ch1I: ldhx    #msg_sca1_inq     ;  Point H:X at the channel 1 scale inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CW_h_next    ;  Return to main_loop until next time through.
knob0CW_h_ch2I: ldhx    #msg_sca2_inq     ;  Point H:X at the channel 2 scale inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CW_h_next    ;  Return to main_loop until next time through.

knob0CW_h_next: jmp     do_key_return     ;  Restore registers and return to main_loop.

;State 2: Get the active channel scale and convert to Y increment units.  Inbuf contains the
;Y scale.  Then query the position of the active cursor.
knob0CW_h_st2:  jsr     get_Y_inc         ;  Convert the scale value in inbuf to Y increments.
                jsr     ascii2fixed       ;  Convert the floating point to BCD.
                clrh                      ;  Clear H for IX1 addressing.
                ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob0CW_h_move: lda     BCD_source-1,x    ;  Load the source data.
                sta     BCD_inc-1,x       ;  Store it in BCD_inc.
                dbnzx   knob0CW_h_move    ;  Loop until done.

                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                lda     curs_active       ;  Get the active cursor number.
```

```
                cmp     #'2'            ;  See if it is 2.
                beq     knob0CW_h_c2    ;  Query cursor 2 if so.  Else query cursor 1.
knob0CW_h_c1:   ldhx    #msg_curs_Hinq1 ;  Point H:X at the cursor 1 inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_h_next  ;  Return to main_loop until next time through.
knob0CW_h_c2:   ldhx    #msg_curs_Hinq2 ;  Point H:X at the cursor 2 inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_h_next  ;  Return to main_loop until next time through.

;State 3: Add the increment to the position and set the new cursor position.
knob0CW_h_st3:  jsr     ascii2fixed     ;  Convert the floating point position to BCD.
                lda     knob_inc_count+6;  Get the number of times to increment for this knob.
                asla                    ;  Multiply it by two for the cursor increment.
                sta     accel_temp      ;  Make a copy of it.

knob0CW_h_accel: jsr    fixed_add       ;  Add the increment.
                clrh                    ;  Clear H for IX1 addressing.
                ldx     #$09            ;  Prepare to copy the 8 bytes of BCD data to BCD_source.
knob0CW_h_copy: lda     BCD_dest-1,x    ;  Load the destination data.
                sta     BCD_source-1,x  ;  Store it in BCD_source.
                dbnzx   knob0CW_h_copy  ;  Loop until done.
                dec     accel_temp      ;  Decrement the temporary acceleration counter.
                bne     knob0CW_h_accel ;  Continue to add the increment until done.

                inc     temp_state      ;  Advance the state variable.
                lda     curs_active     ;  Get the active cursor number.
                cmp     #'2'            ;  See if it is 2.
                beq     knob0CW_h_s2    ;  Set cursor 2 if so.  Else set cursor 1.
knob0CW_h_s1:   ldhx    #msg_curs_Hset1 ;  Point H:X at the cursor 1 set message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_h_next  ;  Return to main_loop until next time through.
knob0CW_h_s2:   ldhx    #msg_curs_Hset2 ;  Point H:X at the cursor 2 set message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_h_next  ;  Return to main_loop until next time through.

;State 4: Send the new cursor position.
knob0CW_h_st4:  jsr     BCD2float       ;  Convert the result to floating point ASCII.
                ldhx    #float_buf      ;  Point H:X at the floating point level value.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                clr     waiting         ;  Clear the waiting bit.
                clr     temp_state      ;  Reset the temporary state variable.
                bclr    5,knobstate+!6  ;  Clear the action required bit for the knob.
                ldhx    #do_process     ;  Get the address of the key processing routine.
                sthx    key_vector      ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return   ;  Restore registers and return to main_loop.

msg_hchan_inq:  db      'SEL:CONTROL?',!10,0
msg_curs_Hinq1: db      'CURS:HBA:POSITION1?',!10,0
msg_curs_Hset1: db      'CURS:HBA:POSITION1 ',0
msg_curs_Hinq2: db      'CURS:HBA:POSITION2?',!10,0
msg_curs_Hset2: db      'CURS:HBA:POSITION2 ',0
;-----------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------
;Knob 0 clockwise VBAR state machine.  On entry, it is known that VBARs or PBARs are active and
;the selected VBAR is stored in curs_active.  The magnify (X10) mode needs to be determined so
;that the increment can determined.  The WFMP:CH1:XIN query command will return the exact
;increment value if the scope is in the magnify mode and .1X the increment if the scope is not
;in magnify mode.

knob0CW_vbar:   lda     out_data_ptr    ;  See if there is RS-232 output data pending.
                bne     knob0CW_v_next  ;  Skip until 232 output data is done.
                lda     waiting         ;  See if we are waiting for reply data.
                bne     knob0CW_v_next  ;  Skip until finished waiting for data.

                lda     temp_state      ;  Get the temporary state variable.
                beq     knob0CW_v_st0   ;  Go to state 0.
                cmp     #$01            ;  See if it is state 1.
                beq     knob0CW_v_st1   ;  Go to state 1 if so.
```

```
                cmp     #$02            ;  See if it is state 2.
                beq     knob0CW_v_st2   ;  Go to state 2 if so.
                cmp     #$03            ;  See if it is state 3.
                beq     knob0CW_v_st3   ;  Go to state 3 if so.
                bra     knob0CW_v_st4   ;  Else it must be state 4.

;State 0: Query the magnify mode.
knob0CW_v_st0:  ldhx    #msg_mag_inq    ;  Point H:X at the magnify inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit
                bra     knob0CW_v_next  ;  Return to main_loop until next time through.

;State 1: Store the magnify mode and query the X axis (time) increment.
knob0CW_v_st1:  bset    0,waiting       ;  Set the waiting bit
                inc     temp_state      ;  Advance the state variable.
                lda     inbuf           ;  Get the magnify mode ('0' on, '1' off).
                sta     curs_chan       ;  Temporarily store it.
                ldhx    #msg_xinc_inq   ;  Point H:X at the X scale inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_v_next  ;  Return to main_loop until next time through.

knob0CW_v_next: jmp     do_key_return   ;  Restore registers and return to main_loop.

;State 2: Convert the increment value to BCD and store it in BCD_inc.  Multiply the increment
;by 10 if the magnify mode is not on.  Then query the position of the active cursor.
knob0CW_v_st2:  jsr     ascii2fixed     ;  Convert the floating point to BCD.
                clrh                    ;  Clear H for IX1 addressing.
                ldx     #$09            ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob0CW_v_move: lda     BCD_source-1,x  ;  Load the source data.
                sta     BCD_inc-1,x     ;  Store it in BCD_inc.
                dbnzx   knob0CW_v_move  ;  Loop until done.
                lda     curs_chan       ;  Get the result of the magnify query.
                cmp     #'0'            ;  See if magnify mode is on.
                beq     knob0CW_v_inq   ;  Do not multiply the increment by 10 if in magnify mode.
                jsr     mult_inc_by_10  ;  Else multiply the increment by 10.
knob0CW_v_inq:  inc     temp_state      ;  Advance the state variable.
                bset    0,waiting       ;  Set the waiting bit
                lda     curs_active     ;  Get the active cursor number.
                cmp     #'2'            ;  See if it is 2.
                beq     knob0CW_v_c2    ;  Query cursor 2 if so.  Else query cursor 1.
knob0CW_v_c1:   ldhx    #msg_curs_Vinq1 ;  Point H:X at the cursor 1 inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_v_next  ;  Return to main_loop until next time through.
knob0CW_v_c2:   ldhx    #msg_curs_Vinq2 ;  Point H:X at the cursor 2 inquiry message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_v_next  ;  Return to main_loop until next time through.

;State 3: Add the increment to the position and set the new cursor position.
knob0CW_v_st3:  jsr     ascii2fixed     ;  Convert the floating point position to BCD.
                lda     knob_inc_count+6;  Get the number of times to increment for this knob.
                sta     accel_temp      ;  Make a copy of it.

knob0CW_v_accel: jsr    fixed_add       ;  Add the increment.
                clrh                    ;  Clear H for IX1 addressing.
                ldx     #$09            ;  Prepare to copy the 8 bytes of BCD data to BCD_source.
knob0CW_v_copy: lda     BCD_dest-1,x    ;  Load the destination data.
                sta     BCD_source-1,x  ;  Store it in BCD_source.
                dbnzx   knob0CW_v_copy  ;  Loop until done.
                dec     accel_temp      ;  Decrement the temporary acceleration counter.
                bne     knob0CW_v_accel ;  Continue to add the increment until done.

                inc     temp_state      ;  Advance the state variable.
                lda     curs_active     ;  Get the active cursor number.
                cmp     #'2'            ;  See if it is 2.
                beq     knob0CW_v_s2    ;  Set cursor 2 if so.  Else set cursor 1.
knob0CW_v_s1:   ldhx    #msg_curs_Vset1 ;  Point H:X at the cursor 1 set message.
                sthx    out_data_ptr    ;  Store it in the output data pointer.
                bra     knob0CW_v_next  ;  Return to main_loop until next time through.
knob0CW_v_s2:   ldhx    #msg_curs_Vset2 ;  Point H:X at the cursor 2 set message.
```

```
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CW_v_next    ;  Return to main_loop until next time through.

;State 4: Send the new cursor position.
knob0CW_v_st4:  jsr     BCD2float         ;  Convert the result to floating point ASCII.
                ldhx    #float_buf        ;  Point H:X at the floating point level value.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+!6    ;  Clear the action required bit for the knob.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return     ;  Restore registers and return to main_loop.

msg_mag_inq:    db      'HOR:FIT?',!10,0
msg_xinc_inq    db      'WFMP:CH1:XIN?',!10,0
msg_curs_Vinq1: db      'CURS:VBA:POSITION1?',!10,0
msg_curs_Vset1: db      'CURS:VBA:POSITION1 ',0
msg_curs_Vinq2: db      'CURS:VBA:POSITION2?',!10,0
msg_curs_Vset2: db      'CURS:VBA:POSITION2 ',0
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;Knob 0 counterclockwise handler interface (decrease cursor position)
;This handler interface hands-off control to the appropriate knob 0 counterclockwise state
;machine depending on the type of cursor selected.

knob_0CCW:      brclr   0,LED_bank_D,knob0CCW_H    ;  See if HBARs are active.
                brclr   1,LED_bank_D,knob0CCW_V    ;  See if VBARs are active.
                brclr   2,LED_bank_D,knob0CCW_V    ;  See if paired cursors are active (handled like VBARs).
knob_0noCurs:   clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+!7    ;  Clear the action required bit for the knob.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return     ;  Restore registers and return to main_loop.

knob0CCW_H:     ldhx    #knob0CCW_Hbar    ;  Load H:X with the address of the hbar handler.
                sthx    key_vector        ;  Queue it to run next time.
                jmp     do_key_return     ;  Restore registers and return to main_loop.
knob0CCW_V:     ldhx    #knob0CCW_Vbar    ;  Load H:X with the address of the vbar handler.
                sthx    key_vector        ;  Queue it to run next time.
                jmp     do_key_return     ;  Restore registers and return to main_loop.
;-------------------------------------------------------------------------------------------


;-------------------------------------------------------------------------------------------
;Knob 0 counterclockwise HBAR state machine.  On entry, it is known that HBARs are active and
;the selected HBAR is stored in curs_active.  The channel to which the cursors are attached
;needs to be determined so that the scale value can be found and subtracted from the current
;position.

knob0CCW_hbar:  lda     out_data_ptr      ;  See if there is RS-232 output data pending.
                bne     knob0CCW_h_next   ;  Skip until 232 output data is done.
                lda     waiting           ;  See if we are waiting for reply data.
                bne     knob0CCW_h_next   ;  Skip until finished waiting for data.

                lda     temp_state        ;  Get the temporary state variable.
                beq     knob0CCW_h_st0    ;  Go to state 0.
                cmp     #$01              ;  See if it is state 1.
                beq     knob0CCW_h_st1    ;  Go to state 1 if so.
                cmp     #$02              ;  See if it is state 2.
                beq     knob0CCW_h_st2    ;  Go to state 2 if so.
                cmp     #$03              ;  See if it is state 3.
                beq     knob0CCW_h_st3    ;  Go to state 3 if so.
                bra     knob0CCW_h_st4    ;  Else it must be state 4.

;State 0: Query the active channel (the channel to which the HBARs are attached).
knob0CCW_h_st0: ldhx    #msg_hchan_inq    ;  Point H:X at the active channel inquiry message.
```

```
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                bra     knob0CCW_h_next   ;  Return to main_loop until next time through.

;State 1: Get the active channel from inbuf (either CH1 or CH2) and query the scale.
knob0CCW_h_st1: bset    0,waiting         ;  Set the waiting bit
                inc     temp_state        ;  Advance the state variable.
                lda     inbuf+2           ;  Get the active channel number.
                sta     curs_chan         ;  Temporarily store it.
                cmp     #'2'              ;  See if it is channel 2.
                beq     knob0CCW_h_ch2I   ;  Handle channel 2 if so, else handle channel 1.
knob0CCW_h_ch1I: ldhx   #msg_sca1_inq     ;  Point H:X at the channel 1 scale inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_h_next   ;  Return to main_loop until next time through.
knob0CCW_h_ch2I: ldhx   #msg_sca2_inq     ;  Point H:X at the channel 2 scale inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_h_next   ;  Return to main_loop until next time through.

knob0CCW_h_next: jmp    do_key_return     ;  Restore registers and return to main_loop.

;State 2: Get the active channel scale and convert to Y increment units.  Inbuf contains the
;Y scale.  Then query the position of the active cursor.
knob0CCW_h_st2: jsr     get_Y_inc         ;  Convert the scale value in inbuf to Y increments.
                jsr     ascii2fixed       ;  Convert the floating point to BCD.
                clrh                      ;  Clear H for IX1 addressing.
                ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob0CCW_h_move: lda    BCD_source-1,x    ;  Load the source data.
                sta     BCD_inc-1,x       ;  Store it in BCD_inc.
                dbnzx   knob0CCW_h_move   ;  Loop until done.

                inc     temp_state        ;  Advance the state variable.
                bset    0,waiting         ;  Set the waiting bit
                lda     curs_active       ;  Get the active cursor number.
                cmp     #'2'              ;  See if it is 2.
                beq     knob0CCW_h_c2     ;  Query cursor 2 if so.  Else query cursor 1.
knob0CCW_h_c1:  ldhx    #msg_curs_Hinq1   ;  Point H:X at the cursor 1 inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_h_next   ;  Return to main_loop until next time through.
knob0CCW_h_c2:  ldhx    #msg_curs_Hinq2   ;  Point H:X at the cursor 2 inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_h_next   ;  Return to main_loop until next time through.

;State 3: Subtract the increment from the position and set the new cursor position.
knob0CCW_h_st3: jsr     ascii2fixed       ;  Convert the floating point position to BCD.
                lda     knob_inc_count+7  ;  Get the number of times to increment for this knob.
                asla                      ;  Multiply it by two for the cursor increment.
                sta     accel_temp        ;  Make a copy of it.

knob0CCW_h_accel: jsr   fixed_sub         ;  Subtract the increment.
                clrh                      ;  Clear H for IX1 addressing.
                ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_source.
knob0CCW_h_copy: lda    BCD_dest-1,x      ;  Load the destination data.
                sta     BCD_source-1,x    ;  Store it in BCD_source.
                dbnzx   knob0CCW_h_copy   ;  Loop until done.
                dec     accel_temp        ;  Decrement the temporary acceleration counter.
                bne     knob0CCW_h_accel  ;  Continue to add the increment until done.

                inc     temp_state        ;  Advance the state variable.
                lda     curs_active       ;  Get the active cursor number.
                cmp     #'2'              ;  See if it is 2.
                beq     knob0CCW_h_s2     ;  Set cursor 2 if so.  Else set cursor 1.
knob0CCW_h_s1:  ldhx    #msg_curs_Hset1   ;  Point H:X at the cursor 1 set message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_h_next   ;  Return to main_loop until next time through.
knob0CCW_h_s2:  ldhx    #msg_curs_Hset2   ;  Point H:X at the cursor 2 set message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_h_next   ;  Return to main_loop until next time through.

;State 4: Send the new cursor position.
```

```
knob0CCW_h_st4:   jsr     BCD2float         ;  Convert the result to floating point ASCII.
                  ldhx    #float_buf        ;  Point H:X at the floating point level value.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  clr     waiting           ;  Clear the waiting bit.
                  clr     temp_state        ;  Reset the temporary state variable.
                  bclr    5,knobstate+!7    ;  Clear the action required bit for the knob.
                  ldhx    #do_process       ;  Get the address of the key processing routine.
                  sthx    key_vector        ;  Make sure it runs next time since this one is complete.
                  jmp     do_key_return     ;  Restore registers and return to main_loop.

;Note: Message strings for this handler are shared with those of the clockwise handler.
;-----------------------------------------------------------------------------------------


;-----------------------------------------------------------------------------------------
;Knob 0 counterclockwise VBAR state machine.  On entry, it is known that VBARs or PBARs are
;active and the selected VBAR is stored in curs_active.  The magnify (X10) mode needs to be
;determined so that the increment can determined.  The WFMP:CH1:XIN query command will return
;the exact increment value if the scope is in the magnify mode and .1X the increment if the
;scope is not in magnify mode.

knob0CCW_vbar:    lda     out_data_ptr      ;  See if there is RS-232 output data pending.
                  bne     knob0CCW_v_next   ;  Skip until 232 output data is done.
                  lda     waiting           ;  See if we are waiting for reply data.
                  bne     knob0CCW_v_next   ;  Skip until finished waiting for data.

                  lda     temp_state        ;  Get the temporary state variable.
                  beq     knob0CCW_v_st0    ;  Go to state 0.
                  cmp     #$01              ;  See if it is state 1.
                  beq     knob0CCW_v_st1    ;  Go to state 1 if so.
                  cmp     #$02              ;  See if it is state 2.
                  beq     knob0CCW_v_st2    ;  Go to state 2 if so.
                  cmp     #$03              ;  See if it is state 3.
                  beq     knob0CCW_v_st3    ;  Go to state 3 if so.
                  bra     knob0CCW_v_st4    ;  Else it must be state 4.

;State 0: Query the magnify mode.
knob0CCW_v_st0:   ldhx    #msg_mag_inq      ;  Point H:X at the magnify inquiry message.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  inc     temp_state        ;  Advance the state variable.
                  bset    0,waiting         ;  Set the waiting bit
                  bra     knob0CCW_v_next   ;  Return to main_loop until next time through.

;State 1: Store the magnify mode and query the X axis (time) increment.
knob0CCW_v_st1:   bset    0,waiting         ;  Set the waiting bit
                  inc     temp_state        ;  Advance the state variable.
                  lda     inbuf             ;  Get the magnify mode ('0' on, '1' off).
                  sta     curs_chan         ;  Temporarily store it.
                  ldhx    #msg_xinc_inq     ;  Point H:X at the X scale inquiry message.
                  sthx    out_data_ptr      ;  Store it in the output data pointer.
                  bra     knob0CCW_v_next   ;  Return to main_loop until next time through.

knob0CCW_v_next:  jmp     do_key_return     ;  Restore registers and return to main_loop.

;State 2: Convert the increment value to BCD and store it in BCD_inc.  Multiply the increment
;by 10 if the magnify mode is not on.  Then query the position of the active cursor.
knob0CCW_v_st2:   jsr     ascii2fixed       ;  Convert the floating point to BCD.
                  clrh                      ;  Clear H for IX1 addressing.
                  ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_inc.
knob0CCW_v_move:  lda     BCD_source-1,x    ;  Load the source data.
                  sta     BCD_inc-1,x       ;  Store it in BCD_inc.
                  dbnzx   knob0CCW_v_move   ;  Loop until done.
                  lda     curs_chan         ;  Get the result of the magnify query.
                  cmp     #'0'              ;  See if magnify mode is on.
                  beq     knob0CCW_v_inq    ;  Do not multiply the increment by 10 if in magnify mode.
                  jsr     mult_inc_by_10    ;  Else multiply the increment by 10.
knob0CCW_v_inq:   inc     temp_state        ;  Advance the state variable.
                  bset    0,waiting         ;  Set the waiting bit
                  lda     curs_active       ;  Get the active cursor number.
                  cmp     #'2'              ;  See if it is 2.
```

```
                beq     knob0CCW_v_c2     ;  Query cursor 2 if so.  Else query cursor 1.
knob0CCW_v_c1:  ldhx    #msg_curs_Vinq1   ;  Point H:X at the cursor 1 inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_v_next   ;  Return to main_loop until next time through.
knob0CCW_v_c2:  ldhx    #msg_curs_Vinq2   ;  Point H:X at the cursor 2 inquiry message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_v_next   ;  Return to main_loop until next time through.

;State 3: Add the increment to the position and set the new cursor position.
knob0CCW_v_st3: jsr     ascii2fixed       ;  Convert the floating point position to BCD.
                lda     knob_inc_count+7  ;  Get the number of times to increment for this knob.
                sta     accel_temp        ;  Make a copy of it.

knob0CCW_v_accel: jsr   fixed_sub         ;  Subtract the increment.
                clrh                      ;  Clear H for IX1 addressing.
                ldx     #$09              ;  Prepare to copy the 8 bytes of BCD data to BCD_source.
knob0CCW_v_copy: lda    BCD_dest-1,x      ;  Load the destination data.
                sta     BCD_source-1,x    ;  Store it in BCD_source.
                dbnzx   knob0CCW_v_copy   ;  Loop until done.
                dec     accel_temp        ;  Decrement the temporary acceleration counter.
                bne     knob0CCW_v_accel  ;  Continue to add the increment until done.

                inc     temp_state        ;  Advance the state variable.
                lda     curs_active       ;  Get the active cursor number.
                cmp     #'2'              ;  See if it is 2.
                beq     knob0CCW_v_s2     ;  Set cursor 2 if so.  Else set cursor 1.
knob0CCW_v_s1:  ldhx    #msg_curs_Vset1   ;  Point H:X at the cursor 1 set message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_v_next   ;  Return to main_loop until next time through.
knob0CCW_v_s2:  ldhx    #msg_curs_Vset2   ;  Point H:X at the cursor 2 set message.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                bra     knob0CCW_v_next   ;  Return to main_loop until next time through.

;State 4: Send the new cursor position.
knob0CCW_v_st4: jsr     BCD2float         ;  Convert the result to floating point ASCII.
                ldhx    #float_buf        ;  Point H:X at the floating point level value.
                sthx    out_data_ptr      ;  Store it in the output data pointer.
                clr     waiting           ;  Clear the waiting bit.
                clr     temp_state        ;  Reset the temporary state variable.
                bclr    5,knobstate+!7    ;  Clear the action required bit for the knob.
                ldhx    #do_process       ;  Get the address of the key processing routine.
                sthx    key_vector        ;  Make sure it runs next time since this one is complete.
                jmp     do_key_return     ;  Restore registers and return to main_loop.

;Note: Message strings for this handler are shared with those of the clockwise handler.
;-----------------------------------------------------------------------------------------

;-----------------------------------------------------------------------------------------
;68HC908GP20 Vector table
        org Vectortable

        dw      reset                     ;  Time Base Vector
        dw      reset                     ;  ADC Conversion Complete
        dw      reset                     ;  Keyboard Vector
        dw      reset                     ;  SCI Transmit Vector
        dw      reset                     ;  SCI Receive Vector
        dw      reset                     ;  SCI Error Vector
        dw      reset                     ;  SPI Transmit Vector
        dw      reset                     ;  SPI Receive Vector
        dw      reset                     ;  TIM2 Overflow Vector
        dw      reset                     ;  TIM2 Channel 1 Vector
        dw      reset                     ;  TIM2 Channel 0 Vector
        dw      keep_time                 ;  TIM1 Overflow Vector
        dw      reset                     ;  TIM1 Channel 1 Vector
        dw      reset                     ;  TIM1 Channel 0 Vector
        dw      reset                     ;  PLL Vector
        dw      reset                     ;  IRQ1 Vector
        dw      reset                     ;  SWI Vector
        dw      reset                     ;  Reset Vector
;-----------------------------------------------------------------------------------------
```

sCo Pilot
Human Interface Enhancement

Floppy disk

The following floppy disk contains the required files for the sCo-Pilot. The file names and descriptions are tabulated below. The disk is in MS-DOS HD format.

| File name | Description | Format |
|---|---|---|
| SPI_IO.ABL | The ABEL source file for the SPI I/O expander CPLD | ASCII text |
| SCOPILOT.ASM | The CPU08 assembly source file for the sCo-Pilot | ASCII text |
| ABSTRACT.TXT | The project abstract for the sCo-Pilot project | ASCII text |