KPC-TM
and
KPC488.2TM
Trigger Master™
Interfaces

User Guide

for the

KPC-TM and KPC488.2TM Trigger Master™ Interfaces

Revision A - February 1993

Part Number: 24461

KEITHLEY DATA ACQUISITION - Keithley MetraByte/Asyst
440 Myles Standish Blvd., Taunton, MA 02780
TEL. 508/880-3000, FAX 508/880-0179

Warranty Information

All products manufactured by Keithley Data Acquisition are warranted against defective materials and workmanship for a period of one year from the date of delivery to the original purchaser. Any product that is found to be defective within the warranty period will, at the option of the manufacturer, be repaired or replaced. This warranty does not apply to products damaged by improper use.

Warning

Keithley Data Acquisition assumes no liability for damages consequent to the use of this product. This product is not designed with components of a level of reliability suitable for use in life support or critical applications.

Disclaimer

Information furnished by Keithley Data Acquisition is believed to be accurate and reliable. However, Keithley Data Acquisition assumes no responsibility for the use of such information nor for any infringements of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of Keithley Data Acquisition.

Note:

Keithley MetraByte™, **Trigger Master™**, and **Trigger-Link™** are trademarks of Keithley Data Acquisition.

BASIC™ is a trademark of Dartmouth College.

IBM(R) and **Micro Channel Architecture(R)** are registered trademarks of International Business Machines Corporation.

PC, XT, AT, and **PS/2** are trademarks of International Business Machines Corporation.

Microsoft(R) is a registered trademark of Microsoft Corporation.

Turbo C(R) and **TurboPascal(R)** are registered trademarks of Borland International.

Contents

Contents

APPENDIXES

Chapter 1
INTRODUCTION

1.1 GENERAL DESCRIPTION

Trigger Master™ is a system trigger controller for instruments and data acquisition boards with external triggering. Trigger Master supports a variety of trigger functions occurring in data acquisition systems. Trigger Master monitors trigger inputs, creates delays, and generates trigger outputs.

An outstanding feature of Trigger Master is its ability to run programs from its own memory and to generate interrupts at appropriate steps of program execution. This allows Trigger Master, with some adjunct data acquisition hardware, to operate as an autonomous data acquisition system in your personal computer (PC) while you use your PC for other purposes.

Trigger Master is implemented on two boards:

- The KPC-TM board provides Trigger Master as a stand-alone plug-in board for the PC™/XT™/AT™ computer.
- The KPC488.2TM board combines Trigger Master with the high performance KPC488.2AT GPIB interface to provide a standard interface with GPIB instruments.

Note: This manual describes Trigger Master. For information on the IEEE-488 functions, refer to the manual *IEEE 488 Interface Boards* which accompanies the KPC488.2TM board.

Figure 1-1 is a block diagram of Trigger Master.

Physically, Trigger Master is equipped with an 8-pin MicroDIN connector which has 6 trigger lines and 2 ground lines. In addition to Trigger-Link™ introduced later in this chapter, the MicroDIN can connect directly to standard BNC connectors using the 8502 Trigger-Link Adapter.

Chapter 2 describes configuring and installing the boards.

Trigger Master employs proprietary chips containing state machines that run from an 8-MHz clock. This allows Trigger Master to respond to a change of trigger inputs by generating a trigger output within 1.25 us. The state machine coordinates all functions including the interfaces to Trigger-Link and the PC bus.

A PC program controls Trigger Master using 11 SCPI-like commands which a driver transforms to microcode instructions, sending them to Trigger Master. Trigger Master supports three modes of operation:

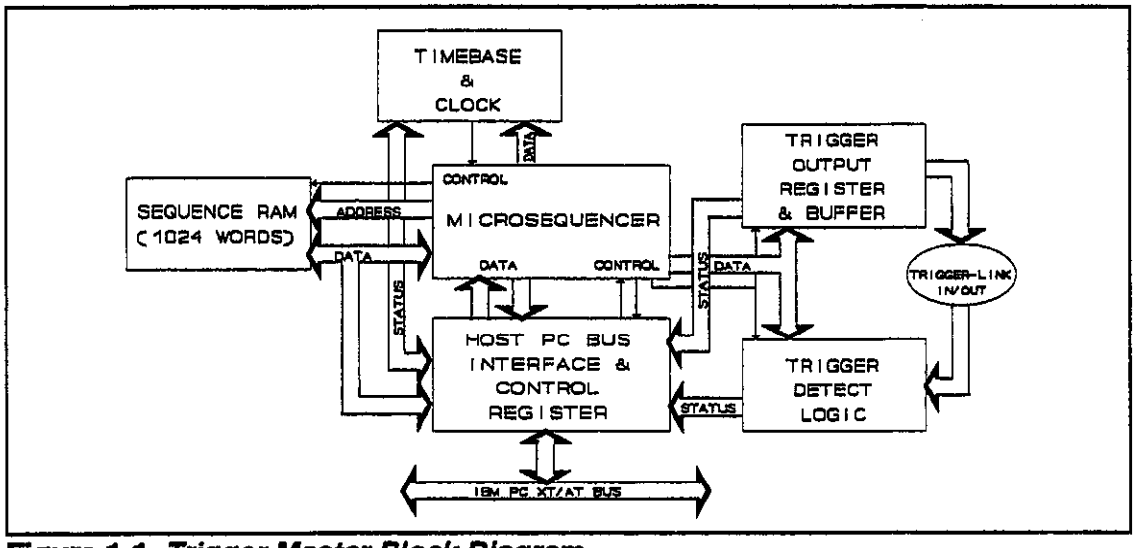


Figure 1-1 ***Trigger Master Block Diagram***

- One mode of operation, *Immediate Mode*, allows some commands to be executed immediately by Trigger Master.

- The second mode of operation, *Program Mode*, allows Trigger Master to store commands in a 1-Kbyte program RAM as they are received for later execution. Storing the program into RAM provides extremely powerful performance, since the state machine can perform two level looping (one nested loop); for repeated operations, the state machine uses available counters. This mode allows you to start program storage and execution at any memory location, therefore allowing several small programs to be resident in memory and allowing you to start them as required.

- The third mode of operation, *Run Mode*, occurs while the Trigger Master executes a program.

A user can generate trigger programs using any of the following methods:

- Pass command strings from the user program to the Trigger Master driver.

- Write trigger programs using an ASCII-output word processor and then compile the programs with STCCOMP.EXE.

- Program interactively using PLAYWIN.EXE and PLAYDOS.EXE, running, respectively, from the Windows and DOS environments.

Any word processor that provides ASCII output allows you to write Trigger Master programs, store them in ASCII, and then "compile" the output using the STCCOMP.EXE program. Chapter 6 describes this procedure. The resulting "object" file is then easily loaded into Trigger Master program memory and executed.

The Trigger Master driver supports up to four boards simultaneously. The driver accepts the following commands:

ARM	Set trigger input condition and wait for trigger
BEGIN	Enter program mode
CONT	Restart a halted program at the next step
DO	Mark the start of a program sequence that is to be performed as a loop
END	Mark the end of a program mode sequence with a HALT and return to immediate mode
FLAG	Write a value to a diagnostic flag register (to trace program execution)
HALT	Halt Trigger Master operation
LOOP	Mark the end of a program loop
TRIG	Generate triggers
WAIT	Cause a program to execute a time delay
X	Begin program execution

The program STCRUNC.OBJ, described in Chapter 6, builds terminate-and-stay-resident programs (TSRs) for programs written in C. These programs are driven by interrupts generated by the Trigger Master; the programs can run in the background in a DOS environment while you run other programs from DOS.

Chapter 3 describes the PLAYDOS.EXE and PLAYWIN.EXE programs which allow you to become familiar with the commands and requests, test your hardware without doing any programming, and generate Trigger Master programs for future execution. PLAYDOS.EXE runs in the DOS environment, while PLAYWIN.EXE executes from Windows. Refer to Appendix B for a quick start on the commands and Chapter 3 for a detailed description of the commands.

In addition to sending commands to Trigger Master you can also request information from Trigger Master. Trigger Master supports the following requests:

ARM	Return information about the trigger detect circuitry
CONT	Return the current Trigger Master program position
FLAG	Read the value from the diagnostic flag register
LOOP	Return the execution status of a Trigger Master program loop
STATUS	Return the value from the Trigger Master status register
TRIG	Return information about the trigger output circuitry
WAIT	Return the remaining delay time

Refer to Appendix C for a quick start on the requests and see Chapter 3 for a detailed description of the requests. Chapter 4 provides programming examples of the commands and requests in the supported languages.

1.2 DISTRIBUTION SOFTWARE

This manual refers to Trigger Master software as the *Distribution Software*. The Distribution Software contains utility files and driver files. Chapter 3 discusses these files.

1.3 SPECIFICATIONS

Channels:	6 Input/Output
Basic Functions:	Trigger Detection Trigger Generation Delay Generation PC Interrupts

Micro Sequencer:

Modes:	Program, Immediate, or Run
Looping:	2 Levels
Loop Repeat:	1 - 4096
Trigger Repeat:	1 - 4096
Sequencer RAM:	1024 bytes
Time Base Drift:	100 ppm max
Trigger Input Pulse Width:	400 ns min
Trigger Output Pulse Width:	5 us
Detection Latency:	900 ns max
Async Trigger Latency:	2.2 us max (trigger in to trigger out)

Programmable Delay:

Range	Resolution
1 us to 65.536 ms	1 us
10 us to 655.36 ms	10 us
100 us to 6.5546 sec	100 us
1 ms to 65.536 sec	1 ms

Trigger Connector:	8-pin microDIN
Modes:	Sync, semi-sync (Trigger-Link), async

Power Consumption:

KPC-TM:	450 mA @5V max
KPC488.2TM:	1650 mA @5V max

Environmental:

Operating Temperature:	0 to +70 C
Storage Temperature:	-25 to +85 C
Humidity:	0 to 95%, non-condensing

Dimensions:

KPC-TM:	4.25 in H x 5.0 in. W (half-slot)
KPC488.2TM:	4.25 in. H x 13.25 in. W (full slot)

Software:

Call Driver Languages:	BASIC™, QuickBASIC, C, TurboPascal®, VisualBASIC for DOS.
Trigger Master DLL:	Operation with Windows 3.x languages. Includes VisualBASIC, Borland C++, C for Windows, and TurboPascal.

1.4 Trigger-Link

Trigger Master supports Trigger-Link™, which brings a new dimension of flexibility, accuracy, and throughput to test and instrumentation systems. This section introduces the features of Trigger-Link.

You can easily change the trigger paths between instruments using Trigger-Link with GPIB commands. The precise trigger signals on Trigger-Link enhance the accuracy and throughput of the system. Even systems which contain instruments without Trigger-Link can benefit by using the 8502 Trigger-Link Adapter. The 8502 Trigger-Link Adapter is the interface to Trigger-Link for conventional BNC trigger connections. Figure 1-2 illustrates various Trigger-Link configurations.

Mechanically, Trigger-Link consists of a cable with six signal paths and two grounds which can be permanently daisy-chained between a group of instruments. The signal paths convey trigger signals between instruments. With GPIB commands, instruments can be configured to use one or more of the signal paths in a variety of modes. Thus the trigger configuration of a group of instruments can be easily altered by software to suit the requirements of a particular test.

Electrically, Trigger-Link provides paths for trigger pulses between instruments, thus eliminating the latencies involved with coordinating trigger functions with the GPIB interface. This greatly increases system throughput and measurement preciseness for all instrument systems. Instruments, which completely embrace the Trigger-Link standard, employ a fast-track link between the trigger input and function execution and between function conclusion and acknowledge output. Today, many instruments service the trigger/acknowledge connectors periodically using a microprocessor that performs other functions. However, this procedure results in unknown and variable timing latencies as well as slower response.

Trigger-Link supports three trigger modes:

SYNC MODE A source sends a trigger pulse or sequence of pulses to synchronize the activities of one or more receivers. There is no acknowledgement from the receiver(s) that they have received a pulse and are responding properly to the trigger(s).

ASYNC MODE The conventional two-wire handshake protocol where triggers are sent on one line and the receiver acknowledges on a second line. Conceivably, multiple instruments could share a common trigger source, but each instrument would require a separate acknowledgement line.

SEMI-SYNC An innovative extension of the async mode which allows a single trigger source and multiple receivers to carry out a handshake on a single line. The trigger source will pulse the trigger line to an active state for about 5 us. Upon receipt of the trigger all receivers will hold the trigger line in the active state before the trigger source goes inactive. Each individual receiver will continue to hold the line active until that receiver is ready to acknowledge it has completed its task. When the line goes inactive the trigger source will know that all receivers have completed their tasks.

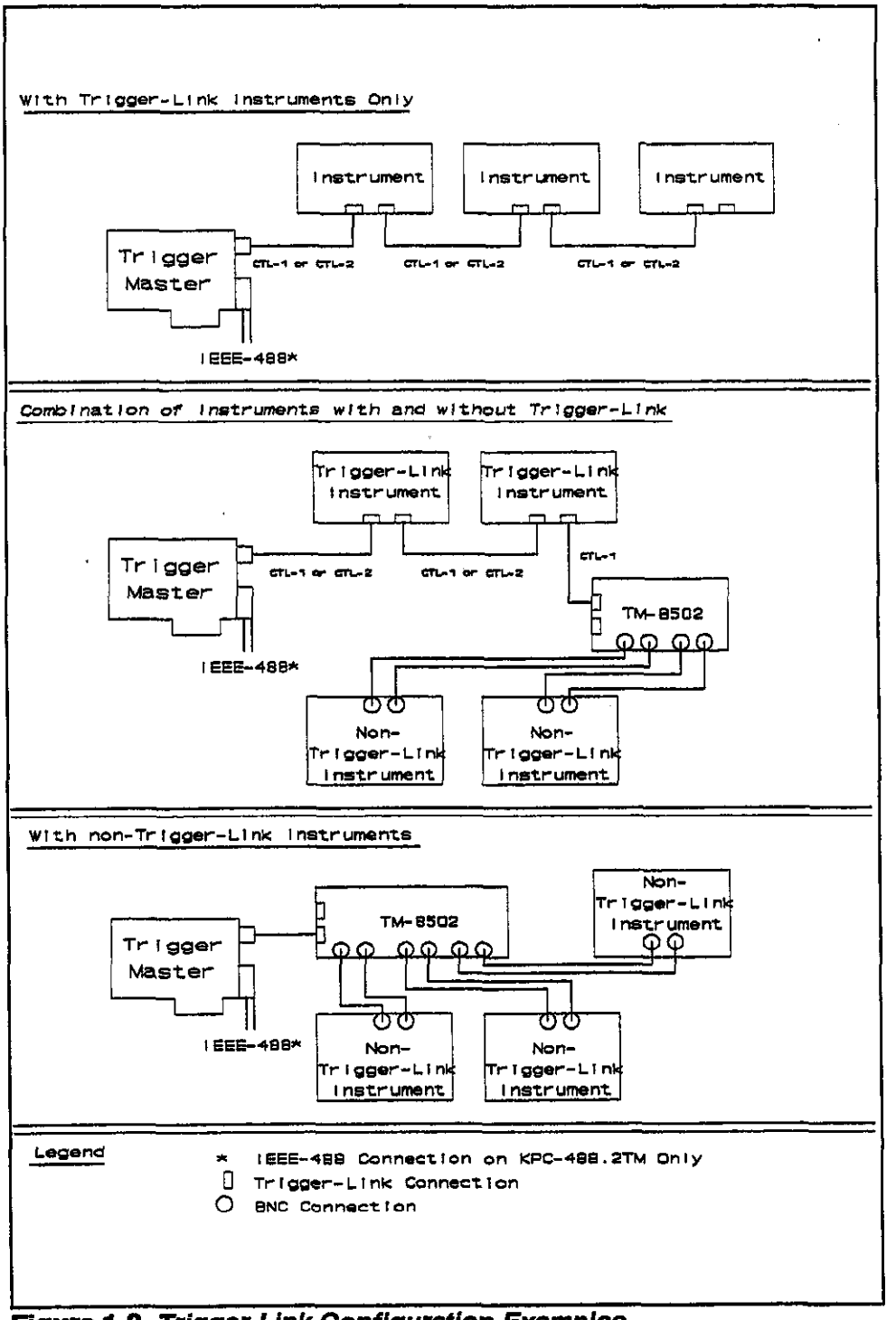


Figure 1-2 ***Trigger-Link Configuration Examples***

Chapter 2
INSTALLING Trigger Master

2.1 INTRODUCTION

The installation of Trigger Master includes the following:

- inspecting the KPC-TM and KPC488.2TM boards.
- setting jumpers and switches on the KPC-TM and KPC488.2TM boards.
- inserting the two boards into your PC.
- attaching all cables to the boards.
- running the PLAYDOS.EXE and PLAYWIN.EXE programs to exercise and verify proper operation of the boards.

2.2 INSPECTING THE BOARDS

Remove each board from their protective packaging by grasping the metal rear panel and removing the board from the anti-static bubble package.

Note: You should handle the boards only by their edges. A static electric discharge can damage the integrated circuits on the boards.

2.3 SETTING UP THE KPC-TM BOARD

The KPC-TM board is a stand-alone system trigger controller which requires four byte-wide I/O addresses. The board contains a switch to set up the base address in increments of four bytes. This switch decodes address lines A9 to A2. The KPC-TM board ships with a default setting of 310(hexadecimal) as shown in Figure 2-1. The position OFF corresponds to a logical 1 and the position ON to a logical 0. Table 2-1 lists the base addresses with the appropriate switch settings for each address.

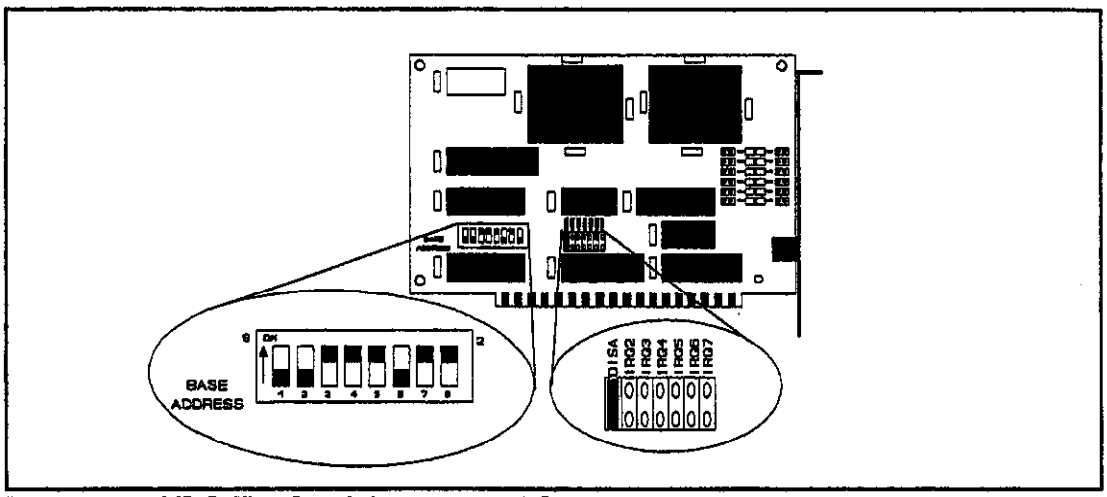


Figure 2-1 ***KPC-TM Card Jumper and Switch Locations.***

You can configure the KPC-TM board to generate interrupts on levels 2 through 7 by changing the jumper on jumper block J2. Trigger Master ships with the interrupts disabled as shown in Figure 2-1.

Note: The KPC-TM Base Address switch settings are position values only. Refer to Table 2-1 for the corresponding Address Line values.

Address			Switch
Decimal	Hexadecimal	Line	Value
512	200	9	1
256	100	8	2
128	80	7	3
64	40	6	4
32	20	5	5
16	10	4	6
8	8	3	7
4	4	2	8

Table 2-1 ***Base Address Switch Settings***

2.4 SETTING UP THE KPC488.2TM BOARD

The KPC488.2TM board implements the trigger master control function and GPIB control function on the same board. Both the trigger master and GPIB functions can generate interrupts. This manual describes the settings for the interrupt jumpers and the switch and jumper settings for the GPIB function. Refer to the accompanying user manual, *IEEE 488 Interface Boards*, for further information. Figure 2-2 shows the locations of the jumpers and switches on the KPC488.2TM card.

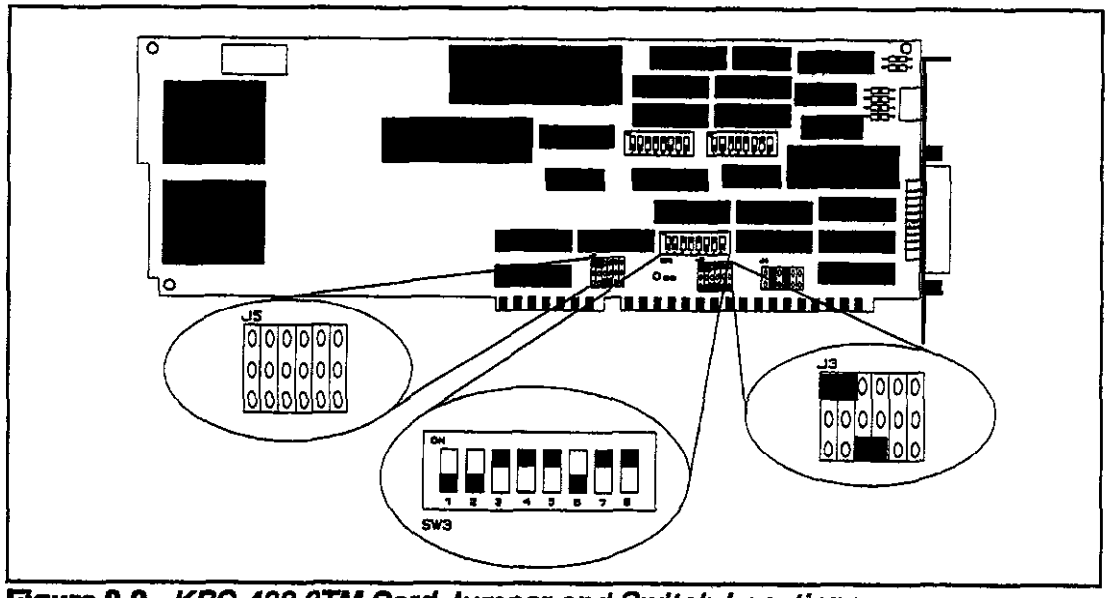


Figure 2-2 ***KPC-488.2TM Card Jumper and Switch Locations***

To prevent the same level from being used by both functions, use the interrupt three-row jumper blocks to select the interrupt.

Use jumper blocks J5 and J3, shown in Figures 3-2 and 3-3, to set the interrupts levels for the GPIB and Trigger Master. The top and middle rows of the jumper blocks set the GPIB interrupt level, and the bottom and middle rows set the Trigger Master level.

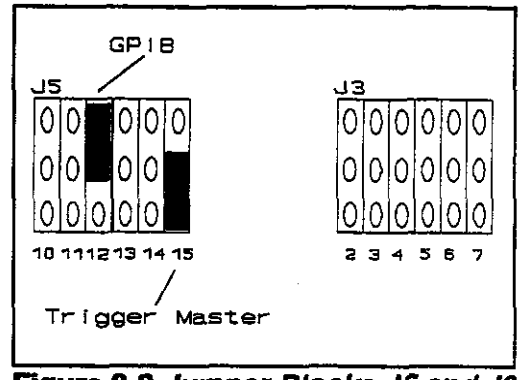


Figure 2-3 ***Jumper Blocks J5 and J3 Example***

- Placing a jumper vertically on the upper and middle rows enables an interrupt level for the GPIB. Placing a jumper vertically on the middle and bottom rows enables an interrupt level for Trigger Master.
- Placing the GPIB jumper horizontally on the upper row disables the GPIB interrupt and placing the trigger master jumper horizontally on the bottom row disables the trigger master interrupt.

Figure 2-3 illustrates enabling interrupt level 12 for the GPIB and enabling interrupt level 15 for Trigger Master (note that level 13 is unavailable on the PC/AT bus).

Trigger Master requires four byte-wide addresses; use switch SW3 to set the base address in increments of 4. The switch decodes address lines A9 through A2. The position OFF corresponds to a logical 1 and the position ON to a logical 0. The boards ship with a default setting of 310(hexadecimal) as shown in Figure 2-2.

2.5 INSTALLING Trigger Master

Follow this procedure to install Trigger Master into your PC.

1. Turn the PC power switch to OFF. Unplug the power cord and disconnect all cables from the rear of the system unit.
2. Remove the cover mounting screws from the rear of the system unit.
3. Slide the system unit cover forward. When the cover can go no further, tilt it up and remove it from the base.

   Note: Install the KPC488.2TM board into a 16-bit slot; the KPC-TM board can use an 8-bit slot.
4. Remove the rear panel cover screw from one of the computer's add-on slots.
5. Press the board firmly into the main board expansion slot.
6. Seat Trigger Master cable in the Micro-DIN connector and align the board before tightening the rear panel mounting screw.
7. Secure the board with the rear panel mounting screw.
8. Align the rear cover, sliding it back into place. Reinstall the mounting screws.
9. Turn on the PC.
10. Make a backup of all application diskettes before copying the applications to your PC's hard disk.
11. Run the PLAYDOS.EXE and PLAYWIN.EXE programs to exercise the boards.

Since the Trigger Master cable should seat completely in the MICRO DIN connector, you may want to insert the cable and test the alignment of the card before tightening the screw holding the bracket.

2.6 RUNNING PLAYDOS.EXE or PLAYWIN.EXE

Check the hardware and exercise Trigger Master after installation by running either the PLAYDOS.EXE or PLAYWIN.EXE program. PLAYDOS.EXE runs from the DOS environment and PLAYWIN.EXE executes in the Windows environment.

PLAYDOS.EXE and PLAYWIN.EXE provide a menu interface to the standard Trigger Master driver calls without requiring the use of a programming language. You can also use these programs to create and document Trigger Master programs.

The driver supports the following calls, which are described in detail in Chapter 3:

STCCMD	Sends Trigger Master commands to the active Trigger Master either for immediate execution or storage in Trigger Master program memory.
STCDUMP	Saves the contents of the active Trigger Master program memory to a binary file.
STCINIT	Checks for the presence of a Trigger Master. If Trigger Master is found, it is initialized and made active. The driver can simultaneously control up to four boards in the same computer.
STCLOAD	Loads the contents of a binary file into Trigger Master program memory.
STCSET	Selects a different Trigger Master to become active (this Trigger Master must have been initialized).
STCSTAT	Requests status information from the active Trigger Master.

The PLAYDOS.EXE and PLAYWIN.EXE programs operate by listing these calls in a main menu. When you select a call from the main menu, a form appropriate to that call appears. Windows or buttons are provided, where necessary, for entering data or making selections specific to that call. After filling in any blanks on the form, press the button for the call to start execution. Any error messages returned from the driver are displayed.

Each form contains a "Help" button that provides assistance on that form. Context-sensitive help is also available. To enable context-sensitive help, you must first include the files PLAYWIN.HLP or PLAYDOS.HLP in the same directory as the executible files PLAYWIN.EXE or PLAYDOS.EXE. Access help by pressing <F1> in PLAYWIN.EXE or Shift+<F1> from PLAYDOS.EXE.

If you select the STCCMD command from the main menu, a new menu appears listing all the possible commands you may include with STCCMD. When you select a command, another form appears that is specific to the that command. When you push the STCCMD button, a window displays the command string sent to the driver. The "Man" selection is an option that allows you to manually enter your own command string (the string can contain multiple commands). If an error occurs during execution of the STCCMD command, the driver displays an error message and places the value "**" into your command string at the point the driver detected the error.

The "Man" form has a button that enables the "Paste" option. With this option on, commands sent without errors from other command forms will also appear in the "Man" form command window. This procedure allows you to document a Trigger Master program as you create it. You can save the contents Trigger Master memory with the STCDUMP command. Since the STCDUMP command creates a non-readable binary file, you can use the "paste" provision to save the contents of the command window to obtain a file of the command sequence used to generate the program.

When you execute the STCSTAT command, the command normally returns the status only once. When you make a status request from PLAYWIN.EXE or PLAYDOS.EXE, a form is created which continually reads and displays the status. This procedure allows you to follow the status changes as Trigger Master executes a program or command.

You can also watch the CONT STATUS request during program creation to determine where the next program instruction will be loaded in memory.

The PLAYDOS.EXE and PLAYWIN.EXE programs provide two methods of choosing a different call or command and exiting a form:

- Choose a new form without closing the current form. When you bring the current form back up, the data you previously entered is still in the form. This procedure simplifies the manual entry of commands, since you can recall the previous ten commands by using the arrow keys at the side of the entry window. You can also size and position forms to suit your needs.

- Click on the upper-lefthand button and choose the close option from the menu displayed. When you next bring up the form, the data is reinitialized. To exit the program, close the main menu (this automatically closes all forms).

Chapter 3
USING THE Trigger Master DRIVER

3.1 INTRODUCTION

The Trigger Master driver supports the following languages: BASICA, Microsoft(R) QuickBASIC, VisualBASIC (for DOS and Windows), Microsoft(R) C, C++, and C for Windows, and Borland(R) TurboPascal and TurboPascal for Windows.

- For Windows applications, a Trigger Master DLL is placed in your Windows directory.
- For DOS environment applications in QuickBASIC, VisualBASIC, C, and TurboPascal, a Trigger Master file is linked with the application program.
- For BASICA, a Trigger Master binary file is loaded with the program.

For maximum efficiency with all languages, the application program makes a direct call to the appropriate driver code using the following calls:

STCINIT	Checks for and initializes a Trigger Master at a specified board address and then sets the driver to control that board. The driver can simultaneously control up to four boards.
STCSET	Switches Trigger Master control to a different board. The board must have been initialized.
STCCMD	Sends commands to Trigger Master.
STCSTAT	Requests status from Trigger Master.
STCLOAD	Loads a binary file into Trigger Master program memory.
STCDUMP	Saves Trigger Master program memory to a binary file.

3.2 USING THE DRIVER

The following sections describe "installing" the driver in each of the supported languages.

Using the Driver with BASICA

Run the following code segment to load the driver In BASICA:

```
260 CLEAR , 62*1024            ' leave 2048 for interface.
270 DEF SEG = 0
280 SG = 256 * PEEK(&H511) + PEEK(&H510)
```

```
290 SG = SG + &H1100
300 DEF SEG = SG              ' DEF SEG must be set = SG in order
310                           ' to call into the driver.
320 BLOAD "STCBAS.BIN", 0
330 ' The following integer variables are set to the offset
340 ' values required.
350 STCINIT = 0               ' Initialize Trigger Master.
360 STCSET = 3                ' Switch to already initialized
365                           ' Trigger Master.
370 STCCMD = 6                ' Send command to Trigger Master.
380 STCSTAT = 9               ' Request information from
385                           ' Trigger Master.
390 STCLOAD = 12              ' Load a program in Trigger Master
395                           ' memory for file.
400 STCDUMP = 15              ' Save Trigger Master memory
405                           ' to a file.
410 '
```

Accessing the Driver from C

For Microsoft(R) C, compile your program and link the resulting object file with *stcc.lib* using a command such as the following:

```
LINK your_file,,,stcc;
```

You will need to include at least the function prototypes from *stc.h* in your program.

> **Note:** For C or C++ programs running under Windows, do not link the program to *stcc.lib*. Instead, copy the file *stclib.dll* to your Windows directory; the function prototypes in *stclib.dll* are identical to those in *stc.h*.

Accessing the Driver from QuickBASIC and VisualBASIC

QuickBASIC

Use one of the following methods when building an executable program from the DOS prompt:

- If you use Version 4 or greater or Version 7 with near strings, link your program to *stcqb.lib*.
- If you use Version 7 with far strings (compiled with /Fs), link your program to *stcqbx.lib*.

To run your program in the appropriate QuickBASIC environment, load the program with one of the following files:

- For Version 7, use the file *stcqb7.qlb*.
- For Version 4.0 to version 7, use the file *stcqb4.qlb*.

In either case, you must include at least the function protocols from the file *stcqb.bi* in your program.

VisualBASIC

Place VisualBASIC function declarations in the Global section.

- For DOS applications:

  To run a program from the environment, use the command **VBDOS /LSTCVBD.QLB** to load VisualBASIC with the Quick Library *STCVBD.QLB*. Include function prototypes for the Trigger Master calls by incorporating the Trigger Master include files with the statement **'INCLUDE stcvbd.bi**. The file *STCVBD.BI* also includes error code definitions and an array to hold error message strings. If you do not need to display error messages, delete the array; otherwise, use the code in *STCVBDI.BAS* to initialize the array. You can build an executable file from the environment or from the command line. To build the file from the command line, first compile each form or BASIC module of your project using the command line compiler. Then, from the command line, link the resulting object modules with the Trigger Master VisualBASIC for DOS library *STCVBD.LIB* to produce the executable file. The following command line example illustrates the production of the file EXVBD.MAK:

```
BC EXVBD.FRM
BC EXVBD.BAS
LINK EXVBD EXVBDI,,,STCVBD.LIB;
```

- For Windows applications:

  Copy the file *STCLIB.DLL* to your Windows directory. You will need to include function prototypes for the Trigger Master calls in your global data. The file *STCVBW.TXT* includes function declarations appropriate for the Global section of a VisualBASIC for Windows application. This file also defines error codes and an error array. If you wish to use the error array, you must include code from *STCVBI.TXT* in the load procedure of your first form. The difference between the *STCVBW.TXT* file and the DOS program file *STCVBD.TXT* is in the function declarations which are appropriate to *STCLIB.DLL*.

Accessing the Driver from TurboPascal

Access the driver from a TurboPascal program (version 6) by including the following statements in your program.

```
{$I stctp.inc} { Function prototype for stctp.obj }
{$L stctp.obj} { Link with stctp.obj }
```

For TurboPascal for Windows, copy the file STCLIB.DLL to your Windows directory and include the following statement in your program.

```
{$I STCTPW.INC} { Function prototype for STCLIB.DLL }
```

The include files contain function prototypes for the calls and define error codes. If you want to display error messages, include an array for error strings. The file STCTPI.PAS contains code that you can add to your program to initialize the error array.

3.3 STCINIT

STCINIT checks for the presence of a board by writing to Trigger Master program memory. STCINIT initially writes the value 0 and then increments this value through 255(decimal). As the memory register is written, Trigger Master automatically increments the on-board memory location so successive memory locations are loaded with increasing values until reaching 255. At this point, STCINIT resets to 0 and repeats the process until it writes to all 1024 memory locations.

STCINIT then reads back the values. The process of reading the values automatically increments the memory location. If the read value matches the write value, memory is cleared and STCINIT returns with no error.

If the board is present, it becomes "active" so that all subsequent commands or requests to the driver will be sent to that board. Up to four boards can be initialized; as each board is initialized, it becomes the "active" board. To reactivate an initialized board, use the STCSET call.

> **Note:** Every time you run a program, you must initialize Trigger Master by calling STCINIT before making any other calls. During initialization, only Trigger Master program memory is cleared; other registers may retain values from previous program execution.

STCINIT requires three arguments as follows:

```
STCINIT(i1%,a1%,i2%)
```

The variables definitions and ranges are as follows:

Variable	Definition	Range
i1% (integer)	Trigger Master reference number	0 - 3
a1%	Trigger Master board address	0 - 7FC
i2% (integer)	Indicates success of call	0 = successful non0 = unsuccessful
	(refer to Appendix A for Trigger Master error messages.)	

Calling STCINIT from BASICA

Use the following BASICA code segment to initialize a Trigger Master at address 30(hex):

```
450 ERRNUM% = 0         'error return variable
610 BRDNUM% = 0
620 BRDADDR% = &H310
630 PRINT "Initialize board "; BRDNUM%; " at address ";
640 PRINT HEX$(BRDADDR%); " hex"
650 CALL STCINIT(BRDNUM%, BRDADDR%, ERRNUM%)
660 IF ERRNUM% THEN GOTO 2130
```

> **Note:** You must define all arguments for the STCINIT call (BRDNUM%, BRDADDR%, ERRNUM%) before making the call.

Calling STCINIT from C

Use the following code segment to initialize Trigger Master in C:

```
int    brd_num = 0;          // integer for Trigger Master board number
int    brd_addr = 0x30;      // integer for Trigger Master board address
int    err;                  // integer to receive error code
stcinit(brd_num, brd_addr, &err);
```

In C you can send values to the driver by placing them directly in the call. For example, you could use the following code:

```
stcinit(0, 0x300, &err);
```

Since the driver returns err, you must have a predefined variable to receive its value.

Calling STCINIT from QuickBASIC and VisualBASIC

Use the following code to call STCINIT from QuickBASIC and VisualBASIC:

```
DIM rerr       AS INTEGER
DIM BrdNum     AS INTEGER
DIM brdAddr    AS INTEGER

BrdNum = 0
brdAddr = &H310
PRINT "Initialize board ";BrdNum;" at address ";
PRINT HEX$(brdAddr); " hex"
CALL stcinit(BrdNum, brdAddr, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

In QuickBASIC and VisualBASIC, you can send values to the driver by placing them directly in the call. For example, you could use the following code:

```
CALL stcinit(0, &H310, rerr)
```

Since the driver returns rerr, you must have a predefined variable to receive its value.

Calling STCINIT from TurboPascal and TurboPascal for Windows

The following code illustrates calling STCINIT from a TurboPascal or TurboPascal for Windows program:

```
stcinit(1,$314,err);
```

3.4 STCSET

Use STCSET in multiboard systems to switch the driver from one board to another. The boards must have been previously initialized with the STCINIT command (described in the previous section). STCSET accepts two arguments as follows:

```
STCSET(i1%,i2%)
```

The argument i1% is an integer in the range of 0 through 3 that identifies the board.

The argument i2% is an integer that receives an error code. A value of 0 indicates no error (refer to Appendix A for a list of error messages).

Calling STCSET from BASICA

The following code segment illustrates the use of STCSET in BASICA:

```
1200 '********* RETURN TO BRD 0 AND SEND TRIGGERS *********
1210 '
1220 BRDNUM% = 0
1230 PRINT "Switch driver control back to Trigger Master #"; BRDNUM%
1240 CALL STCSET(BRDNUM%, ERRNUM%)
1250 IF ERRNUM% THEN GOTO 2130
1260 '

1320 '
1330 '***** GO BACK TO BRD 1 AND WAIT FOR TRIGGERS ********
1340 '
1350 BRDNUM% = 1
1360 PRINT "Switch driver control back to Trigger Master #"; BRDNUM%
1370 CALL STCSET(BRDNUM%, ERRNUM%)
1380 IF ERRNUM% THEN GOTO 2130
1390 '
```

Calling STCSET from C

The following example illustrates calling STCSET within C:

```
printf("Switch driver control back to Trigger Master #0\n");
stcset(0,&err);
if (err != NO_ERROR) err_handler(err);

printf("Switch driver control to Trigger Master #1\n");
stcset(1,&err);
if (err != NO_ERROR) err_handler(err);
```

Calling STCSET from QuickBASIC and VisualBASIC

The following code segment illustrates calling STCSET in QuickBasic and VisualBASIC.

```
BrdNum = 0
PRINT "Switch driver control back to Trigger Master #"; BrdNum
CALL stcset(BrdNum, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT
PRINT "Switch driver control to Trigger Master # 1"
CALL stcset(1, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

Calling STCSET from TurboPascal and TurboPascal for Windows

The following code illustrates calling STCSET from a TurboPascal or TurboPascal for Windows program.

```
stcset(1,err);
```

3.5 STCCMD

STCCMD sends commands to a board. STCCMD accepts three arguments as follows:

```
STCCMD(c1,i1%,i2%)
```

`c1` refers to a string that contains a command to be translated by the driver into microcode. The translated microcode is then sent to Trigger Master either for execution by the on-board state machine or for storage in the Trigger Master memory. The driver parses the command, checking for unknown commands, invalid syntax, or values out of range.

`i1%` is an integer that receives an error code. The driver returns a value indicating the status of errors: a value of 0 indicates no errors; refer to Appendix A for a list of error messages.

`i2%` is an integer that receives the position of the last character the driver parsed. Since the driver stops parsing the command line when it encounters an error, this value provides assistance for error debugging.

Note: Trigger Master executes commands as they are parsed. If Trigger Master discovers an error in a multiple-command string, it executes the commands prior to the error and then returns with an error code.

Refer to Appendix B for a quick introduction to the commands using examples. Run PLAYDOS.EXE or PLAYWIN.EXE to experiment with the commands in a non-program environment. See Appendix A for a description of the error messages.

Command Syntax

This section describes the syntax for the string argument (c1) in the STCCMD call. The STCCMD call supports the following commands: ARM, BEGIN, CONT, DO, END, FLAG, HALT, LOOP, TRIG, WAIT, and X.

General Information

The following rules apply to all STCCMD commands:

- Spell out the commands in their entirety (abbreviations are not supported).
- Complete each command with a semicolon (;).
- Use any combination of uppercase and lowercase letters within strings (the driver is insensitive to the case of characters).
- Do not use embedded spaces within a command. For example, the command "be gin;" is illegal.

Some examples of legal and illegal strings are as follows:

Illegal	Legal
beg;	Begin;
begin	beGiN;

In the example "beg", the driver returns the position 3 and the error 3 which correspond to the "Incomplete Command" error. As soon as the driver encounters an error it returns.

You can group multiple commands together using blank spaces (spaces, tabs, carriage returns, and line feeds) to improve readability. The following examples are equivalent:

```
begin;end;
        begin ;         end;
begin;
end;
```

The driver executes multiple commands contained in a single string when it encounters a semicolon. In the previous example, the driver executes the "begin" command when the driver parses the ";" following "begin". As a result, the driver executes commands one at a time. When the driver encounters an error, the previous commands have already been executed.

Line Numbers

The commands **ARM** and **TRIG** must be followed by one or more line numbers. The line numbers indicate which of the six trigger lines are armed to look for trigger inputs or will generate trigger outputs. When either command specifies multiple lines, the line values must be separated by commas. The following examples illustrate legal and illegal line values:

Legal	Illegal
trig 1, 5;	trig 1 5;
TRIG 6;	
trig3,2,1;	

Note: You may use the same number more than once in a command line. This does not alter the operation and is not flagged as an error.

Extensions

Certain commands accept extensions, which further define the command. You must spell out the entire extension (abbreviations are not accepted) and each extension must be preceded by a colon. The following example initiates Trigger Master program execution with interrupts enabled:

```
x:int;
```

For additional information on `x`, refer to the section "The Command Set".

Integer Arguments

Commands can be followed by one or more integers. An integer cannot contain embedded spaces, but the number may be separated from the command using one or more spaces. The integers in the following example indicate the memory location where program storage should start. The examples of legal and illegal command lines are:

Legal	Illegal
`begin123;`	`begin 1 23;`
`begin 123;`	
`begin  123`	`;`

The driver checks the range of the number and returns an error if the value is out of range.

Time Scales

Two commands require time arguments: WAIT and TRIG. The WAIT command generates time delays in a program running from Trigger Master memory (for further information on both commands, refer to the section "The Command Set"). The WAIT command syntax is:

```
wait n.n t;
```

`n.n` is a floating-point number that specifies a magnitude of time.

`t` indicates a time scale using one of the following three values:

s or S	seconds
m or M	milliseconds
u or U	microseconds

The magnitude of times (`n.n`) that can be used with the WAIT command range from 1 microsecond through 65.535 seconds. You can write time values using any choice of units. For example, the minimum and maximum times may be entered as any of the following values:

Minimum	***Maximum***
1 u	65535000u
.001 m	65535 m
0.000001 s	65.535 s

The magnitude of the time between any leading and trailing zeros must fit in a 16-bit counter; therefore, the range is between 1 and 65535. This limits the so-called resolution of the time scale (how fine a time increment you may specify). This allows the use of up to five digits for 1 to 65535, but only four digits for 6554 to 9999. For example, the values 0.06553400 s and 0.06553500 s are legal, but the value 0.06553600 s is illegal. The driver allows an increase of 0.00000100 s from the value 0.06553400 s to 0.06553500 s. If we attempt to increment the same amount to get to 0.06553600 s, the driver returns the error "TIME OVER RESOLUTION". This requires a rounding up to the next higher value in the digit to the left; in this example, the next larger value of time that can be specified is 0.06554000 s. This value represents an increment of 0.00000500; the next incremental value would be 0.00001000 (to 0.06555000 s).

The time resolution depends on the time value as shown in the following table:

Delay Range	Resolution
0.000001 s to 0.065535 s	1 usec
0.065540 s to 0.65535 s	10 usec
0.65540 s to 6.5535 s	100 usec
6.5540 s to 65.635 s	1000 usec

Note: If you attempt to enter the digits 65536, the driver returns the error "TIME OVER RESOLUTION". Unless your time scale is microseconds, you will be overrange as well.

Sending Commands in the Programming Languages

Commands are sent by calling STCCMD with the appropriate arguments. One of the arguments is the string specifying the command. The general features of those strings have been discussed in the previous section and the detailed use of each string will be discussed in the following section. This section shows the use of the call in each of the programming languages supported.

Calling STCCMD from BASICA

In BASICA, you must define all arguments before making the call.

```
450 ERRNUM% = 0      'error return variable
460 LASTCHR% = 0     'position of last character parsed by driver
470 '
. . .
700 INIT$="begin;"   ' begin program
710 PRINT "Send "; INIT$; " command"
720 CALL STCCMD(INIT$, ERRNUM%, LASTCHR%)
730 IF ERRNUM% THEN GOTO 2130
740 '
```

Calling STCCMD from C

In C, you must declare unsigned variables to receive information from the driver. However, the command string sent to the driver can be inserted directly in the call.

```
char begin_cmd[] = {"begin;"};      // begin program
unsigned    err                     // variable to receive error
unsigned    pos                     // variable to receive position
printf("Send %s command\n",begin_cmd);
stccmd(begin_cmd, &err, &pos);
if (err != NO_ERROR) err_handler(err);
printf("Send %s command\n","arm1:rep500;");
stccmd("arm1:rep500;", &err, &pos);
if (err != NO_ERROR) err_handler(err);
```

Calling STCCMD from QuickBASIC and VisualBASIC

In QuickBASIC and VisualBASIC you must declare integer variables to receive information from the driver. However, the command string sent to the driver can be inserted directly in the call.

```
begincmd$ = "begin;"        ' begin program

DIM rerr   AS INTEGER
DIM post   AS INTEGER

PRINT "Send "; begincmd$; " command"
CALL stccmd(begincmd$, rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Send arm1:rep500; command"
CALL stccmd("arm1:rep500;", rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

Calling STCCMD from TurboPascal and TurboPascal for Windows

In TurboPascal and TurboPascal for Windows you must also declare unsigned variables to receive information from the driver. However, the command string sent to the driver can be inserted directly in the call.

```
const
ArmCmd = 'arm1:rep500;';      { wait for 500 triggers on line 1 }
var
err:        ErrorCodes;       { integer to receive error number }
pos:        Word;             { integer to receive exit
                              position of parser }

stccmd(ArmCmd, err, pos);
if (err <> NO_ERROR)

stccmd('trig1:rep16:m35;', err, pos);
if (err <> NO_ERROR)
```

STCTP.LIB is a special interface to TurboPascal that accepts a standard TurboPascal string. *STCLIB.DLL* is a general DLL that expects so-called "C" strings. To send strings to the DLL in TurboPascal for Windows, create a string with a NULL character on the end outside the call, and pass the "second element" of the string in the call.

```
const
ArmCmd = 'arm1:rep500;' + #0;

var
err: ErrorCodes;
pos: Word;

Stccmd(ArmCmd[1], err, pos);
```

The Command Set

The command set consists of ARM, BEGIN, CONT, DO, END, FLAG, HALT, LOOP, TRIG, WAIT and X. The Trigger Master mode of operation determines how the commands are operated on:

- In immediate mode, Trigger Master executes commands as they are received. The commands BEGIN and CONT operate only in this mode.
- In program mode, the commands are stored in Trigger Master memory for future execution. The commands DO, END, LOOP, and WAIT operate only in this mode.
- In run mode, Trigger Master is executing a program and will only recognize the HALT command.

The remaining commands (ARM, FLAG, HALT, TRIG, and X) can be used in any mode of operation.

The following sections discuss the commands in detail.

ARM

The Trigger Master trigger-detect logic latches trigger edges on the six trigger input lines (the default is high-to-low transitions). The ARM command specifies a trigger input transition pattern to be detected. When the ARM command executes in a Trigger Master stored program, the program waits until the pattern is detected before proceeding to the next program step. In immediate mode, you can loop using the TRIG request (previously described) to detect the pattern.

The ARM command must be followed by one or more line numbers. If you enter multiple line numbers, separate each number using commas. Specifying the same line number more than once has no effect and is not flagged as an error. The following example waits for high-to-low trigger transitions on lines 1 and 5.

```
arm1,5;
```

You can specify the edge to latch by using a + (low-to-high) or - (high-to-low, the default) behind the line number. The following example waits for a low-to-high transition on line 1 and high-to-low transitions on lines 3 and 5.

```
arm 1+, 3-, 5;
```

Note: Because of the latching nature of the detect circuit, the edges need not occur simultaneously and the state of the trigger lines will generally differ from the pattern specified in ARM when the trigger condition is met.

In program mode, ARM supports the REP extension which allows you to wait for the trigger pattern to be repeated from 1 to 4096 times before proceeding. The following command line will wait for 23 repetitions of a trigger pattern of high-to-low transitions on lines 1 and 4. Each time the pattern is detected, latches are automatically cleared and re-armed.

```
arm 1, 4        :rep 23;
```

The general syntax for the ARM command is:

```
ARM {l[+|-| ],...}[:REP nn];
```

The variable `l` must be within the range 1 through 6 and the variable `nn` must be in the range 1 through 4096.

BEGIN

Use BEGIN only in immediate mode to switch Trigger Master to program mode. The BEGIN command optionally accepts a single argument: the Trigger Master program memory address where the program will start loading. This integer argument must be in the range 0 through 1023 (the default starting address, with no argument, is 0). The following example switches the driver to program mode and initializes the program counter to 40.

```
begin 40;
```

The general syntax for the BEGIN command is:

```
BEGIN [nn];
```

The variable `nn` must be in the range 0 through 1023.

Normally, you will start program loading and execution at address 0, but you may also have multiple programs in memory (terminate each program with an END or HALT command).

To load or execute multiple programs, you must know the locations of the instructions. Determine these locations either by building a program from PLAYWIN.EXE or PLAYDOS.EXE or by using the techniques described in Chapter 5, *Creating Programs for Trigger Master Memory.*

CONT

Use CONT in immediate mode to restart a program that was stopped by a HALT command or FLAG[nn]:INT command within the program. You cannot reliably restart a program that has been halted externally (from outside the program). If the program contains additional FLAG[nn]:INT commands, use the CONT command with the INT extension to clear the previous interrupt and arm Trigger Master to generate another interrupt. The following example restarts a program without interrupts:

```
cont;
```

This example restarts a program executing out of Trigger Master memory. The general syntax for the CONT command is:

```
CONT [:INT];
```

Note: If you enable Trigger Master to generate interrupts, you must supply your own interrupt service routines.

DO

Use DO in program mode to mark the start of a sequence of code which is to be repeated. The DO command requires a single argument, which is an integer in the range of 1 through 4096. The argument specifies the number of times the code sequence is to repeat. The code sequence must be terminated by the LOOP command (described later in this section). The driver allows two levels of loops; the driver will flag an error if you attempt to start a third level. In the following example, first command1 executes, then command2; this code sequence repeats 25 times:

```
do 25; command1; command2; loop;
```

The general DO syntax is:

```
DO nn;
```

nn is in the range 1 through 4096

END

Use END in program mode to insert a HALT (described later in this section) and return Trigger Master to immediate mode. The driver will return an error if you attempt to end a program that has DO commands which have not been resolved by a LOOP.

The general END syntax is:

```
END;
```

FLAG

Use FLAG in program mode to insert FLAG commands in a program. As the program executes, FLAG will write a byte to the flag register. The byte should be a value in the range of 0 through 255 (0 is the default). You can then use the FLAG request to read the flag register to determine which milestone your program has reached. If FLAG has the INT extension in program mode, FLAG causes the Trigger Master program to generate an interrupt and halt after writing the flag. Use FLAG in immediate mode for test only; this operation writes a byte in the range of 0 through 255 (0 is the default) to the Trigger Master flag register. For example, the following command writes 68 to the flag register:

```
flag 68;
```

The general syntax for the FLAG command is:

```
FLAG [nn][:INT];
```

The value nn is in the range of 0 through 255.

Use the extension :INT only in program mode.

Note: If you enable Trigger Master to generate interrupts, you must supply your own interrupt service routines.

HALT

The HALT command stops Trigger Master activity, disables Trigger Master hardware interrupts, and clears the hardware interrupt. When HALT executes during Trigger Master program execution, you can use the CONT command to restart the program on the next instruction. Use FLAG with the INT extension in a Trigger Master onboard program to halt a program, write the flag register, and generate an interrupt.

The general syntax for the HALT command is:

```
HALT;
```

LOOP

Use LOOP in program mode to terminate a loop initiated with the DO command.

The general syntax for the LOOP command is:

```
LOOP;
```

TRIG

Use TRIG to generate high-to-low trigger pulses (5 us active low). The TRIG command accepts from 1 through 6 arguments with each argument specifying a line number. Multiple line number arguments must be separated by commas. Repeating the same line number more than once has no effect and does not flag an error. The following example simultaneously generates 5-us pulses on lines 2 and 4.

```
trig 2,4;
```

You can generate a repetitive pulse train of 1 through 4096 pulses by using a REP extension. If you use the REP extension, you must also use the PER extension with a time; this time value specifies the REP period. The following TRIG example generates 72 5-us pulses on lines 2 and 3 with a repetition period of 16 milliseconds.

```
trig 2,3:rep72:per 16m;
```

The maximum period you can specify is 65.535 seconds and the minimum period is 10 microseconds. Refer to the section *Time Scales* in this chapter for a complete discussion on specifying times.

In program mode, you can use the SEMI extension alone, or in combination with the REP extension. The SEMI extension implements the semi-synchronous handshake. In semi-synchronous handshake mode, Trigger Master initiates a 5-us active-low trigger pulse on a line; handshaking devices then become active low within 5 us. To complete the handshake, the devices release the line when they have completed their activity. The handshake is complete when Trigger Master detects the low-to-high transition of the line. In the semi-synchronous handshake mode, the TRIG command does not require the PER extension.

The following example performs a semi-synchronous handshake on line 3.

```
trig 3:semi;
```

Note: The SEMI extension is only valid in program mode.

The general syntax for the TRIG command is:

```
TRIG {l,...}
[ [:REP nn{:PER rr t}:SEMI|:SEMI:PER rr t}]|[:SEMI] ];
```

The variables may contain the following values:

Variable	Definition	Range
l	Line number(s)	1 - 6
nn	Trigger pattern repeat	1 - 4096
rr	Real value of time	*
t	time scale	*

* Refer to the section *Time Scales* in this chapter for a complete discussion on specifying times and the range of values.

WAIT

Use WAIT in program mode to generate time delays during Trigger Master program execution. The WAIT command accepts a single time-delay argument. The following example generates a time delay of 32.3 ms (refer to the section *Time Scales* in this chapter for a complete discussion on specifying times).

```
wait 32.3m;
```

The general syntax for the WAIT command is:

```
WAIT rr t;
```

The variable may contain the following values:

Variable	Definition	Range
rr	Real value of time	*
t	time scale	*

* Refer to the section *Time Scales* in this chapter for a complete discussion on specifying times and the range of values.

X

Use X command to start a Trigger Master program executing at the specified location in Trigger Master program memory location. The X command accepts a single-integer argument which specifies the location of the program in memory. The integer must be in the range of 0 - 1023(decimal) (the default value is 0).

Normally, you will start program loading and execution at address 0, but you may also have multiple programs in memory (terminate each program with an END or HALT command).

To load or execute multiple programs, you must know the locations of the instructions. Determine these locations either by building a program from PLAYWIN.EXE or PLAYDOS.EXE or by using the techniques described in Chapter 5, *Creating Programs for Trigger Master Memory.*

Notes: If Trigger Master is operating in program mode, the driver inserts a HALT at the current memory location. Trigger Master returns to immediate mode prior to starting program execution.

In program mode, the driver returns an error if you attempt to end a program containing DO commands that have not been resolved by a LOOP command.

If your program contains a FLAG command with the INT extension, use the X command with the INT extension to clear a previous interrupt and enable Trigger Master to generate interrupts. The following example starts execution of a program at memory location 178(decimal):

```
x 178;
```

The general syntax for the X command is:

```
X [nn][:INT];
```

The acceptable range for the variable nn is 0 through 1023(decimal).

Note: If you enable Trigger Master to generate interrupts, you must supply your own interrupt service routines.

3.6 STCSTAT

Use the STCSTAT command to return Trigger Master register values. STCSTAT accepts four arguments in the following order:

Argument	Description
1	A string specifying the Trigger Master register value to be returned.
2	An integer to receive the error code returned by the driver after validating the request string.
3	An integer to receive the position of the last character parsed by the driver. You can use this value to pinpoint problems if the status indicates an error.
4	An array of two integers to receive the data.

Appendix B provides a quick introduction to STCSTAT. The PLAYDOS.EXE and PLAYWIN.EXE programs allow you to experiment with requests in a non-programming environment. Refer to Appendix A for a list of the possible error messages.

Note: Once the Trigger Master driver has filled one request, it immediately returns. This is an opposite condition to commands, which perform multiple requests before returning. Therefore, you should not place multiple requests in the same string.

Request Syntax

The STCSTAT command supports seven requests: ARM, CONT, FLAG, LOOP, STATUS, TRIG, and WAIT.

General

You must fully spell out the requests; the command does not accept abbreviations. Do not insert spaces within an argument and follow each request with a semicolon (;). The driver is insensitive to the case of letters and accepts any combination of uppercase and lowercase letters. The following examples illustrate legal and illegal forms of the command.

Legal	Illegal
Arm;	ar;
aRm;	ar m;
	arm

In the example "ar;" the driver returns the position 3 and the error 3 which corresponds to the "INCOMPLETE_COMMAND" error. As soon as it encounters and error, the driver returns.

Since the driver also returns after parsing one request, the driver fulfills the first string and ignores all others. In the following example, the driver returns the value from the `arm` request and ignores the `cont` request.

```
arm; cont;
```

Extensions

Certain requests can contain extensions, such as REP, which indicates the number of repetitions of the trigger input pattern remaining to be detected. For example

```
arm:rep;
```

Extensions must be preceded by a colon and spelled out in their entirety.

Making Requests In Programming Languages

Call the STCSTAT command with appropriate arguments to make requests. One of the arguments is the string containing the command. Refer to the previous sections for the general features of these strings; detailed use of the strings is discussed later in this chapter. This section illustrates the use of the call in each of the supported programming languages.

Calling STCSTAT from BASICA

The following segment illustrates calling STCSTAT from BASICA.

```
450 ERRNUM% = 0      'error return variable
460 LASTCHR% = 0     'position of last character parsed by driver
470 '

550 DIM RESULT%(2)   ' integer array to hold results from STCSTAT
560 '

1490 '
1500 '************** WAIT FOR DELAY ******************************
1510 '
1520 INIT$="wait;"
1530 CALL STCSTAT(INIT$, ERRNUM%, LASTCHR%, RESULT%(0))
1540 IF ERRNUM% THEN GOTO 2130
```

Note: All variables used in the call must be defined before the call is made.

Calling STCSTAT from C

The following segment illustrates calling STCSTAT from the C programming language.

```
unsigned ret_value[2];  // array to receive returned value
int  err;    // int to receive error number
int  pos;    // int to receive exit position of parser

stcstat("wait;", &err, &pos,ret_value);
if (err != NO_ERROR) err_handler(err);
```

Note: All values returned by the driver must be declared before the calls.

Calling STCSTAT from QuickBASIC and VisualBASIC

The following segment illustrates calling STCSTAT from QuickBASIC and VisualBASIC.

```
DIM RetVal(2) AS INTEGER
DIM rerr   AS INTEGER
DIM post   AS INTEGER

CALL stcstat("wait;", rerr, post, RetVal(0))
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

Note: All values returned by the driver must be declared before the calls.

Calling STCSTAT from TurboPascal and TurboPascal for Windows

The following code segment illustrates calling STCSTAT from TurboPascal and TurboPascal for Windows.

```
{ value returned by stcstat }

var
  RetValue: array [0..1] of WORD;
  err:      ErrorCodes;  { integer to receive error number }
  pos:      Word;  { integer to receive exit position of parser }

  stcstat('trig:rep;', err, pos,RetValue[0]);
  Writeln('Waiting for triggers; REP = ',RetValue[0]);
  if (err <> NO_ERROR)
    then ErrHandler(err);
```

Note: All values returned by the driver must be declared before the calls.

STCTP.LIB is a special interface to TurboPascal that accepts a standard TurboPascal string. *STCLIB.DLL* is a general DLL that expects so-called "C" strings. To send strings to the DLL in TurboPascal for Windows, create a string with a NULL character on the end outside the call, and pass the "second element" of the string in the call.

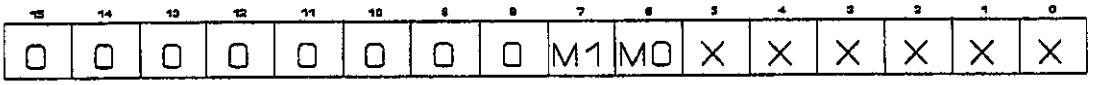
```
const
TrigReq = 'Trig:Rep;' + #0;

var
RetValue: array[0..1] of WORD;
err: ErrorCodes;
pos: Word;

Stcstat(TrigReq[1], err, pos, RetValue[0]);
```

Values Returned by STCSTAT

One STCSTAT argument is a reference to an array of two integers which receives the values from the call. Depending on the request, the first integer will receive either an 8-bit register result, a 10- or 12- bit counter value, or the 16-bit value from the delay counter.

Time delays in Trigger Master are generated by clocking a 16-bit counter using a clock derived from the Trigger Master 8-Mhz clock. This 8-Mhz clock is divided by 8 (to create a 1-us clock) and then by an additional factor of 1, 10, 100 or 1000 (the additional factor depends on the value in the upper two bits of the 8-bit Trigger Input/Prescaler register). In this manner, you can obtain time delays of 1 microsecond through 65.536 seconds.

When you request time delays you also receive the Trigger Input/Prescaler register value in the second integer. From this value, you can derive the multiplier. The bit pattern of the second integer is as follows:



Bits M1 and M0 determine the time multiplier as follows:

M1	M0	Multiplier
0	0	1
0	1	10
1	0	100
1	1	1000

When you request a time, you need only look at the second integer. The driver also uses the second integer to return a code which is helpful in interpreting the result of the operation. The following lists the various codes returned for different commands:

Code	Command
FFF0	ARM;
FFF1	ARM:POL;
FFF2	FLAG;
FFF3	STATUS;
FFF4	TRIG:LATCH;
FFF5	TRIG:IN
FF00	ARM:REP;
	TRIG:REP;
FF01	CONT;
FF02	LOOP;
FF03	LOOP:OUT;

Interpreting Values in BASICA

BASICA handles 16-bit values (integers) as signed values between -32768 and +32767 (Trigger Master returns integer values from unsigned 0 to 65535). Using the following technique, you can interpret results as hexadecimal values or convert them to real values (floating-point single precision).

```
1580 'Correct for BASICA's lack of unsigned
1590 MYVAL# = RESULT%(0)
1600 IF RESULT%(0) < 0 THEN MYVAL# = 65536! + MYVAL#
```

Note: After a timeout occurs, the counter resets to its initial value.

The following code segment interprets a time result returned by STCSTAT in BASICA.

```
450 ERRNUM% = 0      'error return variable
460 LASTCHR% = 0     'position of last character parsed by driver
470 '
480 MYVAL# = 0!      ' value returned from delay count register
490 SCALE% = 0       ' value from prescaler register
500 '
550 DIM RESULT%(2)   ' integer array to hold results from STCSTAT
560 '

1490 '
1500 '************** WAIT FOR DELAY *****************************
1510 '
1520 INIT$="wait;"
1530 CALL STCSTAT(INIT$, ERRNUM%, LASTCHR%, RESULT%(0))
1540 IF ERRNUM% THEN GOTO 2110
1550 PRINT"Waiting for time out"
1560 PRINT"Time Remaining = ";
1570 '
1580 'Correct for BASICA's lack of unsigned
1590 MYVAL# = RESULT%(0)
1600 IF RESULT%(0) < 0 THEN MYVAL# = 65536! + MYVAL#
1610 SCALE% = RESULT%(1) AND &HC0
1620 IF SCALE% = &HC0 THEN MYVAL# = 1000!*MYVAL#
1630 IF SCALE% = &H80 THEN MYVAL# = 100!*MYVAL#
1640 IF SCALE% = &H40 THEN MYVAL# = 10!*MYVAL#
1650 IF MYVAL# >= 1000! THEN GOTO 1680
1660 PRINT MYVAL#;" usec"
1670 GOTO 1740
1680 MYVAL# = MYVAL#/1000!
1690 IF MYVAL# >= 1000! THEN GOTO 1720
1700 PRINT MYVAL#;" msec"
1710 GOTO 1740
1720 MYVAL# = MYVAL#/1000!
1730 PRINT MYVAL#;" sec
1740 '
```

Interpreting Values in C

The following program segment illustrates the use of time values in the C programming language.

Note: After a timeout, the counter resets to its initial value.

```
unsigned ret_value[2];  // array to receive returned value
int  err     // int to receive error number
int  pos;    // int to receive exit position of parser

float time_remaining;  // time from delay count and prescaler

stcstat("wait;", &err, &pos,ret_value);
printf("Time remaining = ");
if (err != NO_ERROR) err_handler(err);
time_remaining = (float) ret_value[0];
switch (ret_value[1] & 0xc0){
         case 0xc0:{
            time_remaining *= 1000;
            break;
            }
         case 0x80:{
            time_remaining *= 100;
            break;
            }
         case 0x40:{
            time_remaining *= 10;
            break;
            }
         }
      if (time_remaining < 1000){
         printf("%.0f usecs\n",time_remaining);
         }
      else{
         time_remaining = time_remaining/1000.0;
         if (time_remaining < 1000){
            printf("%.4f msecs\n",time_remaining);
            }
         else{
            time_remaining = time_remaining/1000.0;
            printf("%.4f secs\n",time_remaining);
            }
         }   time_remaining = ret_value[0];
```

Interpreting Values in QuickBASIC and VisualBASIC

QuickBASIC and VisualBASIC handle 16-bit values (integers) as signed values in the range from -32768 through +32767 (Trigger Master returns values in the range unsigned 0 - 65535). Use the following technique to interpret results as hexadecimal values or convert them to real values.

```
'Correct for QuickBASIC's lack of unsigned
TimeRemaining = RetVal(0)
'
IF RetVal(0) < 0 THEN TimeRemaining = 65536! + TimeRemaining
```

Note: After a timeout, the counter resets to its initial value.

The following code segment interprets a time result returned by the STCSTAT in QuickBASIC.

```
DIM RetVal(2) AS INTEGER
DIM rerr   AS INTEGER
DIM post   AS INTEGER
DIM scale  AS INTEGER
TimeRemaining! = 0
CALL stcstat("wait;", rerr, post, RetVal(0))
PRINT "Waiting for time out"
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
'
'Correct for QuickBASIC's lack of unsigned
TimeRemaining = RetVal(0)
'
IF RetVal(0) < 0 THEN TimeRemaining = 65536! + TimeRemaining

PRINT "Time Remaining = ";
scale = RetVal(1) AND &HC0

SELECT CASE scale
     CASE &HC0
        TimeRemaining = 1000 * TimeRemaining
     CASE &H80
        TimeRemaining = 100 * TimeRemaining
     CASE &H40
        TimeRemaining = 10 * TimeRemaining
END SELECT

IF (TimeRemaining < 1000) THEN
   PRINT TimeRemaining; "usecs"
   ELSE
     TimeRemaining = TimeRemaining / 1000!
     IF (TimeRemaining < 1000) THEN
        PRINT TimeRemaining; "msecs"
        ELSE
          TimeRemaining = TimeRemaining / 1000!
          PRINT TimeRemaining; "secs"
      END IF
END IF
CALL stcstat("status;", rerr, post, RetVal(0))
PRINT "Checking status = "; RetVal(0)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

Interpreting Values in TurboPascal and TurboPascal for Windows

The following program segment illustrates the use of time values in TurboPascal and TurboPascal for Windows.

Note: After a timeout, the counter resets to its initial value.

```
stcstat('wait;', err, pos,RetValue[0]);
Write('Time remaining = ');
if (err <> NO_ERROR)
   then ErrHandler(err);
   TimeRemaining := RetValue[0];

case (RetValue[1] and $c0) of
     $c0: TimeRemaining := 1000*TimeRemaining;
     $80: TimeRemaining := 100*TimeRemaining;
     $40: TimeRemaining := 10*TimeRemaining;
end;

if (TimeRemaining < 1000)
     then Writeln(TimeRemaining, ' usecs')
     else
       begin
          TimeRemaining := TimeRemaining/1000.0;
          if (TimeRemaining < 1000)
          then Writeln(TimeRemaining,' msecs')
            else
            begin
               TimeRemaining := TimeRemaining/1000.0;
               Writeln(TimeRemaining,' secs');
            end;
        end;
```

Note: TurboPascal for Windows performs the WAIT command differently, but processes the time in the same manner.

The Request Set

The request set consists of the requests: ARM, CONT, FLAG, LOOP, STATUS, TRIG, and WAIT. The following sections discuss the requests in detail.

ARM

ARM returns information about the trigger input circuitry. The ARM forms are:

`ARM;` Returns the inverse of the Trigger Mask Register in the first element of the result array and the value FFF0(hex) in the second element of the array. The contents of the first element is in the following format:

7	6	5	4	3	2	1	0
X	X	CH6 MASK	CH5 MASK	CH4 MASK	CH3 MASK	CH2 MASK	CH1 MASK

A MASK bit value of 0 indicates that line is armed to receive a trigger.

`ARM:POL;` Returns contents of the Trigger Polarity Register in the first element of the result array and the value FFF1(hex) in the second element of the array. The contents of the first element is in the following format:

7	6	5	4	3	2	1	0
X	X	CH6 POL	CH5 POL	CH4 POL	CH3 POL	CH2 POL	CH1 POL

A POL (polarity) bit value of 0 arms the driver for a high-to-low transition and a value of 1 arms the driver for a low-to-high transition.

`ARM:REP;` Returns the number of trigger matches yet to be detected (Trigger Repeat Counter) in the first integer of the result array and the value FF00(hex) in the second integer of the array.

CONT

The CONT request returns the value of the Microprogram Counter in the first integer of the result array and the value FF01(hex) in the second integer of the array. The microprogram counter points to the next program step to execute. The syntax for the CONT request is as follows:

```
CONT;
```

This request is useful when loading multiple trigger programs. After loading the first program, the `CONT` request returns the address of the next available location within the sequence RAM. This address would then be used as an argument in the next `BEGIN` command.

FLAG

The FLAG request returns the value of the Diagnostic Flag Register in the first integer of the result array and the value FFF2(hex) in the second integer of the array. The syntax for the FLAG request is as follows:

```
FLAG;
```

LOOP

The LOOP request returns information about the progress of a program through Trigger Master program loops. The LOOP forms are:

LOOP; Returns the value of the Current Loop Counter in the first integer of the result array and the value FF02(hex) in the second integer of the array. The Current Loop Counter value is the number of times you must perform the loop after the current pass.

- When you enter a nested loop, the Current Loop Counter is stored and reloaded with the value appropriate to the new inner loop.
- When you leave the nested loop, the previously stored value will be returned to the Current Loop Counter.

LOOP:OUT; Returns the value the Current Loop Counter that was stored when a nested loop was entered in the first integer of the result array and the value FF03(hex) in the second integer of the array. The stored value will not be cleared when you leave the nested loop; use the FLAG commands at appropriate places to determine if the results of the LOOP:OUT request have any significance.

STATUS

The STATUS request returns the value of the Status Register in the first integer of the result array and the value FFF3(hex) in the second integer of the array. The contents of the Status Register are in the following format:

7	6	5	4	3	2	1	0
REG 2	REG 1	REG 0	TRIG DET	INT	INT EN	LOAD	RUN

The bits REG2, REG1, and REG0 determine which data register is accessed.

A TRIG DET bit value of 0 indicates you have issued an ARM command and a value of 1 means the conditions of your ARM command are met. (The TRIG DET bit is valid only when Trigger Master is not executing a program.) During program execution, this bit is also set, but the bit is automatically cleared on the next microsequencer clock cycle (therefore, you may never see this bit set). During program execution, use the ARM bit of the TRIG request to determine this same information.

An INT bit value of 1 indicates the board has generated an interrupt.

An INT-EN bit value of 1 indicates Trigger Master is set to generate interrupts.

A RUN bit value of 1 indicates a program is executing.

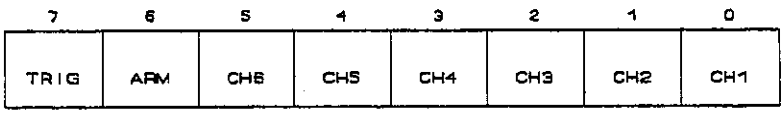
The syntax for the STATUS request is as follows:

```
STATUS;
```

TRIG

The TRIG request returns information about various registers. The TRIG forms take several extensions as follows:

`TRIG;` Returns the value of the Trigger Latch Register in the first integer of the result array and the value FFF4(hex) in the second integer of the array. The format for the Trigger Latch Register is:



The combined values of the TRIG bit and ARM bit are defined as:

TRIG	ARM	Definition
0	0	Inactive
0	1	Armed to Detect Trigger (program execution only)
1	0	Outputting Triggers
1	1	Semi-Sync Output (program execution only)

When executing a program, the ARM bit will be set while waiting for the conditions of your ARM command to be met. If you had programmed a SEMI trigger, the TRIG bit will also be set while awaiting the completion of the handshake.

The CH6-CH1 bit (channels 6-1) values indicate if trigger inputs have been latched. In immediate mode, the channel bits remain set until you issue an ARM command. When executing a program, the bits are cleared on the next microsequencer clock, therefore you may never detect them as set.

`TRIG:REP;` Returns the number of trigger matches yet to be detected or output (Trigger Repeat Counter) in the first integer of the result array and the value FF00(hex) in the second integer of the array.

`TRIG:PER;` Returns the value of the Delay Counter in the first integer of the result array and the Trigger Input/Prescaler register value in the second integer of the array. Refer to the *STCSTAT* subsection "Making Requests in the Programming Languages" for details on interpreting the results.

The Delay Counter resets to its initial value after counting down to 0. You can use a flag after a wait in a program to verify the end of a delay or use the TRIG request to verify the end of a trigger sequence.

`TRIG:IN;` Returns the Trigger Input/Prescaler Register in the first element of the result array and FFF5(hex) in the second element of the array. The format for the first element is as follows:

7	6	5	4	3	2	1	0
CLK1	CLK0	TRIG LINE 6	TRIG LINE 5	TRIG LINE 4	TRIG LINE 3	TRIG LINE 2	TRIG LINE 1

Note: The actual state of the trigger lines can be determined for diagnostic purposes, however trigger detection is based on latched transitions.

WAIT

The WAIT request returns the value of the Delay Counter in the first integer and the Trigger Input/Prescaler register value in the second integer. Refer to the section "Values Returned by STCSTAT" for a discussion on interpreting the results.

The Delay Counter resets to its initial value after counting down to 0. You can use a flag after a wait in a program to verify the end of a delay. The syntax for the WAIT request is as follows:

```
WAIT;
```

3.7 STCLOAD

Use STCLOAD to load a binary file into Trigger Master program memory. The binary file can contain up to 1024 bytes and will normally be generated by STCCOM.EXE (refer to either Chapter 4 or the following STCDUMP command description). STCLOAD accepts two arguments: the first argument is the file name to be loaded, and the second argument is the variable to receive the status returned by the driver at the completion of the command.

Calling STCLOAD from BASICA

The following program segment illustrates calling STCLOAD from BASICA.

```
930 FILENAME$ = "exam.dat"

1080 '
1090 PRINT "Load file "; FILENAME$; " to Trigger Master #"; BRDNUM%
1100 CALL STCLOAD(FILENAME$, ERRNUM%)
1110 IF ERRNUM% THEN GOTO 2130
1120 '
```

Calling STCLOAD from C

The following program segment illustrates calling STCLOAD from the C programming language.

```
char  file_name[] = {"exam.dat"};

printf("Load file %s to Trigger Master # 1\n",file_name);
stcload(file_name, &err);
if (err != NO_ERROR) err_handler(err);
```

Calling STCLOAD from QuickBASIC and VisualBASIC

The following program segment illustrates calling STCLOAD from QuickBASIC and VisualBASIC.

```
filename$ = "examp.dat"

PRINT "Load file "; filename$; " to Trigger Master #"; BrdNum
CALL stcload(filename$, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

Calling STCLOAD from TurboPascal and TurboPascal for Windows

The following program segment illustrates calling STCLOAD from TurboPascal.

```
const
FileName = 'exam.dat';
var
err: ErrorCodes;

stcload(FileName,err);
```

STCTP.LIB is a special interface to TurboPascal that accepts a standard TurboPascal string. *STCLIB.DLL* is a general DLL that expects so-called "C" strings. To send strings to the DLL in TurboPascal for Windows, create a string with a NULL character on the end outside the call, and pass the "second element" of the string in the call.

```
const
FileName = 'exam.dat' + #0;
var
err: ErrorCodes;

stcload(FileName[1],err);
```

3.8 STCDUMP

Use STCDUMP to store the contents of Trigger Master program memory into a file. You can then use the STCLOAD command (previously described) to load the program back into Trigger Master program memory. The STCDUMP command accepts two arguments: the first argument specifies the file name and the second argument is a variable that receives the status returned by the driver at the completion of the command. The file will contain 1024 bytes in binary format.

Calling STCDUMP from BASICA

The following program segment illustrates calling STCDUMP from BASICA.

```
930 FILENAME$ = "exam.dat"
940 PRINT "Save program in Trigger Master #"; BRDNUM%; " to file "; FILENAME$
950 CALL STCDUMP(FILENAME$, ERRNUM%)
960 IF ERRNUM% THEN GOTO 2130
```

Calling STCDUMP from C

The following program segment illustrates calling STCDUMP from the C programming language.

```
char  file_name[] = {"exam.dat"};

printf("Save program in Trigger Master #0 to file %s\n",file_name);
stcdump(file_name, &err);
if (err != NO_ERROR) err_handler(err);
```

Calling STCDUMP from QuickBASIC and VisualBASIC

The following program segment illustrates calling STCDUMP from QuickBASIC and VisualBASIC.

```
filename$ = "examp.dat"

PRINT "Save program in Trigger Master #"; BrdNum; " to file "; filename$
CALL stcdump(filename$, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

Calling STCDUMP from TurboPascal and TurboPascal for Windows

The following program segment illustrates calling STCDUMP from TurboPascal and TurboPascal for Windows.

```
const
FileName = 'exam.dat';
var
err: ErrorCodes;

stcdump(FileName,err);
```

STCTP.LIB is a special interface to TurboPascal that accepts a standard TurboPascal string. *STCLIB.DLL* is a general DLL that expects so-called "C" strings. To send strings to the DLL in TurboPascal for Windows, create a string with a NULL character on the end outside the call, and pass the "second element" of the string in the call.

```
const
FileName = 'exam.dat' + #0;
var
err: ErrorCodes;

stcdump(FileName[1],err);
```

Chapter 4
PROGRAMMING EXAMPLES

4.1 INTRODUCTION

This chapter presents programming examples for each of these languages supported by Trigger Master: BASICA, C, QuickBasic, and TurboPascal.

Many of the programming examples check the status argument after each call; once the program has been debugged, it may only be necessary to verify that the board is available (following the STCINIT command). Both the STCCMD and STCSTAT command set an integer to the position of the last character in the string that was parsed by the driver. If a nonzero error is returned, check the position variable to see where the error was encountered.

Most of the examples use a separate STCCMD call that contains a single command. Using this method allows easier commenting of the code and assists in the debugging process. You may include a single string with multiple commands in the STCCMD call if you separate each command with a semicolon.

Notes: The string length limits are 255 characters for BASICA and 256 characters for TurboPascal.

Trigger Master executes commands as they are parsed. If Trigger Master encounters an error in a multiple-command string, it executes all valid commands prior to the command in error.

In an STCSTAT call, Trigger Master returns after the first semicolon (Trigger Master can only process one request per request string).

4.2 BASICA LANGUAGE EXAMPLE

```
10  'EXAMPGW.BAS",a
20 CLS: KEY OFF
30  '*************************************************************
40  '
50  ' EXAMPGW.BAS
60  '
70  ' Uses two STC boards to demonstrate interfacing the driver
80  ' GW Basic.
90  '
100 ' to wait for 500 trigger inputs on line 1 and then generate
110 ' a delay of .2 s.  The program is dumped to a file.
120 '
130 ' The second STC is initialized as #1.  The program is loaded
140 ' and started to execute.
150 '
160 ' Control is shifted back to STC #0 which is told to generate
170 ' 500 triggers on line 1.
180 '
190 ' Finally board 1 is monitored until it finishes.
200 '
210 '*************************************************************
220 '
230 ' Install BASICA Error Handler
240 ON ERROR GOTO 2200
250 '
260  CLEAR , 62*1024                    ' leave 2048 for interface
270  DEF SEG = 0
280  SG = 256 * PEEK(&H511) + PEEK(&H510)
290  SG = SG + &H1100
300  DEF SEG =SG              ' DEF SEG must be set = SG in order
310                           ' to call into the driver
320  BLOAD "STCBAS.BIN", 0
330 ' The following integer variables are set to the offset
340                                    ' values required
350 STCINIT     = 0   ' Initialize STC
360 STCSET      = 3   ' Switch to already initialized STC
370 STCCMD      = 6   ' Send command to STC
380 STCSTAT     = 9   ' Request information from STC
390 STCLOAD     = 12  ' Load a program in STC memory for file
400 STCDUMP     = 15  ' Save STC memory to a file
410 '
420 'must initialize strings before call (also sets count)
430 INIT$ = STRING$(80,32)   'string to send commands and requests
440 '
450 ERRNUM% = 0       'error return variable
460 LASTCHR% = 0      'position of last character parsed by driver
470 '
480 MYVAL# = 0! ' value returned from delay count register
490 SCALE% = 0  ' value from prescaler register
500 '
510 BASSEG%=-1  ' flag for buffer transfers into BASIC's
520               ' data segment.
530 '
540 '
550 DIM RESULT%(2) ' integer array to hold results from STCSTAT
```

```
560 '
570 PRINT
580 '
590 '*********************  INITIALIZE STC 0 ******************
600 '
610 BRDNUM% = 0
620 BRDADDR% = &H310
630 PRINT "Initialize board "; BRDNUM%; " at address ";
640 PRINT HEX$(BRDADDR%); " hex"
650 CALL STCINIT(BRDNUM%, BRDADDR%, ERRNUM%)
660 IF ERRNUM% THEN GOTO 2110
670 '
680 '****************  PROGRAM STC 0 MEMORY  ********************
690 '
700 INIT$="begin;"                      ' begin program
710 PRINT "Send "; INIT$; " command"
720 CALL STCCMD(INIT$, ERRNUM%, LASTCHR%)
730 IF ERRNUM% THEN GOTO 2110
740 '
750 INIT$="arm1:rep500;"                ' wait for 500 triggers on
760                                     ' line 1
770 PRINT "Send "; INIT$; " command"
780 CALL STCCMD(INIT$, ERRNUM%, LASTCHR%)
790 IF ERRNUM% THEN GOTO 2110
800 '
810 INIT$="wait .2s;"                   ' wait a tenth of a second
820 PRINT "Send "; INIT$; " command"
830 CALL STCCMD(INIT$, ERRNUM%, LASTCHR%)
840 IF ERRNUM% THEN GOTO 2110
850 '
860 INIT$="end;"                         ' end program
870 PRINT "Send "; INIT$; " command"
880 CALL STCCMD(INIT$, ERRNUM%, LASTCHR%)
890 IF ERRNUM% THEN GOTO 2110
900 '
910 '***************** SAVE PROGRAM TO MEMORY ********************
920 '
930 FILENAME$ = "exam.dat"
940 PRINT "Save program in STC #"; BRDNUM%; " to file ";FILENAME$
950 CALL STCDUMP(FILENAME$, ERRNUM%)
960 IF ERRNUM% THEN GOTO 2110
970 '
```

```
980 '***************** INITIALIZE OTHER STC *********************
990 '
1000 BRDNUM% = 1
1010 BRDADDR% = &H314
1020 PRINT "Initialize board "; BRDNUM%; " at address ";
1030 PRINT HEX$(BRDADDR%); " hex"
1040 CALL STCINIT(BRDNUM%, BRDADDR%, ERRNUM%)
1050 IF ERRNUM% THEN GOTO 2110
1060 '
1070 '***************** LOAD PROGRAM TO MEMORY *******************
1080 '
1090 PRINT "Load file "; FILENAME$; " to STC #"; BRDNUM%
1100 CALL STCLOAD(FILENAME$, ERRNUM%)
1110 IF ERRNUM% THEN GOTO 2110
1120 '
1130 '***************** START PROGRAM EXECUTING IN 1 *************
1140 '
1150 INIT$="x;"                          ' end program
1160 PRINT "Send "; INIT$; " command"
1170 CALL STCCMD(INIT$, ERRNUM%, LASTCHR%)
1180 IF ERRNUM% THEN GOTO 2110
1190 '
1200 '************** RETURN TO BRD 0 AND SEND TRIGGERS ***********
1210 '
1220 BRDNUM% = 0
1230 PRINT "Switch driver control back to STC #"; BRDNUM%
1240 CALL STCSET(BRDNUM%, ERRNUM%)
1250 IF ERRNUM% THEN GOTO 2110
1260 '
1270 INIT$="trig1:rep500:per .3m;"   ' generate 500 trigger
1280                                 ' 0.3 millisecs apart
1290 PRINT "Send "; INIT$; " command"
1300 CALL STCCMD(INIT$, ERRNUM%, LASTCHR%)
1310 IF ERRNUM% THEN GOTO 2110
1320 '
1330 '********** GO BACK TO BRD 1 AND WAIT FOR TRIGGERS **********
1340 '
1350 BRDNUM% = 1
1360 PRINT "Switch driver control back to STC #"; BRDNUM%
1370 CALL STCSET(BRDNUM%, ERRNUM%)
1380 IF ERRNUM% THEN GOTO 2110
1390 '
1400 INIT$="trig:rep;"
1410 CALL STCSTAT(INIT$, ERRNUM%, LASTCHR%, RESULT%(0))
1420 IF ERRNUM% THEN GOTO 2110
1430 PRINT"Waiting for triggers; REP = ";RESULT%(0)
1440 IF RESULT%(0) > 0 GOTO 1410
1450 '
1460 PRINT
1470 PRINT "NOTE: Counts = 0 at completion"
1480 PRINT
1490 '
1500 '************** WAIT FOR DELAY ******************************
1510 '
1520 INIT$="wait;"
1530 CALL STCSTAT(INIT$, ERRNUM%, LASTCHR%, RESULT%(0))
1540 IF ERRNUM% THEN GOTO 2110
```

```
1550 PRINT"Waiting for time out"
1560 PRINT"Time Remaining = ";
1570 '
1580 'Correct for BASICA's lack of unsigned
1590 MYVAL# = RESULT%(0)
1600 IF RESULT%(0) < 0 THEN MYVAL# = 65536! + MYVAL#
1610 SCALE% = RESULT%(1) AND &HC0
1620 IF SCALE% = &HC0 THEN MYVAL# = 1000!*MYVAL#
1630 IF SCALE% = &H80 THEN MYVAL# = 100!*MYVAL#
1640 IF SCALE% = &H40 THEN MYVAL# = 10!*MYVAL#
1650 IF MYVAL# >= 1000! THEN GOTO 1680
1660 PRINT MYVAL#;" usec"
1670 GOTO 1740
1680 MYVAL# = MYVAL#/1000!
1690 IF MYVAL# >= 1000! THEN GOTO 1720
1700 PRINT MYVAL#;" msec"
1710 GOTO 1740
1720 MYVAL# = MYVAL#/1000!
1730 PRINT MYVAL#;" sec
1740 '
1750 INIT$="status;"
1760 CALL STCSTAT(INIT$, ERRNUM%, LASTCHR%, RESULT%(0))
1770 IF ERRNUM% THEN GOTO 2110
1780 PRINT "Checking status = "; RESULT%(0)
1790 IF (&H1 AND RESULT%(0))>0 GOTO 1520
1800 '
1810 PRINT
1820 PRINT "Timed out."
1830 PRINT
1840 '
1850 INIT$="wait;"
1860 CALL STCSTAT(INIT$, ERRNUM%, LASTCHR%, RESULT%(0))
1870 IF ERRNUM% THEN GOTO 2110
1880 PRINT"Time Read back = ";
1890 '
1900 'Correct for BASICA's lack of unsigned
1910 MYVAL# = RESULT%(0)
1920 IF RESULT%(0) < 0 THEN MYVAL# = 65536! + MYVAL#
1930 SCALE% = RESULT%(1) AND &HC0
1940 IF SCALE% = &HC0 THEN MYVAL# = 1000!*MYVAL#
1950 IF SCALE% = &H80 THEN MYVAL# = 100!*MYVAL#
1960 IF SCALE% = &H40 THEN MYVAL# = 10!*MYVAL#
1970 IF MYVAL# >= 1000! THEN GOTO 2000
1980 PRINT MYVAL#;" usec"
1990 GOTO 2060
2000 MYVAL# = MYVAL#/1000!
2010 IF MYVAL# >= 1000! THEN GOTO 2040
2020 PRINT MYVAL#;" msec"
2030 GOTO 2060
2040 MYVAL# = MYVAL#/1000!
2050 PRINT MYVAL#;" sec
2060 '
2070 PRINT
2080 PRINT"NOTE: TIME = ORIGINAL DELAY AT COMPLETION."
2090 END
2100 '
2110 '************************  ERROR HANDLER  ******************
2120 '
```

```
2130 PRINT "Driver returned ";ERRNUM%
2140 '
2150 '
2160 '
2170 PRINT
2180 PRINT
2190 STOP
2200 '
2210 '*********************  BASICA ERROR HANDLER  ***************
2220 '
2230 PRINT "IEEE Error In Line ";ERL; "Error Number = ";ERR
2240 STOP
```

4.3 C LANGUAGE EXAMPLE

```
//*****************************************************************
//
// exampc.c
//
// Uses two STC boards to demonstrate interfacing the driver in C
//
//   The first STC is initialized as #0 and loaded with a program
//   to wait for 500 trigger inputs on line 1 and then generate
//   a delay of .2 s.  The program is dumped to a file.
//
//   The second STC is initialized as #1.  The program is loaded
//   and started to execute.
//
//   Control is shifted back to STC #0 which is told to generate
//   500 triggers on line 1.
//
//   Finally control is returned to #1 where the progress of the
//   program is monitored.
//
//*****************************************************************

#include <stdio.h>
#include <string.h>
#include <graph.h>
#include <stdlib.h>
#include "stc.h"
//
// value returned by stcstat
//

unsigned int ret_reg[2], ret_value[2];
               // status values returned
               // if [0] is 0xfff0 -> [1] is ARM MASK
               // if [0] is 0xfff1 -> [1] is POLARITY MASK
               // if [0] is 0xfff2 -> [1] is FLAG REG
               // if [0] is 0xfff3 -> [1] is STATUS
               // if [0] is 0xfff4 -> [1] is TRIGGER LATCH
               // if [0] is 0xfff5 -> [1] is TRIG IN/PRE
               // if [0] is 0xff00 -> [1] is REP COUNTER
               // if [0] is 0xff01 -> [1] is PROG COUNTER
               // if [0] is 0xff02 -> [1] is LOOP COUNTER
               // if [0] is 0xff03 -> [1] is OUTER LOOP COUNT
               // otherwise [1] is prescaler and
               // [0] is unsigned time

char  file_name[ ] = {"exam.dat"};

char  begin_cmd[ ] = {"begin;"};              // begin program
char  arm_cmd[ ]   = {"arm1:rep500;"};        // wait for 500 triggers on line 1
char  wait_cmd[ ]  = {"wait .2s;"};           // wait a tenth of a second
char  end_cmd[ ]   = {"end;"};                // end program

char  x_cmd[ ]     = {"x;"};          // start execution

char  trig_cmd[ ]  = {"trig1:rep500:per .3m;"};
                                              // generate 500 triggers .3 millisecs apart
```

```
//
// function prototype for error handler
//

void err_handler(int err_num);
void main()
{
   int   err;        // int to receive error number
   int   pos;        // int to receive exit position of parser


   float time_remaining;   // time from delay count and prescaler

   _clearscreen(_GCLEARSCREEN); // clear entire screen

   printf("EXAMPC.C\n\n");

   printf("Initialize board 0 at address 310 hex\n");
   stcinit(0,0x310,&err);
   if (err != NO_ERROR) err_handler(err);

   printf("Send %s command\n",begin_cmd);
   stccmd(begin_cmd, &err, &pos);
   if (err != NO_ERROR) err_handler(err);

   printf("Send %s command\n",arm_cmd);
   stccmd(arm_cmd, &err, &pos);
   if (err != NO_ERROR) err_handler(err);

   printf("Send %s command\n",wait_cmd);
   stccmd(wait_cmd, &err, &pos);
   if (err != NO_ERROR) err_handler(err);

   printf("Send %s command\n",end_cmd);
   stccmd(end_cmd, &err, &pos);
   if (err != NO_ERROR) err_handler(err);

   printf("Save program in STC #0 to file %s\n",file_name);
   stcdump(file_name, &err);
   if (err != NO_ERROR) err_handler(err);

   printf("Initialize board 1 at address 314 hex\n");
   stcinit(1,0x314,&err);
   if (err != NO_ERROR) err_handler(err);
   printf("Load file %s to STC # 1\n",file_name);
   stcload(file_name, &err);
   if (err != NO_ERROR) err_handler(err);

   printf("Send %s command to start execution ",x_cmd);
   printf("of STC #1\n");
   stccmd(x_cmd, &err, &pos);
   if (err != NO_ERROR) err_handler(err);

   printf("Switch driver control back to STC #0\n");
   stcset(0,&err);
   if (err != NO_ERROR) err_handler(err);

   printf("Send %s command to send triggers ",trig_cmd);
   printf("from STC #0 to STC #1\n");
   stccmd(trig_cmd, &err, &pos);
   if (err != NO_ERROR) err_handler(err);
```

```
printf("Switch driver control to STC #1\n");
stcset(1,&err);
if (err != NO_ERROR) err_handler(err);

do
   {
   stcstat("trig:rep;", &err, &pos,ret_value);
   printf("Waiting for triggers; REP = %d\n",ret_value[0]);
   if (err != NO_ERROR) err_handler(err);
   }
while (ret_value[0]>0); // wait for trigger inputs

do
   {
   stcstat("wait;", &err, &pos,ret_value);
   printf("Time remaining = ");
   if (err != NO_ERROR) err_handler(err);
   time_remaining = (float) ret_value[0];
   switch (ret_value[1] & 0xc0){
      case 0xc0:{
         time_remaining *= 1000;
         break;
         }
      case 0x80:{
         time_remaining *= 100;
         break;
         }
      case 0x40:{
         time_remaining *= 10;
         break;
         }
      }
   if (time_remaining < 1000){
      printf("%.0f usecs\n",time_remaining);
      }
   else{
      time_remaining = time_remaining/1000.0;
      if (time_remaining < 1000){
         printf("%.4f msecs\n",time_remaining);
         }
      else{
         time_remaining = time_remaining/1000.0;
         printf("%.4f secs\n",time_remaining);
         }
      }
   stcstat("status;", &err, &pos,ret_reg);
   printf("Checking status = %x hex\n",ret_reg[0]);
   if (err != NO_ERROR) err_handler(err);
   }
while (0x01 & ret_reg[0]); // wait while program is executing
```

```
   stcstat("wait;", &err, &pos,ret_value);
   printf("\nTimed out. Wait returns ");
   if (err != NO_ERROR) err_handler(err);
   time_remaining = (float) ret_value[0];
   switch (ret_value[1] & 0xc0){
      case 0xc0:{
         time_remaining *= 1000;
         break;
         }
      case 0x80:{
         time_remaining *= 100;
         break;
         }
      case 0x40:{
         time_remaining *= 10;
         break;
         }
      }
   if (time_remaining < 1000){
      printf("%.0f usecs\n",time_remaining);
      }
   else{
      time_remaining = time_remaining/1000.0;
      if (time_remaining < 1000){
         printf("%.4f msecs\n",time_remaining);
         }
      else{
         time_remaining = time_remaining/1000.0;
         printf("%.4f secs\n",time_remaining);
         }
      }

   printf("\nNOTE: At completion WAIT; returns the original delay.\n");

}  // End of main

void err_handler(int err_num)
{
   printf("error = %s\n",error_msg[err_num]);
   exit(0);
}
```

4.4 QuickBASIC EXAMPLE

```
'*********************************************************************
'
' EXAMPQB.BAS
'
' Uses two STC boards to demonstrate interfacing the driver in
'    QuickBASIC for versions above QuickBASIC versions 4.0 and
'    greater.
'
'    The first STC is initialized as #0 and loaded with a program
'    to wait for 500 trigger inputs on line 1 and then generate
'    a delay of .2 s.  The program is dumped to a file.
'
'    The second STC is initialized as #1.  The program is loaded
'    and started to execute.
'
'    Control is shifted back to STC #0 which is told to generate
'    500 triggers on line 1.
'
'    Finally control is returned to #1 where the progress of the
'    program is monitored.
'
' INCLUDE FILE : STCQB.BI
'
' QuickBASIC ENVIRONMENT :
'      Use DOS command QBx /L STCQBx [filename.bas]
'          Where x will be 4 or 7 depending on whether your
'          version is 7 or less.
'
'*********************************************************************

DECLARE SUB ErrorExit (rerr%)

'$INCLUDE: 'stcqb.bi'

' values returned by stcstat

  DIM RetVal(2) AS INTEGER
                ' status values returned
                ' if [0] is 0xfff0 -> [1] is ARM MASK
                ' if [0] is 0xfff1 -> [1] is POLARITY MASK
                ' if [0] is 0xfff2 -> [1] is FLAG REG
                ' if [0] is 0xfff3 -> [1] is STATUS
                ' if [0] is 0xfff4 -> [1] is TRIGGER LATCH
                ' if [0] is 0xfff5 -> [1] is TRIG IN/PRE
                ' if [0] is 0xff00 -> [1] is REP COUNTER
                ' if [0] is 0xff01 -> [1] is PROG COUNTER
                ' if [0] is 0xff02 -> [1] is LOOP COUNTER
                ' if [0] is 0xff03 -> [1] is OUTER LOOP COUNT
                ' otherwise [1] is prescaler and
                ' [0] is unsigned time

   filename$ = "examp.dat"

   begincmd$ = "begin;"        ' begin program
   armcmd$ = "arm1:rep500;"    ' wait for 500 triggers on line 1
   waitcmd$ = "wait .2s;"      ' wait a tenth of a second
   endcmd$ = "end;"            ' end program
   xcmd$ = "x;"                ' start execution
   trigcmd$ = "trig1:rep500:per .3m;"   ' generate 500 trigger .3 millisecs apart
```

```
DIM rerr     AS INTEGER
DIM post     AS INTEGER
DIM BrdNum   AS INTEGER
DIM brdAddr AS INTEGER
DIM scale    AS INTEGER

TimeRemaining! = 0

CLS                    ' clear entire screen

PRINT "EXAMPQB.BAS"
PRINT

BrdNum = 0
brdAddr = &H310
PRINT "Initialize board "; BrdNum; " at address ";
PRINT HEX$(brdAddr); " hex"
CALL stcinit(BrdNum, brdAddr, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Send "; begincmd$; " command"
CALL stccmd(begincmd$, rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Send "; armcmd$; " command"
CALL stccmd(armcmd$, rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Send "; waitcmd$; " command"
CALL stccmd(waitcmd$, rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Send "; endcmd$; " command"
CALL stccmd(endcmd$, rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Save program in STC #"; BrdNum; " to file "; filename$
CALL stcdump(filename$, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT
BrdNum = 1
brdAddr = &H314
PRINT "Initialize board "; BrdNum; " at address ";
PRINT HEX$(brdAddr); " hex"
CALL stcinit(BrdNum, brdAddr, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Load file "; filename$; " to STC #"; BrdNum
CALL stcload(filename$, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Send "; xcmd$; " command to start execution "
PRINT "of STC #"; BrdNum
CALL stccmd(xcmd$, rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
```

```
PRINT
BrdNum = 0
PRINT "Switch driver control back to STC #"; BrdNum
CALL stcset(BrdNum, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT "Send "; trigcmd$; " command to send triggers "
PRINT "from the first STC to the second"
CALL stccmd(trigcmd$, rerr, post)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

PRINT
BrdNum = 1
PRINT "Switch driver control to STC #"; BrdNum
CALL stcset(BrdNum, rerr)
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)

DO
   CALL stcstat("trig:rep;", rerr, post, RetVal(0))
   PRINT "Waiting for triggers; REP = "; RetVal(0)
   IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
LOOP WHILE (RetVal(0) > 0)

PRINT
PRINT "NOTE: Counts = 0 at completion."
PRINT

DO
   CALL stcstat("wait;", rerr, post, RetVal(0))
   PRINT "Waiting for time out"
   IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
   '
   'Correct for QuickBASIC's lack of unsigned
   TimeRemaining = RetVal(0)
   '
   IF RetVal(0) < 0 THEN TimeRemaining = 65536! + TimeRemaining
   PRINT "Time Remaining = ";
   scale = RetVal(1) AND &HC0
   SELECT CASE scale
      CASE &HC0
         TimeRemaining = 1000 * TimeRemaining
      CASE &H80
         TimeRemaining = 100 * TimeRemaining
      CASE &H40
         TimeRemaining = 10 * TimeRemaining
   END SELECT

   IF (TimeRemaining < 1000) THEN
      PRINT TimeRemaining; "usecs"
   ELSE
      TimeRemaining = TimeRemaining / 1000!
      IF (TimeRemaining < 1000) THEN
         PRINT TimeRemaining; "msecs"
      ELSE
         TimeRemaining = TimeRemaining / 1000!
         PRINT TimeRemaining; "secs"
      END IF
   END IF
   CALL stcstat("status;", rerr, post, RetVal(0))
   PRINT "Checking status = "; RetVal(0)
   IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
LOOP WHILE (&H1 AND RetVal(0))

PRINT
PRINT "Timed out."
PRINT
CALL stcstat("wait;", rerr, post, RetVal(0))
IF rerr <> NOERROR THEN CALL ErrorExit(rerr)
'
'Correct for QuickBASIC's lack of unsigned
TimeRemaining = RetVal(0)
'
```

```
IF RetVal(0) < 0 THEN TimeRemaining = 65536! + TimeRemaining

PRINT "Time Remaining = ";
scale = RetVal(1) AND &HC0
SELECT CASE scale
   CASE &HC0
      TimeRemaining = 1000 * TimeRemaining
   CASE &H80
      TimeRemaining = 100 * TimeRemaining
   CASE &H40
      TimeRemaining = 10 * TimeRemaining
END SELECT

IF (TimeRemaining < 1000) THEN
   PRINT TimeRemaining; "usecs"
ELSE
   TimeRemaining = TimeRemaining / 1000!
   IF (TimeRemaining < 1000) THEN
      PRINT TimeRemaining; "msecs"
   ELSE
      TimeRemaining = TimeRemaining / 1000!
      PRINT TimeRemaining; "secs"
   END IF
END IF

PRINT
PRINT "NOTE: Time = original delay at completion."

END

SUB ErrorExit (rerr%)
   PRINT "error = "; ErrorMsg$(rerr%)
   END
END SUB
```

4.5 TurboPascal EXAMPLE

```
{*****************************************************************

examtp.pas

Uses two STC boards to demonstrate interfacing the driver in
TurboPascal.

    The first STC is initialized as #0 and loaded with a program
    to wait for 500 trigger inputs on line 1 and then generate
    a delay of .2 s.  The program is dumped to a file.

    The second STC is initialized as #1.  The program is loaded
    and started to execute.

    Control is shifted back to STC #0 which is told to generate
    500 triggers on line 1.

    Finally control is returned to #1 where the progress of the
    program is monitored.

*****************************************************************
}

program examptp;

{$I stctp.inc} { Include function prototypes for functions in sctp.obj }
{$L stctp.obj}  { Link with stctp.obj }

const

  FileName = 'exam.dat';
  BeginCmd = 'begin;';                  { begin program}
  ArmCmd   = 'arm1:rep500;';            { wait for 500 triggers on line 1 }
  WaitCmd  = 'wait .2s;';               { wait a tenth of a second}
  EndCmd   = 'end;';                    { end program}
  XCmd     = 'x;';                      { start execution }
  TrigCmd  = 'trig1:rep500:per .3m;'; { generate 500 triggers, .3 millisecs apart}

var
{ value returned by stcstat }

  RetReg:      array [0..1] of WORD;
  RetValue:    array [0..1] of WORD;

              { status values returned
                if [0] is 0xfff0 -> [1] is ARM MASK
                if [0] is 0xfff1 -> [1] is POLARITY MASK
                if [0] is 0xfff2 -> [1] is FLAG REG
                if [0] is 0xfff3 -> [1] is STATUS
                if [0] is 0xfff4 -> [1] is TRIGGER LATCH
                if [0] is 0xfff5 -> [1] is TRIG IN/PRE
                if [0] is 0xff00 -> [1] is REP COUNTER
                if [0] is 0xff01 -> [1] is PROG COUNTER
                if [0] is 0xff02 -> [1] is LOOP COUNTER
                if [0] is 0xff03 -> [1] is OUTER LOOP COUNT
                otherwise [1] is prescaler and [0] is unsigned time }

   err:                 ErrorCodes;   { integer to receive error number }
   pos:                 Word; { integer to receive exit position of parser }
   TimeRemaining:       Real; { time from delay count and prescaler }
   ErrorMsg: array[ErrorCodes] of string [26] ;

procedure ErrHandler(ErrNum: ErrorCodes);

begin
   Writeln('error = ', ErrorMsg[ErrNum]);
   Halt
end;
begin
```

```
(*   _clearscreen(_GCLEARSCREEN); // clear entire screen*)
ErrorMsg[NO_ERROR                        ] := 'NO ERROR';
ErrorMsg[UNRECOGNIZED_COMMAND            ] := 'UNRECOGNIZED COMMAND';
ErrorMsg[INCOMPLETE_COMMAND              ] := 'INCOMPLETE COMMAND';
ErrorMsg[ARM_NEEDS_LINE                  ] := 'ARM NEEDS LINE';
ErrorMsg[NEED_SEMI_COLON                 ] := 'NEED SEMI COLON';
ErrorMsg[OUT_OF_CHARS                    ] := 'OUT OF CHARS';
ErrorMsg[NO_COMMAND                      ] := 'NO COMMAND';
ErrorMsg[NEED_ANOTHER_LINE               ] := 'NEED ANOTHER LINE';
ErrorMsg[ILLEGAL_EXTEN                   ] := 'ILLEGAL EXTEN';
ErrorMsg[REP_OVER_RNG                    ] := 'REP OVER RNG';
ErrorMsg[NOT_IN_IMMED_MODE               ] := 'NOT IN IMMED MODE';
ErrorMsg[ADD_OVER_RNG                    ] := 'ADD OVER RNG';
ErrorMsg[NOT_IN_PROG_MODE                ] := 'NOT IN PROG MODE';
ErrorMsg[DO_NEEDS_VALUE                  ] := 'DO NEEDS VALUE';
ErrorMsg[DO_OVER_RANGE                   ] := 'DO OVER RANGE';
ErrorMsg[EXCEEDS_DO_LEVEL                ] := 'EXCEEDS DO LEVEL';
ErrorMsg[FLAG_OVER_RANGE                 ] := 'FLAG OVER RANGE';
ErrorMsg[NOT_IN_LOOP                     ] := 'NOT IN LOOP';
ErrorMsg[WAIT_NEEDS_VALUE                ] := 'WAIT NEEDS VALUE';
ErrorMsg[X_OVER_RANGE                    ] := 'X OVER RANGE';
ErrorMsg[TIME_OVER_RESOLUTION            ] := 'TIME OVER RESOLUTION';
ErrorMsg[TIME_OVER_RANGE                 ] := 'TIME OVER RANGE';
ErrorMsg[NEED_TIME_SCALE                 ] := 'NEED TIME SCALE';
ErrorMsg[NEED_TIME_VALUE                 ] := 'NEED TIME VALUE';
ErrorMsg[SECS_OVER_RNG                   ] := 'SECS OVER RNG';
ErrorMsg[MSECS_OVER_RNG                  ] := 'MSECS OVER RNG';
ErrorMsg[USECS_OVER_RNG                  ] := 'USECS OVER RNG';
ErrorMsg[SECS_UNDER_RNG                  ] := 'SECS UNDER RNG';
ErrorMsg[MSECS_UNDER_RNG                 ] := 'MSECS UNDER RNG';
ErrorMsg[USECS_UNDER_RNG                 ] := 'USECS UNDER RNG';
ErrorMsg[TRIG_NEEDS_LINE                 ] := 'TRIG NEEDS LINE';
ErrorMsg[NEED_EXTENSION                  ] := 'NEED EXTENSION';
ErrorMsg[PER_REQUIRES_REP                ] := 'PER REQUIRES REP';
ErrorMsg[REP_NEEDS_PER_OR_SEMI           ] := 'REP NEEDS PER OR SEMI';
ErrorMsg[STC_PREVIOUSLY_INITIALIZED]       := 'STC PREVIOUSLY INITIALIZED';
ErrorMsg[STC_NOT_PRESENT                 ] := 'STC NOT PRESENT';
ErrorMsg[STC_ALREADY_ACTIVE              ] := 'STC ALREADY ACTIVE';
ErrorMsg[STC_NOT_INITIALIZED             ] := 'STC NOT INITIALIZED';
ErrorMsg[STC_NUM_OUT_OF_RNG              ] := 'STC NUM OUT OF RNG';
ErrorMsg[DUPLICATE_STC_ADDR              ] := 'DUPLICATE STC ADDR';
ErrorMsg[UNRESOLVED_LOOP                 ] := 'UNRESOLVED LOOP';
ErrorMsg[FILE_NAME_TOO_LONG              ] := 'FILE NAME TOO LONG';
ErrorMsg[PROB_READING_FILE               ] := 'PROB READING FILE';
ErrorMsg[PROB_OPENING_READ_FILE          ] := 'PROB OPENING READ FILE';
ErrorMsg[PROB_WRITING_FILE               ] := 'PROB WRITING FILE';
ErrorMsg[PROB_OPENING_WRT_FILE           ] := 'PROB OPENING WRT FILE';
ErrorMsg[FILE_NOT_FOUND                  ] := 'FILE NOT FOUND';
ErrorMsg[PATH_NOT_FOUND                  ] := 'PATH NOT FOUND';
ErrorMsg[TOO_MANY_OPEN_FILES             ] := 'TOO MANY OPEN FILES';
ErrorMsg[ACCESS_DENIED                   ] := 'ACCESS DENIED';
ErrorMsg[INVALID_ACCESS_CODE             ] := 'INVALID ACCESS CODE';
ErrorMsg[UNRECOGNIZED_REQUEST            ] := 'UNRECOGNIZED REQUEST';
ErrorMsg[DRIVE_NOT_READY                 ] := 'DRIVE NOT READY';
ErrorMsg[NO_REP_WITH_SEMI                ] := 'NO REP WITH SEMI';
```

```
ErrorMsg[INSUFFICIENT_PROG_MEMORY  ] := 'INSUFFICIENT PROG MEMORY';
ErrorMsg[PROB_CREATING_WRITE_FILE  ] := 'PROB CREATING WRITE FILE';
ErrorMsg[IN_RUN_MODE                 ] := 'IN RUN MODE';
ErrorMsg[ADDRESS_EXCEEDS_3FC  ] := 'ADDRESS EXCEEDS 3FC';

  Writeln('EXAMPTP.PAS');
  Writeln;

  Writeln('Initialize board 0 at address 310 hex');
  Writeln;

  stcinit(0,$310,err);
  if (err <> NO_ERROR)
    then ErrHandler(err);

  Writeln('Send ',BeginCmd,' command');
  stccmd(BeginCmd, err, pos);
  if (err <> NO_ERROR)
    then ErrHandler(err);

  Writeln('Send ',ArmCmd, ' command');
  stccmd(ArmCmd, err, pos);
  if (err <> NO_ERROR)
    then ErrHandler(err);

  Writeln('Send ',WaitCmd,' command');
  stccmd(WaitCmd, err, pos);
  if (err <> NO_ERROR)
    then ErrHandler(err);

  Writeln('Send ',EndCmd,' command');
  stccmd(EndCmd, err, pos);
  if (err <> NO_ERROR)
    then ErrHandler(err);
  Writeln('Save program in STC #0 to file ',FileName);
  stcdump(FileName, err);
  if (err <> NO_ERROR)
    then ErrHandler(err);

  Writeln('Initialize board 1 at address 314 hex');
  stcinit(1,$314,err);
  if (err <> NO_ERROR)
    then ErrHandler(err);

  Writeln('Load file ',FileName,' to STC # 1');
  stcload(FileName, err);
  if (err <> NO_ERROR)
    then ErrHandler(err);

  Write('Send ',XCmd,' command to start execution ');
  Writeln('of STC #1');
  stccmd(XCmd, err, pos);
  if (err <> NO_ERROR)
    then ErrHandler(err);
```

```
Writeln('Switch driver control back to STC #0');
stcset(0,err);
if (err <> NO_ERROR)
  then ErrHandler(err);

Write('Send ',TrigCmd,' command to send triggers ');
Writeln('from STC #0 to STC #1');
stccmd(TrigCmd, err, pos);
if (err <> NO_ERROR)
  then ErrHandler(err);

Writeln('Switch driver control to STC #1');
stcset(1,err);
if (err <> NO_ERROR)
  then ErrHandler(err);

repeat
   stcstat('trig:rep;', err, pos,RetValue[0]);
   Writeln('Waiting for triggers; REP = ',RetValue[0]);
   if (err <> NO_ERROR)
     then ErrHandler(err);
until (RetValue[0]=0); (* wait for trigger inputs *)
```

```
repeat
   stcstat('wait;', err, pos,RetValue[0]);
   Write('Time remaining = ');
   if (err <> NO_ERROR)
     then ErrHandler(err);
   TimeRemaining := RetValue[0];

   case (RetValue[1] and $c0) of
      $c0: TimeRemaining := 1000*TimeRemaining;
      $80: TimeRemaining := 100*TimeRemaining;
      $40: TimeRemaining := 10*TimeRemaining;
   end;

   if (TimeRemaining < 1000)
      then Writeln(TimeRemaining, ' usecs')
      else
      begin
         TimeRemaining := TimeRemaining/1000.0;
         if (TimeRemaining < 1000)
         then Writeln(TimeRemaining,' msecs')
         else
         begin
            TimeRemaining := TimeRemaining/1000.0;
            Writeln(TimeRemaining,' secs');
         end;
      end;

   stcstat('status;', err, pos,RetReg[0]);
   Writeln('Checking status = ',RetReg[0],' hex');
   if (err <> NO_ERROR)
     then ErrHandler(err);
until (($01 and RetReg[0])=0); (*wait while program is executing *)

stcstat('wait;', err, pos,RetValue[0]);
Writeln;
Writeln('Timed out. Wait returns ');
if (err <> NO_ERROR)
  then ErrHandler(err);
```

```
   TimeRemaining := RetValue[0];
   case (RetValue[1] and $c0) of
      $c0: TimeRemaining := 1000*TimeRemaining;
      $80: TimeRemaining := 100*TimeRemaining;
      $40: TimeRemaining := 10*TimeRemaining;
   end;

   if (TimeRemaining < 1000)
      then Writeln(TimeRemaining, ' usecs')
      else
      begin
         TimeRemaining := TimeRemaining/1000.0;
         if (TimeRemaining < 1000)
         then Writeln(TimeRemaining,' msecs')
         else
         begin
            TimeRemaining := TimeRemaining/1000.0;
            Writeln(TimeRemaining,' secs');
         end;
      end;

   Writeln;
   Writeln('NOTE: At completion WAIT; returns the original delay.');
end.
```

Chapter 5

CREATING PROGRAMS FOR Trigger Master MEMORY

5.1 INTRODUCTION

One Trigger Master feature is its ability to run programs from its own memory. If your word processor can create an ASCII/DOS output, you can create programs and then "compile" them using the STCCOM.EXE program. The section "The Command Set" in Chapter 3 describes the available commands. All commands, except BEGIN, CONT, and X, are supported in programs for Trigger Master memory (the END command is optional). You may use tabs, spaces, and returns to make your file more readable and enter comments with beginning and ending asterisks.

Note: You can also develop programs from PLAYWIN.EXE and PLAYDOS.EXE.

To use STCCOM enter the following command line at the DOS prompt.

```
stccom YourSourceFile
```

STCCOM generates a list file with the suffix .LST; this file lists each command on a separate line.

- If there is no error in the command, the first column of the listing shows the location in memory where the command starts and the command.
- If there is an error, the command is listed followed by the error message.

If your file contains no errors, STCCOM generates a load file with the suffix .LOD; this file can be loaded into Trigger Master using STCLOAD.

This sample source file produces the listing which follows:

Source File

```
* This is a file for use with STCCOM *

do 23;          * perform loop 23 times *
   Arm:rep40;   * arm to detect 40 triggers on line 1 *
   wait 200u;   * wait 200 micro seconds *
   trig2;       * generate trigger on line 2
loop;
```

List File

```
07/14/92
16:16:33

* This is a file for use with STCCOM *
   0                                                        do 23;
* perform loop 23 times *
Arm:rep40; ARM_NEEDS_LINE
* arm to detect 40 triggers on line 1 *
   4                                                        wait 200u;
* wait 200 micro seconds *
  10                                                        trig2;
* generate trigger on line 2
loop;
 TERM_COMMENT_WITH_ASTERISK

Total Errors Detected = 2
```

If you correct the source file errors and recompile the file, the following files are produced:

Source File (corrected version)

```
* This is a file for use with STCCOM *

do 23;          * perform loop 23 times *
   Arm1:rep40;  * arm to detect 40 triggers on line 1 *
   wait 200u;   * wait 200 micro seconds *
   trig2;       * generate trigger on line 2 *
loop;
```

Listing File

```
This is a listing for try2.scr created
07/14/92
16:18:21

* This is a file for use with STCCOM *
   0                                                        do 23;
* perform loop 23 times *
   4                                                        Arm1:rep40;
* arm to detect 40 triggers on line 1 *
  10                                                        wait 200u;
* wait 200 micro seconds *
  16                                                        trig2;
* generate trigger on line 2 *
  25                                                        loop;

Total Errors Detected = 0
LOD file created with 26 entries
```

Chapter 6

CREATING A BACKGROUND DATA ACQUISITION SYSTEM FOR DOS

6.1 INTRODUCTION

Trigger Master is capable of performing the control functions required by a small data acquisition system. While running a program from its own memory, Trigger Master can generate trigger outputs, wait for trigger inputs, and generate delays. By occasionally calling on the PC to move data, make decisions, or bring other resources to bear, Trigger Master can implement a data acquisition system that shares the PC with another DOS function.

This chapter describes creating a Terminate-and-Stay-Resident (TSR) program for Trigger Master. The TSR program operates in the background while you execute another DOS program. You create the TSR by linking STRCRUNC.OBJ to the program module that you have written and compiled in C. Execute the resulting .EXE file by typing the file name. For example, to execute the sample program TSRC, type the following at the DOS prompt:

```
>TSRC
```

You can only install a Trigger Master TSR once. If you attempt to install a second TSR while one Trigger Master TSR is currently installed, the system will return an error message. You must de-install one TSR before installing another Trigger Master TSR or running a TSR again. De-install a TSR by using the /d option in the command line, as shown in the following example:

```
>TSRC /d
```

When you de-install a TSR, you receive a message that the TSR is de-installed.

If you successfully de-install the TSR, you can then run your program again. TSRs control certain computer interrupts; when you attempt to de-install the TSR, the TSR tries to return interrupts to the state prior to the installation of the TSR. If this procedure is successful, the TSR is de-installed.

> **Note:** Occasionally, another program that controls interrupts will execute after the TSR is installed. If this occurs, you may have to reboot the computer in order to run the TSR again.

The TSR program executes as any other program. When the TSR terminates, it returns the DOS prompt allowing you to run another program. Unlike other programs, however, a portion of the TSR code remains resident in memory when the TSR terminates.

While your computer is executing your current application, such as a word processor or spread sheet application, the TSR code remains dormant. When certain events occur, such as an interrupt from Trigger Master, the computer switches to the TSR resident code. The TSR performs whatever task Trigger Master requires and then returns to the previously executing application. Other events that can switch control to the TSR are interrupts from a GPIB controller or another board in your computer. Generally, you will be unaware that you are sharing the computer; you may notice an interruption if the computer must move a lot of data between instruments or disks.

This chapter discusses the general structure of the TSR and the functions that are available to log data and wait on interrupts. The chapter also presents an example of a TSR, a portion of the log file generated by the example TSR, and the details involved in constructing a TSR in the C programming language.

6.2 THE TSR STRUCTURE

Create your TSR by linking your C program to the object file STCRUNC.OBJ. If you are using a GPIB controller, you must include the appropriate IEEE library in your link statement.

You must name the main module of your program STCTEST. The STCTEST module will call procedures from STCRUNC.OBJ, IEEE.LIB, or custom functions and procedures. Although, all your test programs must contain STCTEST, your source and .EXE files can be given any name. The details of constructing the TSR are discussed later in this chapter.

The intent of the sample TSR is to use one Trigger Master to control a measurement and then log data to a disk. Therefore, this TSR makes use of specialized and restricted calls. These calls, which are discussed in detail later in this section, are as follows:

NOKPC488	Produces an error exit if a KPC488.2TM controller is not found.
MISSINGGPIBDEV	Produces an error exit if a GPIB device is not found.
STCRUN	Initializes Trigger Master and GPIB (if used), starts Trigger Master program execution, and converts your program into a dormant TSR that waits for a Trigger Master interrupt.

Note: The STCRUN call must appear in your program following the NOKPC488 and MISSINGGPIBDEV calls, but before any other calls.

WAITONSTC	Restarts Trigger Master and waits for Trigger Master interrupts.
JMPWAITSTC	Starts Trigger Master at a new program location and waits for Trigger Master interrupts.
WAITONGPIB	Waits for an interrupt generated by a KPC488.2TM controller.
WAITONAUX	Waits for an arbitrary interrupt.

STCFLAG	Returns the value of the Trigger Master flag register to a TSR.
STCLOGBIN	Logs binary data to disk.
STCLOGDATE	Logs the date to disk.
STCLOGFLAG	Logs the Trigger Master flag register value to disk.
STCLOGPROGCNT	Logs the Trigger Master program location to disk.
STCLOGSTR	Logs a string to disk.
STCLOGTIME	Logs the time to disk.
STCEXIT	Terminates the TSR.

Normally, your STCTEST program will perform certain initialization functions. If you are using a KPC488xxx GPIB controller, you will generally initialize the controller and certain other devices. If the controller or devices are not present, you should exit the program using the NOKPC488 or MISSINGGPIBDEV calls; this procedure displays an error message.

At some point you will call STCRUN to start executing the Trigger Master onboard program, convert the program into a TSR (with a return to DOS), and cause the TSR to wait for an interrupt from Trigger Master. When Trigger Master generates an interrupt, control is returned to the TSR. At that point, you can take one of the following actions:

- Log data to the disk using one of the STCLOGxxx functions.
- Read the Trigger Master flag register using STCFLAG.
- Put the TSR back to sleep by resuming operation of Trigger Master and then generate another interrupt using WAITONSTC or JMPWAITSTC.
- Put the TSR back to sleep to wait for another interrupt using WAITONGPIB or WAITONAUX.
- Terminate the TSR using STCEXIT.

NOKPC488 and MISSINGGPIBDEV

You can use one KPC488xxx GPIB controller at the standard address in your TSR. At the beginning of your program, you can use the IEEE library call gpib_board_present to determine if a GPIB controller is present. If a controller is not present, you should call NOKPC488 to exit your program. Since the TSR can only support a single GPIB controller, the NOKPC488 call does not require an argument.

If you are using GPIB devices, you may find that a GPIB device times out. If this happens, you should exit your program by calling MISSINGGPIBDEV with the GPIB device address as the argument.

If you have not yet called STCRUN, the calls NOKPC488 or MISSINGGPIBDEV will display an error message on your monitor. Otherwise, a tone is generated, an error message is placed in your log file, and program execution stops.

STCRUN

STCRUN must be the first Trigger Master call (following the calls NOKPC488 and MISSINGGPIBDEV) in your program. STCRUN initializes Trigger Master and software, causes Trigger Master to start executing its program, and starts your program operating as a dormant TSR waiting to be awakened by an interrupt from Trigger Master.

STCRUN accepts seven arguments and incorporates the functions of the STCINIT and STCLOAD calls. The first argument is your Trigger Master board address and the second argument is the name of the file to be loaded into Trigger Master program memory. The third argument is the name of the disk file to which the TSR is to log data. The fourth argument is the interrupt level specified for Trigger Master (the purpose of this TSR is that it will remain dormant until awakened by an interrupt from Trigger Master). STCRUN checks for a valid XT or AT level, but has no way of knowing if you are using an AT board.

Every call which waits for an interrupt accepts a "ticks" argument. This argument allows you to specify the length of time the TSR should wait for an interrupt to occur before logging off with an error message.

- A "ticks" value of 0 disables time checking; the TSR will never log off, but will wait forever for an interrupt.

- Nonzero "ticks" values specify the number of computer clock interrupts the TSR should wait before logging off; the computer generates a clock interrupt about 18.2 times per second or about 1 clock interrupt every 55 msec. You should always specify a minimum of 2 ticks since you may set your ticks just before a clock interrupt.

STCRUN accepts two final arguments that are interrupt levels generated by either your KPC488xxx GPIB controller or an auxiliary board. Choosing level 0 for these interrupts disables the interrupts. If you choose a level for the GPIB controller, the KPC488xxx will be initialized to generate an interrupt on the receipt of an SRQ (GPIB Service Request). You must perform a serial poll of the GPIB device to clear the SRQ.

Setting up a nonzero auxiliary interrupt sets up an interrupt handler for the interrupt; it is your responsibility to program the card to generate the desired interrupt and, if necessary, to clear the board's interrupt.

STCRUN checks for valid interrupt levels and verifies that different levels are used by the different functions, however STCRUN does not know if you are duplicating a level that is used by another board in your computer.

> **Note:** If you use an AT level, ensure your board supports the AT levels. You board jumpers must reflect the levels specified in STCRUN.

If STCRUN encounters a problem, such as a opening a file or a duplicate interrupt level, it displays an error on the monitor and terminates program execution. If STCRUN does not encounter any problems, it opens a log file, your program becomes a TSR, and the computer displays a line indicating the TSR is installed. You are then returned to the DOS prompt.

The opened log file contains a standard header similar to the following:

```
Prog Name: TSRC.EXE
Date: NOV 02, 1992
Time: 14:27:53.76
Memory paragraphs used (in hex): 0510
```

The first three lines contain the program name and date and time of creation. The fourth line indicates the number of memory paragraphs (in hexadecimal notation) your TSR occupies. To convert the memory paragraphs to bytes, add a 0 to the end of the number and convert to decimal. In the example, the TSRC.EXE program uses about 21 Kbytes (5100(hex)). The program contains the GPIB library.

WAITONSTC and JMPWAITSTC

Calling WAITONSTC or JMPWAITSTC executes Trigger Master again and puts the TSR back to sleep (to wait for another Trigger Master interrupt). These calls contain the tick parameter, described for STCRUN to avoid a hang condition if no interrupt is generated. Without the tick parameter, the WAITONSTC command is basically a CONT:INT command that causes the Trigger Master program to start executing at the next step.

The JMPWAITSTC command accepts a second parameter which tells the program where to start execution and to perform an X nn:INT command. This process allows you to have multiple programs resident in Trigger Master program memory, jumping to different code based on previous results or repeating a program any number of times. If you create your Trigger Master program as an ASCII file and compile it using STCCOM, you can determine your jump addresses from the list generated by STCCOM.

WAITONGPIB and WAITONAUX

Calling WAITONGPIB or WAITONAUX puts the TSR back to sleep to wait for an interrupt from a KPC488xxx controller or other card. Both calls contain a tick parameter (refer to the STCRUN description for further information).

STCFLAG

The STCFLAG call returns the value of the STCFLAG register, enabling a program to make decisions based on the progression through the Trigger Master program.

STCLOG

Since the purpose of the TSR is to log data to a disk, there are a number of log calls to facilitate the logging function. If a problem is encountered logging data (for example, a diskette is absent), the TSR will create a tone and attempt to log the data again after approximately 30 seconds. The TSR will make 10 attempts to log the data, and, if it is still unsuccessful, will stop executing. The following list describes the log calls.

STCLOGBIN Logs a block of memory bytes. You must specify the beginning address of the memory area and the number of bytes to be logged.

STCLOGDATE Acquires the current date from DOS and logs it to the disk as shown in the following example:

```
NOV 02, 1992
```

STCLOGFLAG Acquires the current value of the Trigger Master flag register and logs it to the disk.

STCLOGPROGCNT Acquires the current value of the Trigger Master program memory location and logs it to the disk.

STCLOGSTR Allows you to annotate your log. For example, the C code:

```
stclogstr("\r\nStart of GPIB loop\r\n");
```

results in the entry:

```
Start of GPIB loop
```

STCLOGTIME Acquires the current time from DOS and logs it to the disk as shown in the following example:

```
14:27:53.76
```

STCEXIT

The STCEXIT call shuts down the TSR when the TSR is done. STCEXIT disables any board interrupts and attempts to replace any original interrupt vectors intercepted by the TSR. The TSR generates a log-off entry in the log file similar to the following example:

```
Program Terminated Normally
NOV 02, 1992
14:28:18.25
Vector 08h returned
Vector 09h returned
Vector 10h returned
Vector 13h returned
Vector 16h returned
Vector 28h returned
Vector 0Dh returned
Vector 74h returned
```

The "Vector ... returned" lines show the interrupts that the TSR intercepted. In this example, STCEXIT was able to return all the vectors to the ones that were in place before the program ran; it may now be possible to deinstall the TSR. The program code is also still resident and monitoring the "multiplex" interrupt. To remove the code from memory, deinstall the TSR by running it again using a "/d" option. In the example, the TSR has the executable file TSRC.EXE. To deinstall the TSR, enter the following:

```
>TSRC /D
```

The monitor displays a message indicating the success or failure of deinstalling the TSR. If the log indicates that one or more of the vectors could not be returned, do not deinstall the TSR.

6.3 A TSR LOG

The following example illustrates a portion of the log generated by the program in the section that follows. Data that has been deleted from the example is indicated using an ellipsis (...).

```
Prog Name: TSRC.EXE
Date: NOV 02, 1992
Time: 14:27:53.76
Memory paragraphs used (in hex): 0510

Start of GPIB loop
14:27:53.98
14:28:05.07
At end of GPIB loop
NDCV+03.98368E+0,B001
NDCV+03.98736E+0,B002
. . .
NDCV+04.31335E+0,B099
NDCV+04.31571E+0,B100

Start of Trigger Master loop
14:28:06.39
14:28:17.05
At end of Trigger Master loop
NDCV+04.31937E+0,B001
NDCV+04.32178E+0,B002
. . .
NDCV+04.54026E+0,B099
NDCV+04.54151E+0,B100

Program Terminated Normally
NOV 02, 1992
14:28:18.25
Vector 08h returned
Vector 09h returned
Vector 10h returned
Vector 13h returned
Vector 16h returned
Vector 28h returned
Vector 0Dh returned
Vector 74h returned
```

6.4 A TSR EXAMPLE

The following C program can run as a TSR.

```
//*************************************************************//
// TSRC.C is C source code for a program which illustrates the
//   use of stcrunc.obj and ieee-c.lib to create a TSR program
//   to run an Trigger Master.
//
//   This example uses a PCIP-AWFG as a source and a Keithley
//   196 as a measuring and data storage device. Alternatively, a
//   DAS-50 could be used to collect data.
//
//   The AWFG is loaded with a sequence of values which it will
//   step through as it receives external triggers from
//   Trigger Master.
//
//   After some initialization the 196 will be triggered to
//   measure the output from the AWFG and store the value in the
//   196's internal memory. Next Trigger Master will trigger the
//   AWFG causing the AWFG to step to its next output value. The
//   Trigger Master will generate a delay to allow the AWFG output to
//   stabilize (as well as any device or circuitry between the AWFG and
//   the meter).
//   The meter will be commanded to make the next measurement
//   and the process will repeat the required number of times.
//   Finally the stored values will be retrieved and stored to
//   disk.
//
//   To illustrate the features of the trigger link the test
//   will be repeated twice: first using the GPIB to control the
//   196 measurement, and then using the trigger link.
//
//   The first time through, GPIB GETs (group execute triggers)
//   will be used to initiate the reading and the 196 will
//   respond with an SRQ (service request). During the second
//   run, Trigger Master will trigger the 196 External Trigger input and
//   monitor the 196 Voltmeter Complete output.
//
//*************************************************************
#include "stcrun.h"          // function prototypes from stcrunc.obj.
#include "ieee-c.h"          // function prototypes from ieee-c.lib.
#define K196 12              // GPIB address of Keithley 196.
#define DATA_PTS 100         // Number of points to acquire - must
                             //  agree with Trigger Master program.
#define DATA_LEN 23          // Length of data string returned by
                             //  Trigger Master.
```

```
void far pascal stctest()
{
  int BrdType;                  // variable for gpib driver board - 0
                                //  if no board present.
  int status;                   // status returned by gpib calls - 0 if
                                //  time out.
  int l;                        // number of bytes transferred by GPIB.
  int poll;                     // result of a serial poll.
  int index;                    // index for repeats.

  static char r[DATA_PTS*DATA_LEN];
                                // array for data returned by K196.
  BrdType = gpib_board_present();
                                // check for presence of KPC488.
  if (BrdType == 0) NoKPC488();

  initialize (21,0);            // make KPC488 a controller
                                // at address 21.

  // ********************** NOTE **************************
  // Comment out the following line if you are using the
  // KPC488.2 controller. The KPC488.2 does not support
  // listener_present.

  if(!listener_present(K196)) MissingGpibDev(K196);        // check for K196

  // Set up to measure using GPIB control
  stcrun(0x310,         // Load Trigger Master at address 310 hex
    "tsr.lod",          // with program TSR.LOD,
    "tsr.log",          // log data to TSR.LOG,
    5,                  // use interrupt level 5 for Trigger Master,
    22,         // allow 19 ticks(1+ sec) for Trigger Master to return
    12,         // use interrupt level 12 on GPIB
    0);         // don't use AUX interrupt
  spoll(K196,&poll,&status); // clear srq

  send (K196,"F0R0T3Q0I0M8Y0K3X",&status);
        // device command to set 196 to:
        // dc volts, autorange, 1 measurement on
        // GET, store each acquisition in 196
        // memory. store continuously,
        // generate SRQ when reading is done,
        // terminate strings with <CR><LF>
        // and do not send EOI.

  poll = 0;
  while (poll&16 != 16)  spoll(K196,&poll,&status);
                        // wait for ready
  stclogstr("\r\nStart of GPIB loop\r\n");
  stclogtime(); // log the time at the start of the loop
  stclogstr("\r\n");
                        // place carriage return and line feed in log
```

```
  for (index = 0;index < DATA_PTS; index++){
    transmit("UNT UNL MTA LISTEN 12 GET",&status);
                        // trigger 196 via GPIB
    waitongpib(95);     // wait for SRQ from 196 with 5-second time out
    spoll(K196,&poll,&status);          // clear srq
    waitonstc(19);                      // wait while Trigger Master triggers AWFG
    }
  stclogtime();                 // log the time at the end of the loop
  stclogstr("\r\nAt end of GPIB loop\r\n");

// The above C language FOR LOOP synchronizes with the following
//  Trigger Master program loop.

//WAIT 10U; * DUMMY WAIT *
//FLAG0:INT;        * RETURN TO PROGRAM *
//              * AS REQUIRED FOR DEVICES BETWEEN AWFG AND METER) *
//DO 99;        * 99 = DATA_PTS-1 (NEED TO MAKE DATA_PTS-1 CHANGES *
//  TRIG1;      * ISSUE TRIGGER TO STEP Trigger Master *
//  WAIT 1U;          * WAIT FOR Trigger Master TO SETTLE
//                    (LEAVE ADDITIONAL TIME *
//                    *  AS REQUIRED FOR DEVICES BETWEEN
//                    AWFG AND METER) *
//  FLAG1:INT;        * INTERRUPT PROG *
//LOOP;
//FLAG2:INT;                  * RETURN TO PROGRAM TO FINISH OUT LOOP *
//
//HALT;               * NOT REALLY REQUIRED BUT ILLUSTRATES
//                     THE USE OF *
//HALT;               * JMPWAIT Trigger Master *

// Set up to retrieve stored values

  send (K196,"B1M0X",&status);      // device command to set 196 to:
                                    // read back memory and not generate SRQs
  transmit("UNT UNL MLA TALK 12",&status);    // set 196 to talk
  rarray(r,DATA_PTS*DATA_LEN,&l,&status);     // enter data from 196
  transmit("UNT UNL",&status);      // shut down GPIB bus
  stclogstr(r);                     // log receive array to disk

  stclogstr("\r\n");                // place carriage return and line feed in log
// Set up to measure using Trigger Master control
  send (K196,"T7M16Q0I0X",&status);       // device command to set 196
                                    // to: make one reading on an external trigger
  waitongpib(95);                   // wait for SRQ from 196
                                    //  with 5 second time out
  spoll(K196,&poll,&status);        // clear srq

  stclogstr("Start of Trigger Master loop\r\n");
  stclogtime(); // log the time at the start of the loop
  stclogstr("\r\n");    // place carriage return and line feed in log
  jmpwaitstc(34,0);     // execute next portion of Trigger Master program
                        // program is separated needlessly just to
                        // illustrate the use of jmpwaitstc
                        // get jump address from tsr.lst
```

```
// * THIS PORTION OF PROGRAM PERFORMS Trigger Master
//  CONTROLLED ACQUISITION *
//
//FLAG3;
//DO 100;     * 100 = DATA_PTS *
//  TRIG1;    * ISSUE TRIGGER TO STEP Trigger Master *
//  WAIT 1U;          * WAIT FOR Trigger Master TO SETTLE
//                    (LEAVE ADDITIONAL TIME *
//              * AS REQUIRED FOR DEVICES BETWEEN AWFG AND METER) *
//  TRIG2;    * ISSUE TRIGGER TO 196 EXTERNAL TRIGGER INPUT *
//  ARM3;     * WAIT FOR RESPONSE FROM 196 VOLTMETER
//             COMPLETE OUTPUT *
//LOOP;
//FLAG4:INT;          * INTERRUPT PROG *
  stclogtime(); // log the time at the end of the loop
  stclogstr("\r\nAt end of Trigger Master loop\r\n");

// Set up to retrieve stored values
  send (K196,"B1M0X",&status);       // device command to set 196 to:
                                     // read back memory and not generate SRQs
  transmit("UNT UNL MLA TALK 12",&status);     // set 196 to talk
  rarray(r,DATA_PTS*DATA_LEN,&l,&status);      // enter data from 196
  transmit("UNT UNL",&status);       // shut down GPIB bus
  stclogstr(r);                      // log receive array to disk
  stclogstr("\r\n");                 // place carriage return and line feed in log
  stcexit();                         // measurement done, shut down TSR
}
```

6.5 CREATING A TSR FOR C

The intent of this TSR is to service Trigger Master and log data in the background. Therefore, your program should not perform I/O other than the special log functions provided. Your program should not use any of the standard C include files; use STCRUN instead of having a function called *main* your program. You can call other functions or procedures from STCTEST.

You should compile your program without stack checking and the normal C libraries. For Microsoft C, compile your program with the following command line:

```
cl /As /Zl /Gs /c yourprog.c
```

Then, link the result to stcrunc.obj and the IEEE library (if required) as follows:

```
link stcrun.obj yourprog.obj,yourprog.exe,,ieee488
```

Appendix A
Trigger Master ERROR MESSAGES

This appendix contains an alphabetical list of Trigger Master error messages and their definitions.

Error Message	Definition
ADD OVER RNG	Address in a BEGIN command exceeds 1023.
ADDRESS EXCEEDS 3FC	The maximum Trigger Master address is 3FC.
ARM NEEDS LINE	ARM command requires line number(s).
DO NEEDS VALUE	The DO command requires a loop count (1 - 4096).
DO OVER RANGE	Number of loops specified with the DO command exceeds 4096.
DRIVE NOT READY	A disk drive was not ready.
DUPLICATE STC ADDR	You tried to initialize a Trigger Master board with the same address as a previously initialized board.
EXCEEDS DO LEVEL	Only 1 nested DO loop is allowed (you have issued 3 DO commands without a LOOP).
FILE NAME TOO LONG	Strings specifying files to read or write cannot exceed 80 characters.
FLAG OVER RANGE	The FLAG command contained a value greater than 255.
ILLEGAL EXTEN	Illegal extension following a : (colon) in a command or request.
INCOMPLETE COMMAND	Command or request may be missing characters.
IN RUN MODE	HALT is the only valid command in run mode.
INSUFFICIENT PROG MEM	There is not enough room in Trigger Master memory for the command.
MSECS OVER RNG	Time specified in the TRIG or WAIT command exceeded 65635 milliseconds.
MSECS UNDER RNG	Time specified in the TRIG or WAIT command was less than .01 (TRIG) or .001 (WAIT) milliseconds.
NEED ANOTHER LINE	A line number followed by a comma in the ARM or TRIG command requires another line number.
NEED EXTENSION	Colon (:) must be followed by an extension.

Error Message	Definition
NEED SEMICOLON	All commands and requests must end with a ; (semicolon).
NEED TIME SCALE	Times for the TRIG or WAIT command must be specified using one of the following letters: s(seconds), m(milliseconds), or u(microseconds).
NEED TIME VALUE	The TRIG and WAIT commands require a time between 10 microseconds and 65.535 seconds (TRIG) and between 1 microsecond and 65.535 seconds (WAIT).
NO COMMAND	Command or request contained no printable characters.
NO ERROR	No error detected.
NO REP WITH SEMI	A TRIG command with the SEMI option cannot also have the REP option.
NOT IN IMMED MODE	The commands BEGIN and CONT can only be used in immediate mode.
NOT IN LOOP	A DO command must be issued before a LOOP.
NOT IN PROG MODE	The commands DO, END, LOOP, and WAIT and the extension SEMI can only be used in program mode.
OUT OF CHARS	Commands, requests, and extensions must be complete.
PER REQUIRES REP	The TRIG command with a REP extension in immediate mode requires a PER extension.
PROB CREATING WRT FILE	File could not be created by STCDUMP.
PROB OPENING READ FILE	File could not be opened for reading.
PROB OPENING WRT FILE	File could not be opened for writing.
PROB READING FILE	File was opened for reading but a problem was encountered while reading.
PROB WRITING FILE	File was opened for writing but a problem was encountered while writing.
REP NEEDS PER OR SEMI	The TRIG command with a REP extension in program mode requires a PER extension and/or a SEMI extension.
REP OVER RNG	Number of repetitions specified in an ARM or TRIG command exceeds 4096.
SECS OVER RNG	Time specified in the TRIG or WAIT command exceeded 65.635 seconds.
SECS UNDER RNG	Time specified in the TRIG or WAIT command was less than .00001 (TRIG) or .000001 (WAIT) seconds.

Error Message	Definition
START COMMENT WITH ASTERISK	Comments used with sources for STCCOM must start with an asterisk.
STC ALREADY ACTIVE	You tried to activate an already active Trigger Master board.
STC NOT INITIALIZED	You tried to activate a Trigger Master board which has not been initialized.
STC NOT PRESENT	The driver can not find a Trigger Master board at the address specified.
STC NUM OUT OF RNG	Use 0-3 to specify a Trigger Master board.
STC PREVIOUSLY INITIALIZED	You tried to initialize a previously initialized Trigger Master board.
TERM COMMENT WITH ASTERISK	Comments used with sources for STCCOM must end with an asterisk.
TIME OVER RANGE	Times for the TRIG or WAIT command exceeds 65.535 seconds.
TIME OVER RESOLUTION	Times for the TRIG and WAIT commands in the range of 65536 through 99999 can only be specified to four digits (65540-99990). **Note:** When this error occurs, the time will frequently also be over range unless you are in the microsecond range.
TRIG NEEDS LINE	TRIG command requires line number(s).
UNRECOGNIZED COMMAND	Command may be misspelled.
UNRECOGNIZED REQUEST	Requests must be spelled exactly.
UNRESOLVED LOOP	You tried to exit program mode with an END or X command and there are more DO commands in your program than LOOP commands.
USECS OVER RNG	Time specified in the TRIG or WAIT command exceeded 65636000 microseconds.
USECS UNDER RNG	Time specified in the TRIG or WAIT command was less than 10 (TRIG) or 1 (WAIT) microsecond.
WAIT NEEDS VALUE	The WAIT command requires a time with a scale.
X OVER RANGE	The address with the X command exceeds 1023.

Appendix B
COMMAND QUICK START

This appendix contains examples for the structure of the strings required by STCCMD to accomplish various tasks. Refer to Chapter 3 for a complete discussion of the strings; see Appendix A for a list of error messages returned by the calls.

B.1 GENERATE TRIGGER OUTPUTS

The following examples generate trigger outputs (Active low pulse, 5us long).

- Generate a trigger on lines 1, 2 and 5:

```
trig 1,2,5;
```

Note: Separate multiple line numbers with commas.

- Generate five triggers, 15 milliseconds apart on line 2:

```
trig 2:rep 5:per 15m;
```

Note: Use s(seconds), m(milliseconds) or u(microseconds) to designate time scales.

- Generate two triggers on line 6 in the semi-sync mode:

```
trig 6:rep 2:semi;
```

Note: Use semi-sync in program mode only.

B.2 WAIT FOR TRIGGER INPUTS

The following examples generate a wait for trigger inputs condition.

- Wait for high-to-low triggers on lines 1, 4 and 5:

```
arm 1,4,5;
```

Note: Separate multiple line numbers with commas.

- Wait for low-to-high trigger on line 6.

```
arm 6+;
```

- Wait for seven repetitions of high-to-low triggers on line 2:

```
arm 2:rep7;
```

Note: Use in program mode only.

B.3 ENTER PROGRAM MODE

The following examples enter program mode.

- Put Trigger Master in program mode and start program execution at location 0:

```
begin;
```

- Put Trigger Master in program mode and start program execution at location 24:

```
begin 24;
```

B.4 SET UP and TERMINATE PROGRAM LOOP (Program Mode Only)

The following examples set up and terminate a program loop:

- Start loop of 45 repetitions:

```
do 45;
```

- Terminate loop:

```
loop;
```

B.5 GENERATE A WAIT (Program Mode Only)

The following example generates a wait condition.

- Generate a wait of 3.22 seconds:

```
wait 3.22s;
```

Note: Use s(seconds), m(milliseconds) or u(microseconds) to designate time scales.

B.6 TRACK PROGRAM EXECUTION and GENERATE INTERRUPTS

The following examples track program execution and generate an interrupt.

- Write 76 into the flag register:

```
flag 76;
```

Note: Use the flag request to read the flag register.

- Write 36 to the flag register, generate an interrupt, and halt the program:

```
flag 36:int;
```

Note: Programs using :INT should be started with X:INT and continued with CONT:INT.

B.7 EXIT PROGRAM MODE

The following example exits program mode.

- Exit program mode:

```
end;
```

B.8 INITIATE PROGRAM EXECUTION

The following examples start program execution.

- Start program execution at location 0:

```
x;
```

Note: If the program contains the command `FLAG[nn]:INT`, use `X:INT`.

- Start program execution at location 300:

```
x 300;
```

Note: If the program contains the command `FLAG[nn]:INT`, use `X 300:INT`.

B.9 HALT Trigger Master EXECUTION

The following example halts program execution on Trigger Master.

- Halt Trigger Master program execution:

```
halt;
```

Note: You can use this command in immediate mode or insert it within a program. When you insert the command in a program, restart the program with the command `CONT` from immediate mode.

B.10 CONTINUE EXECUTION of HALTED PROGRAM

The following example continues execution of a halted program.

- Continue execution of halted program:

```
cont;
```

Note: If the program contains the command `FLAG[nn]:INT`, use `CONT:INT`.

Appendix C
REQUEST QUICK START

This appendix presents examples for the content of strings required by STCSTAT to make various requests. Refer to Chapter 3 for a complete discussion of the strings; see Appendix A for a list of error messages returned by the calls.

C.1 Check Remaining Trigger Inputs Established by ARM Command

The following example checks any remaining trigger inputs established by the ARM command.

- Determine if Trigger Master is waiting for trigger inputs:

```
arm:rep;
```

C.2 Check Remaining Trigger Outputs Established by TRIG Command

The following example checks for remaining trigger outputs established by the TRIG command.

- Determine if Trigger Master is outputting triggers:

```
trig:rep;
```

C.3 Check the Actual State of the Trigger Lines

The following example checks the actual state of the trigger lines. This command is useful for hardware debugging purposes if the trigger detect circuitry uses latched edges.

- Check the state of the trigger lines:

```
trig:in;
```

C.4 Check Time Remaining Before Next Trigger

The following example checks the time remaining before the next trigger.

- Check the remaining time:

```
trig:per;
```

Note: After executing the trigger, the time is reset to the initial period. Use one of the following commands to determine what is happening: FLAG, TRIG:REP, STATUS, or CONT.

C.5 Check Program Progress

The following examples check the progress of the program.

- Check for program still running:

```
status;
```

  Note: Bit 0 (the lowest bit) is set during program execution.

- Check memory location of next instruction to execute:

```
cont;
```

- Check value of flag register that can be updated during program execution to determine location in program:

```
flag;
```

- Check if program has generated interrupt:

```
status;
```

  Note: Bit 3 is set during interrupt request.

C.6 Check Remaining Loop Count

The following example checks the remaining loop count.

- Check progress through current loop:

```
loop;
```

- If program is in nested loop, check count remaining in outer loop:

```
loop:out;
```

  Note: This command is meaningful only within an executing nested loop.

C.7 Check Remaining Delay Time

The following example checks the remaining delay time.

- Check remaining delay time:

```
wait;
```

  Note: After a delay, the time is reset to the initial delay time. Use a `CONT or FLAG request` to determine if you are beyond a wait instruction.