Microprocessor Development Dreams Come True!

On-Line Help menus, Command Glossary, and Word List.

Windows can be used to view source files, previous traces, and more.

Symbol translation or source code line display.

Screen displays scroll off into history buffer – can be viewed later.

Pop-up Mode Selection panel called by soft key.

Disassembly of code in memory can be compared with trace in adjacent window.

Symbolic debug register display.

Context sensitive prompt line.

NEW UniLab II™: FOUR INSTRUMENTS IN ONE!

Here are all the development tools you ever dreamed of integrated into one PC-controlled system:

- **An Advanced 48-Channel Bus State Analyzer**
- **An 8/16-Bit Universal Emulator**
- **A Built-In EPROM Programmer**
- **An Input Stimulus Generator**

The synergy of these instruments that were designed together to work together saves you time and money. All UniLab II commands and menus are seamlessly integrated into a single, super-efficient working environment.

An Integrated Software Environment, too!

Imagine being able to split your screen and look at real-time program traces and the source code that produced them at the same time! Then go to the On-Line Help or pop-up Mode Panels instantly.

If you see something on a trace that doesn't look like last time, you can hold it in one window while you scroll back through your previous displays.

If you set a breakpoint and single-step, you can then go back to using the analyzer without missing a beat. You can even execute a DOS batch file from UniLab to edit, assemble, and link, then automatically load the new program and symbol table. UniLab uses the full power of the PC.

Find bugs fast with Hardware-assisted Debugging

The traditional way to look for bugs is to single-step through suspect parts of the code until you catch it in the act. This requires a lot of guessing and wasting time.

With UniLab's built-in analyzer, you eliminate the guesswork. Just describe the bug symptom as a trigger, and let the UniLab hardware search for it as your program runs in real time. UniLab will show you a trace of the program steps leading up to the symptom, almost like magic.

A friendly user interface

UniLab lets you use commands or menus – or a mixture of both. An on-line manual, soft-key help screens, a glossary of commands and their parameters, with full-screen writeups are also at the ready.

Reconfigure for any 8 or 16-bit processor in seconds

Thanks to our unique approach to emulation, changes between processor types require only cable and diskette changes. At last count, we specifically support over 120 microprocessors.

Bonus! The built-in EPROM Programmer and Stimulus Generator are simply icing on the cake.

Affordable capability

How much does all this superior capability cost? A lot less than our less able competitors, and probably a lot less than you expect. Our products are sold with a Money Back Guarantee, and our crack team of Applications Engineers is standing by if you need help. Get the full story on the amazing UniLab II and how it can liberate your development projects, today.

ACTION COUPON

Send me info on UniLab II and your No-Risk 10-Day Evaluation!

Name:_____ Title: _____

Company: _____

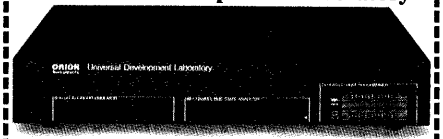
Address: _____

City: _____

State/ZIP: _____

Tel.: _____ Ext. _____

UniLab II™
Universal Development Laboratory



Orion Instruments, Inc.

ORION

702 Marshall Street
Redwood City, California 94063

CALL TOLL FREE: 1-800-245-8500

In California (415) 361-8883

Specifications UniLab II™ – Universal Development Laboratory

Host Computer Interface
RS-232C cable, 19,200 or 9,600 baud, switch selectable.

Diskette Formats
IBM PC 5¼", MS-DOS

In-Circuit Emulator
Download time: 1 second for 2K bytes including 16-bit block error check.
195 ns maximum access time ROM emulation memory standard. For use with microprocessor clock rates up to 10 MHz.
150 ns optional high speed maximum access time ROM emulation memory. For use with microprocessors to 13 MHz.
32K × 8-bit or 16K × 16-bit standard. Programmable by cable, program option.
Expandable to 128K bytes with optional plug-in board.
20-bit enable address decoding.
Individual 2K segments can be selected in any combination within 17-bit field.
Stand-alone operation possible as a ROM emulator.
16-bit idle register loops target CPU allowing loading of emulation RAM and resumption of program execution.
Optional, target processor-specific software gives full debug capability including register and target memory display and change, breakpoints, and single stepping.
Program loading software: from hex or binary disk files, hex serial download, memory image, ROM read.

Bus-State Analyzer
48-bit wide Trace Display and Memory.
48 data inputs. Two groups of 8 can be separately clocked.
6 clock signal inputs. Gated to form one bus clock:
Clock edge filter prevents re-trigger before 100 ns.
395 ns minimum bus cycle (10 MHz 68000).
297 ns with optional high-speed option.
Address demultiplexing latches included – also used by emulator.

Analyzer Trigger
4 step sequential trigger.
RAM truth tables allow search for any function of 8-bits at each 8-bit group, for each step.
8 truth tables per step × 4 steps = 32 @ 256-bit tables.
16-bit inside/outside range detection on address lines.
4-bit segment enable gives 20-bit address capability.
Pass Counter: wait up to 65,382 events or cycles before 4th step.
Before/After/At Pass count trigger enable.
Delay Counter: wait up to 65,382 events or cycles to stop trace.
Filter feature: Records only cycles which satisfy trigger.
Oscilloscope sync output. (Sync on trigger).
Interrupt output: Interrupt target on trigger (if enabled).
LED indicates searching for trigger. Stand-alone operation possible while waiting for trigger.

Software Features
Menu or command driven with single context for all four instruments:
- 48 Channel Bus State Analyzer
- In-Circuit Emulator
- PROM Programmer
- Stimulus Generator

Extensible macro capability.
Cursor key control of text and trace display.
Pop-up mode switch panel.
Split screen displays, user definable.
On-line glossary.
Menu-driven shell displays equivalent command lines.
40 user-definable soft function keys.
Bonus features: Calculator, ASCII table, IC pinout library, memo message feature, direct DOS access, EGA/ECD support (or use monochrome display).
On-line assembler (for selected processors).

Software Options
Graphical Software Performance Measurement.

EPROM/EEPROM Programmer
Smart programming algorithm for high speed.
28-pin zero insertion force socket handles 24 and 28 pin devices.
Programs single supply EPROMs and EEPROMs.
Programs 2716, TMS2516, 2532, 48016, 2732A, 2764/128, 27256/64A/128A, 27512 devices.
Optional module required for 27512.

Signal Inputs
TTL logic levels. (74ALS inputs)
.1 ma maximum loading includes emulator & analyzer.

Signal Outputs
TTL logic levels (74LS244 outputs).
100 ohms forward terminating resistors on Emulator data lines.
Reset output: open collector, 7406 thru 47 ohms.
Interrupt output: open collector, 7406, low true.
9 Stimulus outputs (at EPROM socket) (8255 NMOS outputs).

Physical Data
Size: 2.1 in. high × 13 in. wide × 7.8 in. deep. (53 × 330 × 198 mm. H × W × D).
Weight: 4 lbs. (1.8 Kg). 11 lbs. (5Kg) shipping weight.
Fits easily in a slim-line brief case.

Power
100 KHz switching supply built in.
110v +/– 10% 50/60 Hz input. 15 Watts (standard), or 220v +/– 10% 50/60 Hz input. 15 Watts (optional).

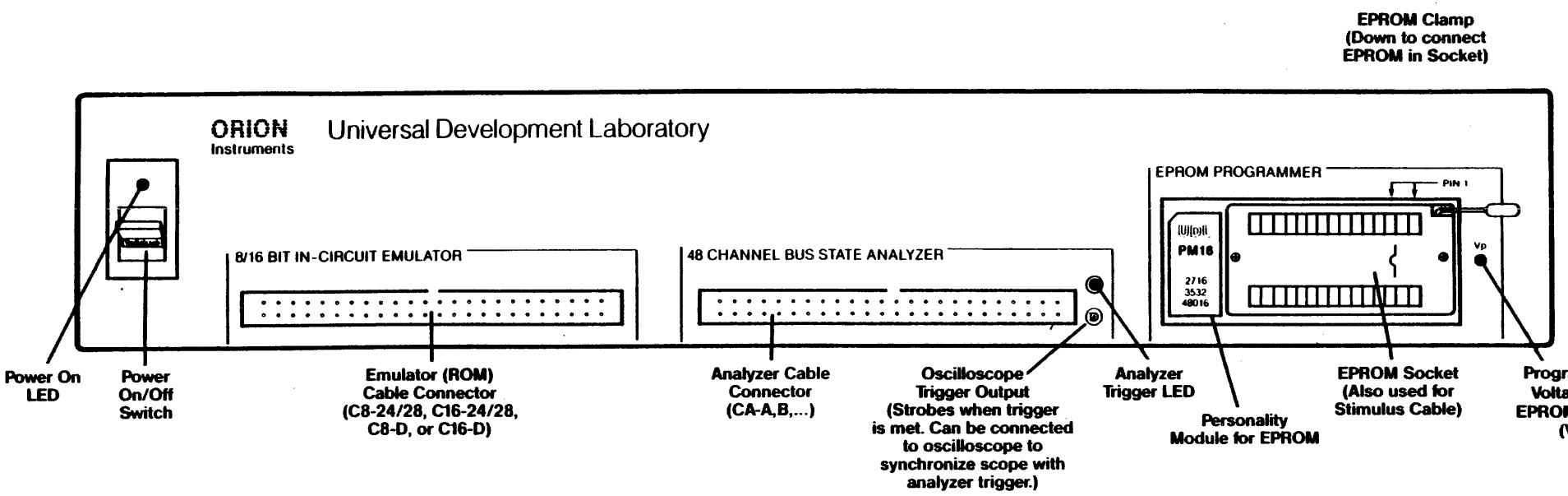
Accessories Included
Users Manual
Reference Manual
Jumper wiring tool (3M)
40-pin IC clip
16-pin IC clip
RS-232 cable
Input stimulus cable
Component clip adaptor probes (2).

Accessory Options
Personality Paks for most popular microprocessors:
ROM Emulator cable 8-bit, 24-pin version unless otherwise specified.
Analyzer cable pre-configured for your target processor.
Disassembler/Debugger Software
Includes single-step, symbolic entry and display, target memory and register display/change, program start/stop/branch, input/output.
Programming Module for 27512.
Personality Paks are available for more than 120 of the most popular microprocessors including: Intel, Motorola, Zilog, National, Rockwell, NEC, RCA, Hitachi, AMD, Mostek, and Signetics. Consult Orion's Personality Pak Portfolio for details.

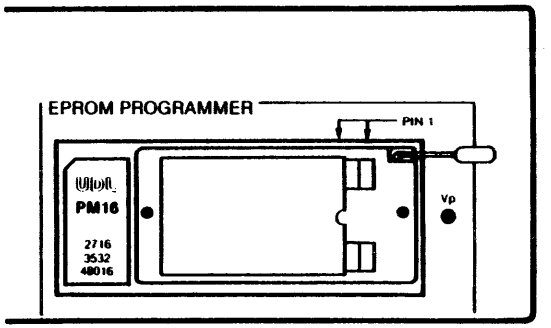
ORION
Instruments

UniLab II™

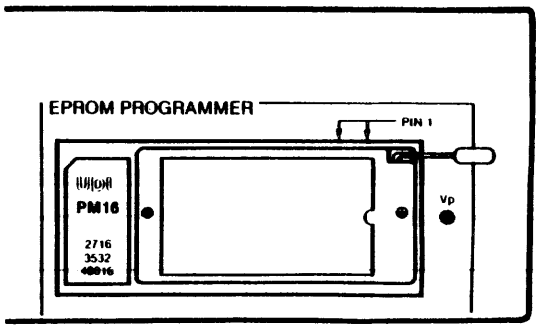
Volume Two
Reference Manual



24-pin Package is shifted all the way to the left



24 Pin EPROM in Programming Socket



28 Pin EPROM in Programming Socket

UniLab Block Diagram

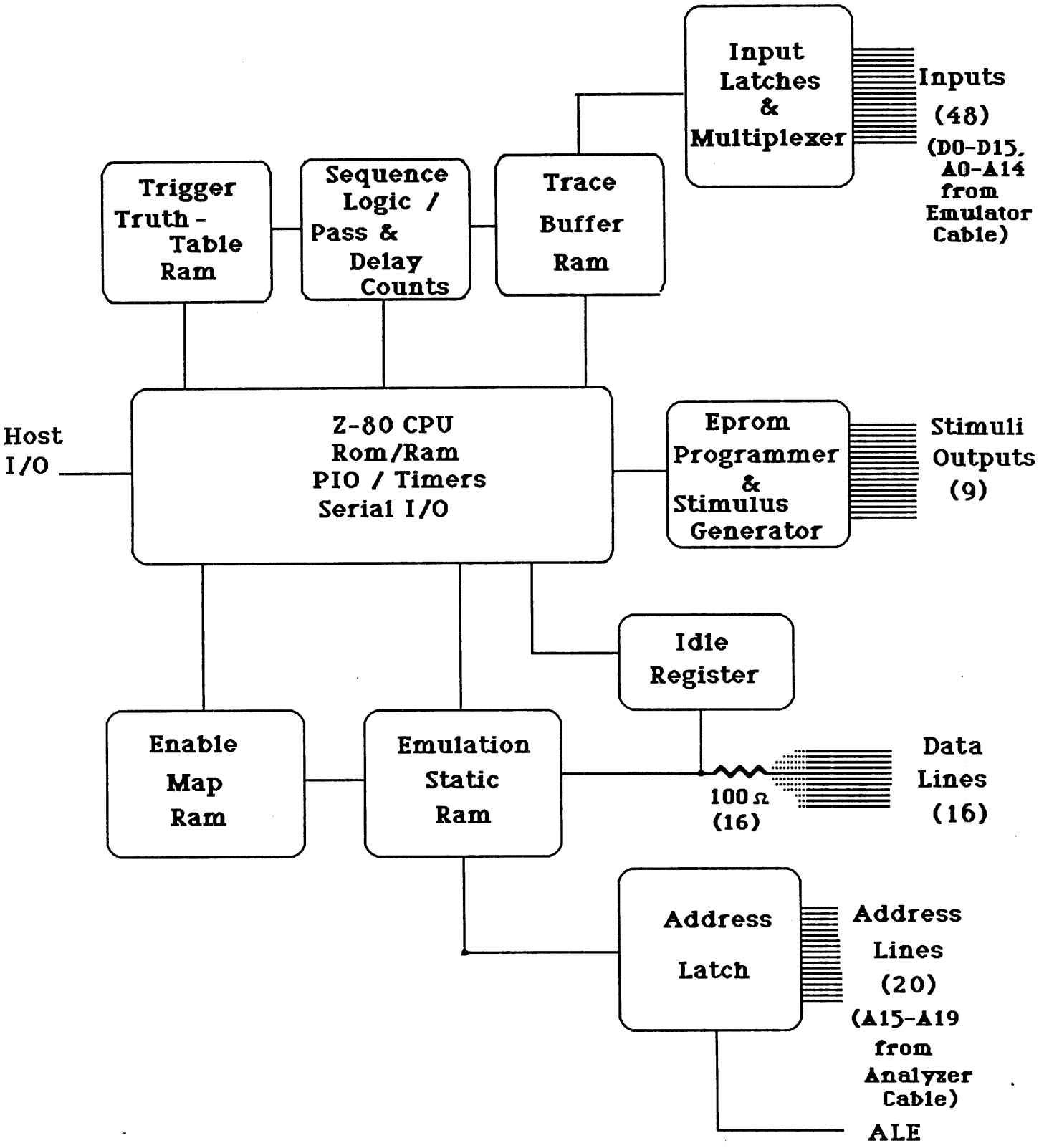


Table of Contents
UniLab Manual

VOLUME ONE
UniLab User's Guide

VOLUME TWO
UniLab Reference Manual

The information in this reference manual applies to both the UniLab and the OptiLab.

Chapter Six:
The UniLab in Detail

Contents:

Overview

This chapter covers the capabilities of the UniLab II in detail. It's meant primarily as a reference chapter.

This overview discusses the topics covered by the chapter.

Review: What the UniLab does

The UniLab lets you look at the bus activity on your microprocessor control board. The UniLab captures a bus cycle in its trace buffer whenever your microprocessor:

writes data to memory,
reads data from memory,
sends to a port,
reads from a port,
or fetches an opcode from ROM.

Capture bus activity

The UniLab can "freeze" this trace buffer at any time, and thus capture a record of bus activity. It then sends this record to your host computer, where you can:

examine it,
compare it to previous traces,
save it,
or print it.

Each line of the trace display includes the address and the data that appeared on the bus. When you have your trace disassembler enabled, you will see the assembly language instructions that were fetched from ROM.

See: **Section One: Interpret the Trace Display**

Program memory

Before you can capture a trace of your program, you have to load it into the UniLab's emulation memory (except when you run the program from a PROM chip-- see page 6-38).

See: **Section Two: Ready and Load Memory.**

Once you have the program in emulation ROM, you can look at the program, and change it.

See: **Section Three: Examine and Alter Memory.**

Capture the activity you need to see

You want to look at only a few of the millions of bus cycles that happen each second. You tell the UniLab what cycles you want to see by describing a "trigger event." The UniLab watches for that event on the bus.

See: **Section Four: Set a Trigger (generate a trace).**

Record what you did

You can save any trace, any section of memory, or the current symbol table. You can also save the current state of the UniLab software.

While you work with the UniLab, you can send all screen displays to the screen and to a file or a printer or both. Or you can choose a mode which logs on the printer only the commands that access memory.

See: **Section Five: Save Information.**

Look at the Internal State of the Processor

You can set a breakpoint in your program, and then restart the target board. The program will run to the breakpoint, then show you the register display when it stops. Or you can use the **NMI** command to achieve instantaneous DEBUG control.

After you have gained debug control you can:

continue to another breakpoint,
single step through your program,
examine and change RAM, emulation ROM, and internal registers,
or leave debug control.

See: **Section Six: Breakpoints and the DEBUG.**

Save your code to silicon

Once you've completed testing your program, you can program an EPROM or EEPROM with the UniLab. See Appendix G for a list of PROMs that Orion supports.

See: **Section Seven: Program EPROMs.**

"Mock up" peripheral inputs

Sometimes you need to see how your microprocessor board responds to an input from a peripheral device. The stimulus generator of the UniLab allows you to produce any 8 bit signal you want-- or toggle individual lines.

See: **Section Eight: Generate Stimuli.**

Make use of special features and shortcuts

The UniLab makes full use of the function keys of your personal computer, including **ALT**ered, **SHIFT**ed, and **CTRL**ed function keys, and the keys of the numeric key pad.

Some of the function keys are pre-assigned to help screens (see On-Line Help chapter) or to commands. The others are left available for you to assign as you please.

See: **Section Nine: Special keys.**
See also the **Special Features** section of Chapter Three.

Function key 8 has a special effect-it gives you access to the pop-up panels, where you can easily change many options, including display and logging features.

See: **Section Ten: Mode Panels.**

Function key 2 also is special-- it splits the screen, gives you the ability to look at different parts of your trace at the same time, or to examine a textfile along with a breakpoint display, or . . .

See: **Special Features** section of Chapter Three.

1. Interpret the Trace Display

Introduction

This section covers the trace display-- the record of bus activity that the UniLab captures for you.

The trace examples show a Z80 processor and an Intel 8096 processor.

Why you care about the trace display:

You want to find the bugs in your system. Bugs cause undesirable behavior in your system, which you can track down by looking at the record of bus activity on your board-- the trace display.

Contents

1.1 Feature Summary

While you examine a trace, you can turn these options on and off to alter the display:

Option	Mode Panel	Commands
Disassemble code	Yes	DASM DASM'
Substitute symbolic names for numbers	Yes	SYMB SYMB'
Show CONTrol column	Yes	SHOWC SHOWC'
Show MISCellaneous column	Yes	SHOWM SHOWM'
Binary number base for MISC	Yes	2 =MBASE
Fixed header	Yes	HDG HDG'
Stop display after each screen	Yes	PAGINATE PAGINATE'
Define symbolic names	NO	IS SYMFILE SYMLOAD
Show source lines in trace	NO	SOURCE SOURCE'

Mode panels:	**Commands:**
1. ANALYZER modes	
DISASSEMBLER	DASM DASM'
SYMBOLS	SYMB SYMB'
2. DISPLAY modes	
MISC COLUMN	SHOWM SHOWM'
CONT COLUMN	SHOWC SHOWC'
MISC # BASE	=MBASE
PAGINATE	PAGINATE PAGINATE'
FIXED HEADER	HDG HDG'

You can look at any portion of a trace you want:

Feature	Cursor key	Command
Show trace from top	HOME	**TT**
Show next step of trace	Down Arrow	none
Show next page of trace	PgDn	**TR**
Show trace from step <n> (resets default to n)	none	<n> **TN**
Show trace from step <n>, with no effect upon the default	none	<n> **TNT**
Dump trace buffer from UniLab	none	**TD**

You can save and compare traces (details in **Save Information**):

Feature	Command
Save a trace to a file	**TSAVE** <file name>
Compare last <n> cycles of saved trace to current trace	<n> **TCOMP** <file name>
Compare saved trace to result of current trigger specification	<count> **SC** <file name>

1.2 The Trace: What Each Column Means

The header line of the display labels all but one of the columns:

cy# CONT ADR DATA (unlabeled column) **HDATA MISC**

Each column displays a different piece of information:

cy# shows you what cycle you are looking at, relative to the trigger event. The trigger event is always labeled as cycle zero.

This column starts with an **f** when you produce a filtered display.

CONT shows you what the UniLab sees on the control inputs, and on the upper four bits of the address inputs.

The UniLab uses four of its inputs, labeled as

C7, C6, C5, and **C4**

to determine whether the bus cycle is a fetch, read, or write. The first digit of the **CONT** column shows those four inputs as a hexadecimal digit. The disassembler needs this information, but you can ignore it-- except when you are trouble shooting the wiring of the connection from your UniLab to your target board.

The second digit shows the four highest bits of the 20 bit address inputs to the UniLab, labeled as

A19 through **A16.**

While working with most 8-bit processors, these wires are not attached to anything, and so float high, at logic level one. The Z80, 8085, and NSC-800 processors don't follow this general rule-- they have one of these upper four wires connected to a processor pin. See the explanation on page 6-13.

You can use the mode panel (hit function key 8) to hide this column.

ADR shows the first 16 bits of the address bus, A0 through A15. See the Disassembler Note below.
The highest four bits, A16 to A19, appear as the right-hand digit in the CONT column.

DATA shows you what data was put on the bus. Depending on the type of the cycle, that "data" is either a data value or a machine language instruction. The data is 8 bits or 16, depending on the processor.

The center (unlabeled) **column**
shows the disassembled instructions. Data reads and writes are also identified.

This column appears when you work with the disassembler enabled, as you usually will.

HDATA shows you what values the UniLab reads on D8 through D15. This column only appears with 8 bit processors.

The UniLab doesn't use the full 16 bits of data input when working with processors that have an 8 bit wide external data bus. That makes these 8 inputs available to you to gather more information about the outputs of other chips or ports on your board.

MISC shows you what values the UniLab reads on M0 through M7, the **MISC**ellaneous inputs. These wires are always available for you to connect anywhere you want on your board.

The number base is normally binary, but you can change it with the mode panel (**F8**). You can also use the mode panel to hide this column.

Disassembler note

With a processor-specific disassembler enabled, each line of the trace shows a complete assembly language instruction, no matter how many bytes it takes. On those lines that show an instruction which takes more than one cycle to fetch from memory, the **cy#** column contains the cycle number of the first fetch, and the **ADR** column contains the address of the first byte of the instruction (the first word on 16-bit processors).

The **MISC** and **HDATA** columns show only the state of those inputs during the last cycle of the instruction. Use the mode panel (**F8**) to turn off the disassembler if you want to see the state of these inputs during every bus cycle.

1.3 Sample Traces Examined

This section shows two sample traces and explains the first few lines of both in detail.

The 8-bit processor example shows the Orion test program for a Z80 processor. The 16-bit processor example shows a trace of the test program for the Intel 8096.

A trace of the test program for your processor appears in the Target Application Note for your Disassembler/DEBUG software.

An 8 bit processor: Z80 trace

The following display shows a trace of the test program for the Z80 microprocessor. The test program was first loaded into the UniLab from disk with **LTARG**, and then started up with **STARTUP**. The **STARTUP** command captures a trace of the bus cycles starting at the reset address-- for the Z80, address 0000.

cy#	CONT	ADR	DATA		HDATA	MISC
0	B7	0000	310019	LD SP,1900	11111111	11111111
3	B7	0003	3E12	LD A,12	11111111	11111111
5	B7	0005	015634	LD BC,3456	11111111	11111111
8	B7	0008	119A78	LD DE,789A	11111111	11111111
B	B7	000B	21DEBC	LD HL,BCDE	11111111	11111111
E	B7	000E	C5	PUSH BC	11111111	11111111
F	D7	18FF	34 write		11111111	11111111
10	D7	18FE	56 write		11111111	11111111
11	B7	000F	C1	POP BC	11111111	11111111
12	F7	18FE	56 read		11111111	11111111
13	F7	18FF	34 read		11111111	11111111
14	B7	0010	3C	INC A	11111111	11111111
15	B7	0011	3C	INC A	11111111	11111111
		.		.	.	
		.		.	.	
		.		.	.	
		.		.	.	
2B	B7	0027	3C	INC A	11111111	11111111
2C	B7	0028	3C	INC A	11111111	11111111
2D	B7	0029	C30300	**JP 3**	11111111	11111111
30	B7	**0003**	3E12	LD A,12	11111111	11111111

This simple program is an infinite loop. It first initializes several registers, starting with the stack pointer.*

Then the program pushes a value on the stack and pops the same value, to demonstrate the working stack. Notice the cycles associated with memory reads and writes. These show you register and memory locations, just when you most want to know them.

After that come a series of "increment register A" instructions. The last command in the program, at address 29H, is an unconditional jump back to address 3, so that the program goes back to the second instruction.

* You must have a working stack for DEBUG commands to work.

An examination of the first two lines

This section dissects the first two lines of the Z80 trace. For the sake of simplicity, the **HDATA** and **MISC** columns (which were not attached to anything on the board) are not displayed.

```
cy#
  0  B7 0000 310019    LD SP,1900
  3  B7 0003 3E12      LD A,12
```

The first line of the display starts with cycle zero, which means that this cycle was the "trigger event."

The UniLab trace buffer captured a **31** on bus cycle 0, **00** on bus cycle 1, and **19** on cycle 2. These hexadecimal numbers were then translated by the disassembler into **LD SP,1900.**

The second line is labeled as cycle three, which lets you know that the Z80 microprocessor required three bus cycles to read the first instruction from ROM.

```
    CONT
0   B7 0000 310019    LD SP,1900
3   B7 0003 3E12      LD A,12
```

The **CONT**rol column shows two different types of information. The high four bits of the byte (nibble) shows the control inputs, **C4** through **C7.** The low nibble shows the highest four bits of the address inputs, **A16** through **A19.** Both nibbles are important to the proper functioning of the disassembler and emulation ROM. You will only have to pay attention to this column if you suspect that the wires carrying these signals are improperly connected.

The high nibble is used by the processor-specific disassembler to distinguish between cycle types.

If the wires that carry the control signals have been incorrectly connected, then the disassembler will not work properly. The disassembler needs these control signals to classify each bus cycle as a fetch or read or write.

The first two lines both show the microprocessor fetching an instruction from ROM. With the Z80, **B** in the control column always indicates an instruction fetch, **D** marks the write cycles, and **F** marks a read.

```
    CONT
0   B7 0000 310019    LD SP,1900
3   B7 0003 3E12      LD A,12
```

The low nibble carries information that is used by the emulation ROM. It is useful while troubleshooting, but otherwise is only for the curious. If you are curious, read on.

If the value of this nibble matches the value set by **=EMSEG** then the UniLab's emulation ROM will check whether the address is enabled.

The emulation ROM will put data on the bus only when the low 16 bits of the address fall into an enabled range (**EMENABLE**) and the number on inputs **A16** through **A19** match **=EMSEG.**

Every line of the Z80 test program display will have 7 as the upper four bits of the address. Three of the address inputs to the UniLab, A18, A17, and A16, are left to float high.

A19, however, is connected to the MREQ pin of the microprocessor. This "active low" output of the Z80 goes low when the processor Memory is REQuired. The Z80 needs memory access on all bus cycles, except when it writes to or reads from a port.

Thus, these four inputs to the UniLab are usually 0111, which is hexadecimal 7. When the MREQ signal goes high, the 7 becomes F-- for example, when the Z80 is addressing a port address rather than memory.

If you have a different processor, your UniLab's inputs will be connected differently.

```
        ADR
0   B7 0000 310019    LD SP,1900
3   B7 0003 3E12      LD A,12
```

The first line shows address 0000, the reset address for the Z80. The Z80 starts executing code from this address whenever it receives a reset signal. The second line shows address 0003, since the first instruction occupies bytes at addresses 0, 1, and 2.

```
                DATA
0   B7 0000 310019    LD SP,1900
```

The first byte of the first instruction is 31 hex, which decodes as a command to load an immediate value into the stack pointer. The stack pointer of the Z80 holds a 16 bit value.

```
                DATA
0   B7 0000 310019    LD SP,1900
```

That immediate value is 1900. Notice that the two bytes appear on the bus in reverse order, following the Intel convention, rather than the one adopted by Motorola.

```
                DATA
0   B7 0000 310019    LD SP,1900
3   B7 0003 3E12      LD A,12
```

The second instruction loads an immediate value into the A register. This register of the Z80 only holds an eight bit value.

```
                DATA
0   B7 0000 310019    LD SP,1900
3   B7 0003 3E12      LD A,12
```

The whole instruction only takes up 2 bytes, since the Z80 only needs one byte of data for the A register.

An 8096 trace-- 16-bit processor

The display below shows the trace of the test program for the 8096. The trace is shown only to highlight the difference between an 8-bit trace and a 16-bit trace.

Notice that there is no **HDATA** column in the trace, and that instead the UniLab shows 16-bits of data for each bus cycle.

The 8096 has a full 16-bit external data bus. With each bus cycle, the UniLab records a 16-bit word of either opcode or data.

A brief discussion of the trace appears on the following page.

cy#	CONT	ADR	DATA				MISC
0	FF	2080	A1004118		LD	SP,#4100	11111111
2	FF	2084	A100A01C		LD	AX,#A000	11111111
4	FF	2088	A100B01E		LD	BX,#B000	11111111
6	FF	208C	A100C020		LD	CX,#C000	11111111
8	FF	2090	A100D022		LD	DX,#D000	11111111
A	FF	2094	CA1C		PUSH	[AX]	11111111
C	EF	A000	A000	read			11111111
D	CF	40FE	A000	write			11111111
B	FF	2096	CE1C		POP	[AX]	11111111
F	EF	40FE	A000	read			11111111
10	CF	A000	A000	write			11111111
E	FF	2098	071C		INC	AX	11111111
11	FF	209A	071C		INC	AX	11111111
12	FF	209C	071C		INC	AX	11111111
13	FF	209E	071C		INC	AX	11111111
14	FF	20A0	071C		INC	AX	11111111
15	FF	20A2	071C		INC	AX	11111111
16	FF	20A4	071C		INC	AX	11111111
17	FF	20A6	071C		INC	AX	11111111
18	FF	20A8	071C		INC	AX	11111111
19	FF	20AA	071C		INC	AX	11111111
1A	FF	20AC	071C		INC	AX	11111111
1B	FF	20AE	071C		INC	AX	11111111
1C	FF	20B0	E7E5FF		**LJMP**	**2098**	11111111
1F	FF	**2098**	071C		INC	AX	11111111
20	FF	209A	071C		INC	AX	11111111

This simple program functions just about the same as the Z80 test program. Both are infinite loops that first initialize several internal registers, next push and pop a value, and then go through a series of "increment A" instructions.

The last command of the 8096 test program is a jump back to the first "increment A" instruction.

```
      CONT
0   FF 2080 A1004118        LD    SP,#4100
```

Remember that you can usually ignore the CONT column. But if you want to pay attention to it, notice that for the 8096, **F** marks a fetch, **C** marks a write and **E** marks a read cycle.

```
      CONT
0   FF 2080 A1004118        LD    SP,#4100
```

All four of the high address inputs to the UniLab, A19 through A16, float high. They are not attached to anything on the target board.

```
          ADR
0   FF 2080 A1004118        LD    SP,#4100
```

The reset address for the 8096 is 2080. Contrast this to the Z80, which has a reset address of 0000.

```
                DATA
0   FF 2080 A1004118        LD    SP,#4100
2   FF 2084 A100A01C        LD    AX,#A000
4   FF 2088 A100B01E        LD    BX,#B000
```

The opcode A1 decodes as a load of an immediate value into an internal register.

```
                DATA
0   FF 2080 A1004118        LD    SP,#4100
2   FF 2084 A100A01C        LD    AX,#A000
4   FF 2088 A100B01E        LD    BX,#B000
```

The last byte of the 4 byte instruction tells the 8096 which register to load the immediate value into.

```
                DATA
0   FF 2080 A1004118        LD    SP,#4100
2   FF 2084 A100A01C        LD    AX,#A000
4   FF 2088 A100B01E        LD    BX,#B000
```

The two-byte immediate value appears on the bus as the second and third bytes of the instruction. As with the Z80, the bytes appear on the bus in reverse order.

1.4 Moving through Your Trace Display

When the UniLab sends its trace buffer to the host machine, the host displays it starting from either cycle 0 or whatever cycle number was last set with <n> **TN.**

You will sometimes see everything you needed to know in the first screenful of the trace.

But much of the time you will need to look at a different part of the trace.

A small but complete set of commands moves you through the trace buffer.

Dumping the trace

Usually the UniLab will automatically dump the trace into the host computer. But if the trace buffer in the UniLab does not fill (especially when producing aa filtered trace) then you will need to manually dump the trace to the host, with **TD.**

Look at next line of trace

Use the Down Arrow key (number 2 on the numeric key pad) to see the line of the trace that follows the "current" line.

The current line is usually the last one that you displayed on the screen. However, refer to the discussion of the history mechanism on the second page following.

Look at the next screen of trace

Use the Pg Dn key (number 3 on the numeric key pad) to see the next screenful of the trace, starting from the "current" line (or use the command **TR**).

Look at the trace, starting from cycle number <N>

Use one of the two commands: <n> **TN** or <n> **TNT.**

TN will also reset the default cycle number that **T** displays from (normally -5).

Use **TNT** to look at a particular cycle of the trace, without changing the default used by **T.**

To look at the trace starting from the top

The HOME key (number 7 on the numeric key pad) shows you the trace from the top (or use the command **TT**).

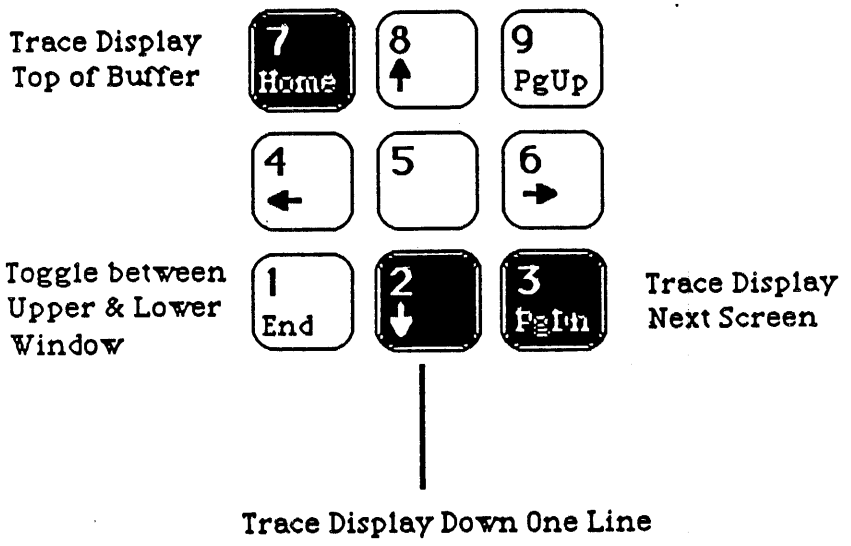
The trace and the "history" mechanism

Everything that goes by on the full screen or the lower window gets saved by the history mechanism of the UniLab. This handy feature allows you to review your past actions and past traces.

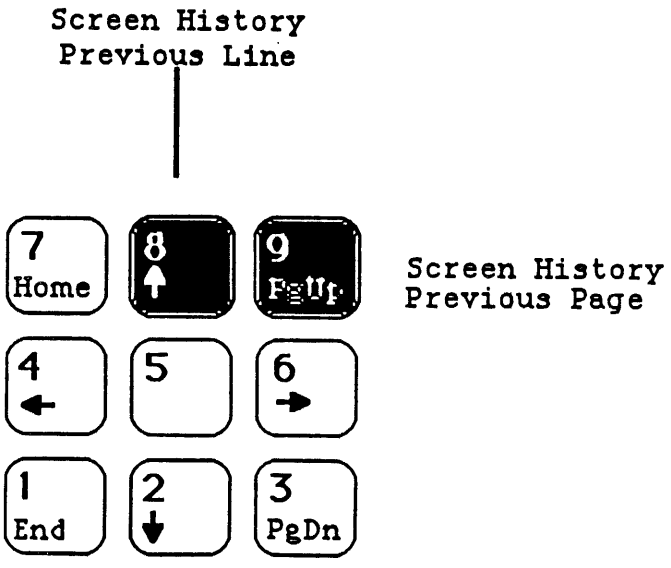
The PgUp key (number 9 on the numeric key pad) shows you one screen full of history.

The Up Arrow key (number 8 on the numeric key pad) shows you one line of the history.

Cursor Keys and the Trace Buffer Display



Cursor Keys and the Screen History



The trace, the history, and the Down Arrow

The UniLab "remembers" the cycle number of the last line of trace buffer you saw. Whenever you use one of the trace commands that start displaying from the "current" cycle number (PgDn, **TR**, or Down Arrow), the UniLab will normally start the display from after that last line.

However, if you use the Up Arrow and PgUp keys to look at the history of your session, you can end up with the cursor sitting on a line of trace display. The UniLab will temporarily call that cycle the current cycle.

If you want to display the trace buffer starting from the cycle that the cursor is on, then you don't need to worry. But if instead you want to start displaying from the line of the trace buffer that you **last** displayed, first press ENTER to get the cursor to a blank line, and then use any display command or key.

1.5 Symbolic Names in the Trace Display

Most people find it convenient to assign symbolic names to numbers. For example, LOOP.START conveys more information than address 2098. You will find your traces easier to read if you have symbolic names assigned to important addresses, ports and data.

You can load in the symbol table that your assembler generated, and have the same symbolic names that your source program assigned.

Or you can assign symbolic names one by one, using the **IS** command to give names to numbers.

You should not use a symbol name that is identical to a UniLab command. That would prevent you from using the command because the new interpretation of the name takes precedence.

Entering **SYMB'** tells the UniLab to ignore the symbol definitions.

Choosing symbol file formats

Enter the **SYMTYPE** command to get the menu of predefined symbol table formats:

SYMBOL FILE FORMAT MENU

```
F1   2500AD SOFTWARE
F2   2500AD SOFTWARE (ABBREVIATED)
F3   ALLEN ASHLEY
F4   INTEL DEVELOPMENT FORMAT
F5   MICROTEK FORMAT
F6   OTHER FIXED FORMAT
F7   MANX AZTEC C  (value, variable_length_name)
F8   AVOCET        (variable_length_name, value)
F10  RETURN TO COMMAND MODE
```

At this point you can select the desired format from the menu. If the format you require is not on the menu, see the subsection on **SYMFIX** on page 6-27 and in Chapter Seven.

Enter **SAVE-SYS** to make the selection permanent. (You can still change it again with **SYMTYPE**, then save the system again.)

If you don't use **SAVE-SYS**, the format you choose will only be used during the current session with the UniLab.

Load symbol table from file

After choosing the symbol file format, use:

SYMFILE <filename>

to load symbols in from a file. You will be prompted for the file name if you do not include it on the command line.

SYMFILE clears out the symbol table before loading the file. You can load in several symbol files, by using **SYMFILE+** to load each additional file.

Define individual symbols

A single symbol can be defined at any time with:

<n> **IS** <name> .

For example if you enter **1234 IS DELAYLOOP**, then DELAYLOOP will be displayed instead of the 1234, whenever 1234 occurs on the trace display.

You can also use DELAYLOOP in trigger specs, or to set breakpoints. For example:

DELAYLOOP AS

Toggle symbol translation on and off

To turn the symbol translation feature on for the trace display with **SYMB** or the Mode Panel (function key 8). Use the Mode Panel or **SYMB'** to turn the symbol translation off.

Note that enabling translation of the symbols will not change anything unless you have some symbols defined.

You can greatly improve readability of a hex trace by identifying the crucial subroutines and storage areas. If you are programming in a high-level language you can identify the run-time routines for improved readability.

Redefine a symbol

You can redefine a symbol at any time, simply by using with **IS** to define it again (only the most recent definition will be found). You cannot clear out only one symbol definition, but you can forget an entire symbol table with **CLRSYM.**

Save a symbol table as a file

You can save an existing symbol table as a named file with **SYMSAVE**, and reload a previously saved table from disk with **SYMLOAD.**

Setting the size of the symbol table

You can allocate up to hexadecimal 80 K (128K decimal) to the symbol table.

The size of the symbol table is set by giving the command:

‹hex # of Kbytes› **=SYMBOLS**

then saving the newly altered UniLab software with **SAVE-SYS.** You must exit the program with **BYE** and start it again.

The size of the symbol table is allocated when the program starts up, and cannot be changed on the fly.

Use the command **?FREE** to find out how many bytes are allocated to the symbol table and to the line history. That display appears in decimal base, not hexadecimal.

Symbol example

The trace printout below shows a disassembled trace with symbol translation.

First eight symbol names were entered by hand:

```
1900  IS Init.Stack
3  IS Start.Loop
29 IS End.Loop
10 IS First.IncA
3456 IS Init.BC
789A IS INIT.DE
BCDE IS INIT.HL
28 IS LAST.INCA
```

And then **F9** was pressed, to get a trace of the startup:

```
cy#                ADR DATA
  0                0000 310019    LD SP,INIT.STACK
  3 START.LOOP     0003 3E12      LD A,12
  5                0005 015634    LD BC,INIT.BC
  8                0008 119A78    LD DE,INIT.DE
  B                000B 21DEBC    LD HL,INIT.HL
  E                000E C5        PUSH BC
  F                18FF 34 write
 10                18FE 56 write
 11                000F C1        POP BC
 12                18FE 56 read
 13                18FF 34 read
 14 FIRST.INCA     0010 3C        INC A
 15                0011 3C        INC A
  .                  .              .
  .                  .              .
  .                  .              .
 2A                0026 3C        INC A
 2B                0027 3C        INC A
 2C LAST.INCA      0028 3C        INC A
 2D END.LOOP       0029 C30300    JP START.LOOP
 30 START.LOOP     0003 3E12      LD A,12
 32                0005 015634    LD BC,INIT.BC
 35                0008 119A78    LD DE,INIT.DE
 38                000B 21DEBC    LD HL,INIT.HL
 3B                000E C5        PUSH BC
```

After these symbols have been loaded in, you can set a trigger or a breakpoint using the symbolic name:

LAST.INCA AS

The last example, below, shows breakpoint displays with these same symbols defined:

```
RESET END.LOOP RB resetting

AF=2B28 (sz-a-pnc) BC=3456 DE=789A HL=BCDE IX=FFFF IY=FDFF SP=1900
END.LOOP     0029  C30300   JP START.LOOP          (next step)  ok

SSTEP NMI
AF=2B28 (sz-a-pnc) BC=3456 DE=789A HL=BCDE IX=FFFF IY=FDFF SP=1900
START.LOOP   0003  3E12     LD A,12                (next step)  ok

N
AF=1228 (sz-a-pnc) BC=3456 DE=789A HL=BCDE IX=FFFF IY=FDFF SP=1900
             0005  015634   LD BC,INIT.BC          (next step)  ok
```

Reading other symbol table file formats

If the format you need is not included in the **SYMTYPE** menu, use **SYMFIX** to describe the format of other fixed length format files.

Fixed length format

The **SYMFIX** command is used to define parameters for any symbol file format which uses fixed length records. The 6 parameters for the SYMFIX command are as follows:

a = offset from start of record to start of name field.
b = 1 if address is 4 ASCII digits or 0 if 16-bit binary.
c = offset from start of record to start of addr field.
d = 1 if binary address has most significant byte first.
e = pad characters used to fill between symbols.
f = record length in bytes

Examples

The format of the 2500AD global symbol table is:

0 0 21 1 0 24 SYMFIX

The format for the ALLEN-ASHLEY symbol table is:

0 1 D 0 21 12 SYMFIX

Variable length format

If you have a variable length format symbol file, use the **AVOCET** choice in the menu if the format is NAME followed by VALUE.

Use the **MANX AZTEC C** choice if the format is VALUE followed by NAME.

1.6 Toggling Display Options On and Off

You use the mode panel to alter the trace display options. Press **F8** to enter mode panel, which will appear in the upper right hand corner of your screen. See section 10 of this chapter for complete information on the mode panel options.

Depending on what you need, you can do everything from displaying only machine code to displaying source code lines in your trace.

These are the trace display options that you can toggle on and off in the mode panel:

Display assembly language instructions in trace display

Substitute symbolic names for numbers

Show CONTrol column

Show MISCellaneous column

Binary number base for HDATA and MISC

Fixed header

Stop display after each screen

Show source lines in trace.

Mode panels in brief

Within the mode panel, use the up and down arrow keys to move from one option to another. Press **F8** repeatedly to rotate through the three panels.

Press the right arrow key to toggle the current option.

Press **F1** to get a brief help message for the current option.

Press **END** to exit from the mode panels.

See section 10 of this chapter for more information.

2. Ready and Load Memory

Introduction

This section covers emulation memory--

how to tell the UniLab what addresses to emulate,
how to load information into emulation ROM,
and how to save the data in emulation ROM.

The UniLab controls your target processor by emulating program memory. When the processor tries to fetch instructions or data from an address that has been **emulator enabled** (**EMENABLE**), the UniLab's emulation ROM responds on the bus.

Remember that the UniLab replaces your ROM rather than your microprocessor, and then watches the bus while the processor runs.

Contents

2.1 Feature Summary

Feature	Menu	Commands
Report Emulation STATus	Yes	**ESTAT**
Choose 64K Segment	Yes	<hex digit> **=EMSEG**
Enable 2K blocks within 64K segment	Yes	<addr> **EMENABLE**
Load from an Intel HEX format file	Yes	**HEXLOAD**
Load from a binary file	Yes	<from addr> <to addr> **BINLOAD**
Save a block of memory to disk	Yes	<from> <to> **BINSAVE**
Enable minimal memory and load test program	Yes	**LTARG**
Load from a ROM	Yes	Commands are available (see below), but use of the menus is recommended

Commands: **Menus:**

ENABLE PROGRAM MEMORY MENU

ESTAT	F1	DISPLAY CURRENT STATUS OF EMULATION MEMORY
<addr> **EMENABLE**	F2	ENABLE ONE BLOCK OF EMULATION MEMORY
ALSO <addr> **EMENABLE**	F3	ADD ANOTHER BLOCK OF MEMORY
=EMSEG	F4	SET A16-A19 MEMORY SEGMENT BITS
EMCLR	F5	DISABLE ALL EMULATION MEMORY
	F10	RETURN TO MAIN MENU

LOAD OR SAVE PROGRAM MENU

HEXLOAD	F1	LOAD INTEL HEX FILE
<from> <to> **BINLOAD**	F2	LOAD BINARY OBJECT FILE
<from> <to> **BINSAVE**	F3	SAVE A BLOCK OF MEMORY TO DISK FILE
LTARG	F4	LOAD A SAMPLE PROGRAM
	F10	RETURN TO MAIN MENU

PROM READER MENU

<from> <to> **RPROM**	F1	READ 2716/48016 - use PM16
<from> <to> **R2532**	F2	READ 2532 - use PM16
<from> <to> **R2732A**	F3	READ 2732 - use PM32
<from> <to> **RPROM**	F4	READ 2764 - use PM64
<from> <to> **RPROM**	F5	READ 27128 - use PM64 (PM56 for 27128A)
<from> <to> **RPROM**	F6	READ 27256 - use PM56
<from> <to> **R27512**	F7	READ 27512 - use PM512
	F9	Go to Prom Programmer Menu
	F10	RETURN TO MAIN MENU

See also Appendix G for more info on EPROMs.

2.2 Emulation ROM

You use the UniLab to watch the execution of a program on your microprocessor board. Microprocessors usually run a program that is loaded into ROM or RAM. While using the UniLab, you load the program into emulation ROM.

32K or more of emulation memory

The standard UniLab contains 32K bytes of 195 ns static RAM which functions as emulation ROM. An optional expansion board can expand this capacity up to 128K bytes. This RAM appears to the target system as ROM, and cannot be altered by the target microprocessor.

Cable Connections

Most of the data and address lines are connected by plugging the emulator cable into a single PROM socket in the target system, as explained in the **Installation** chapter (two sockets for a 16 bit data bus).

Many ROMs can be emulated with one connection socket, but the sockets of emulated ROM must be empty to prevent bus contention.

Using the Emulator without the Analyzer

The analyzer cable must be hooked up for the emulator to operate properly. In the unlikely event that you want to use the emulation ROM without using the analyzer, you must still connect the analyzer cable from the UniLab to your board.

The UniLab must see the full address bus to emulate properly, and some of the address signals are picked up by the analyzer cable.

20-bit addresses

Since the UniLab accepts a 20 bit address input, it can handle target systems with up to 1 megabyte of active memory, or even more if you connect chip select logic to the A19 input to the UniLab.

Emulate throughout a 128K region

Any combination of 2K byte memory segments can be enabled, as long as they are all in the same 128K region. That is, you can emulate 2K chunks scattered throughout the range C0000 through DFFFF, since that forms one 128K region.

However, you would **not** be able to emulate some memory within the range 10000 to 1FFFF and other memory in the range 30000 to 3FFFF, since those two 64K segments are not contained within the same 128K region.

The general rule-- all emulated memory must have the same value for the upper three bits of the full 20 bit address. You set this value with **=EMSEG.** You rarely need to change this value.

Watch out

Since the 32K UniLab emulates 128K of address space in only 32K of physical RAM, each physical location represents four emulation ROM addresses. This can cause problems if you are not aware of it.

For example, the four addresses 00000, 08000, 10000 and 18000 all refer to the same physical memory location.

If you try to enable both **0 TO 7FF** and **8000 TO 87FF,** then you will find that both sections of memory always contain the same data. Those two ranges of emulation ROM both refer to the same RAM locations in the 32K UniLab.

2.3 Getting Ready. . .

Before you can start working on your program, you have to enable the emulation ROM and load your software into the UniLab's emulation memory.

When you enable a section of memory, you are telling the UniLab what addresses you want it to respond to.

The minimal memory necessary

The **LTARG** command enables a 2K section of memory and loads in a simple test program (on some packages, such as the 8096, the **LTARG** command enables several 2K sections). If you are in doubt about what memory to enable for your processor, type in **LTARG**, and note the values of **=EMSEG** and **EMENABLE.**

For example, the Z80 test program sets up the variables as:

```
LTARG
 Emulator Memory Enable Status:
                    7 =EMSEG
          0 TO   7FF EMENABLE
```

In general, you have to enable the reset address for your processor. The only exception occurs when you want to analyze a program running from ROM chips instead of emulated ROM.

The exception: Running a program from ROM chip

When you want to run a program entirely from ROM on your target board you must first use **EMCLR** to clear emulation memory, and then use the Mode Panel option **SWI VECTOR** (**S**oft**W**are **I**nterrupt **VECTOR**) or the command **RSP'** to disable the DEBUG.

If you don't disable the software interrupt vector under these circumstances, then the UniLab will give you an error message when you try to start the analyzer. When the DEBUG is enabled, the UniLab writes information into the reserved area when you start the analyzer. If the reserved area is not being emulated, you will get the message "Debug Control not established."

You can use ROM chips for some of your program, and use emulation memory for the rest of it.

No matter what else you do, you must always either emulate the reserved area (see appendix H) or disable the DEBUG.

The high four address bits

Address bits A16 to A19 are set with <hex number> **=EMSEG**, where hex number is the desired digit for A19-A16. With this command you tell the UniLab which 64K segment of memory you want to emulate, out of the possible 1 megabyte that a processor with a 20 bit address bus can access.

The value of **=EMSEG** is initialized to the correct value. You will probably never have to alter it.

The UniLab uses **=EMSEG** to decide whether to put data on the bus. When the processor tries to fetch information from ROM, the UniLab first checks whether the upper four bits of its address inputs match the value of **=EMSEG.**

If the UniLab finds that the upper four bits match, then it checks whether the lower 16 bits of the address are enabled.

On many 8 bit processors these inputs just float high, so that the **=EMSEG** value is usually F (1111 binary).

Emulating two 64K segments

You can emulate address in 128K, as long as the two 64K segments are neighboring segments (that is, only A16 differs). Use **=EMSEG** to set the values of **A16** through **A19** that the UniLab's emulation ROM will respond to.

For example, 4 is 0100 in hexadecimal,
while 5 is 0101, so that you could give this enable command:

```
     4 =EMSEG    0 TO  7FF EMENABLE
ALSO 5 =EMSEG 1000 TO 17FF EMENABLE
```

The other 16 bits

You can enable the UniLab's memory 2K at a time. The memory that you enable will be in the 64K segment that you last set with **=EMSEG.**

You are telling the UniLab what range of addresses on A0 through A15 you want it to respond to.

To enable ROM from address 0 to 17FF you type:

0 TO 17FF EMENABLE

You could instead enable locations F800 to FFFF by entering:

F800 TO FFFF EMENABLE

The **TO** is necessary to indicate an address range. If you enter <16 bit address> **EMENABLE** the single 2K segment which includes that address will be enabled. For example,

1000 EMENABLE

will enable the 2K segment 0 to 17FF.

Enabling several areas

Each **EMENABLE** statement usually clears out the previous settings. However, if you use **ALSO** you can have the UniLab respond to both the previous setting and the new one.

For example, to enable 0 to 17FF and F800 to FFFF, you type:

0 TO 17FF EMENABLE ALSO F800 TO FFFF EMENABLE

Be careful when enabling several areas with a 32K UniLab. The 128K area within which you can enable 2K blocks gets mapped onto the 32K of the UniLab. This means that 0000 addresses the same memory location as 8000. 2000 refers to the same location as A000.

Keeping track

Every time you issue the **EMENABLE** command, the system will display the complete resulting memory enable status. If you want to see this enable status display without changing enables, just enter **ESTAT.**

Saving the settings

Once you have them set properly for a project, you will want to can save the enable settings of emulation memory. The contents of emulation memory can also be saved, with the **BINSAVE** command.

You enter:

SAVE-SYS <filename>

after you enable the memory to save the current state of the UniLab software, including the emulation memory settings.

From then on, when you start the program by typing in the filename, the proper area of the UniLab's memory will already be enabled.

2.4 Load the Target Programs into Memory

You load in your program after you have enabled a section of memory large enough to contain your program.

You can load opcodes into memory from a disk file, by hand, from a ROM chip or from an Orion test program.

Load from disk files

The UniLab software provides you with four different ways to load a program from a file on disk. Depending on your assembler or compiler, you will chose one of these methods.

1. If you compiled or assembled the code into a binary file on a disk, then load it with

<from addr> <to addr> **BINLOAD** <filename>

The filenames usually end in **.BIN**, **.COM**, or **.TSK**. You will be prompted for the file name if you do not include it on the command line.

The program will be loaded starting at the address you gave.

You save memory to a file with the **BINSAVE** command.

2. Read Intel-format HEX object files from a disk with

HEXLOAD <file name>.

You will be prompted for a file name if you do not include it on the command line. The addresses will automatically be converted to the correct ones for the host image of the target program. This method is much slower than **BINLOAD.**

If your assembler will only make Intel Hex files, you can still use the UniLab command **BINSAVE** to make a binary format file. Just load the hex file the first time, and then use

<from> <to> **BINSAVE** <filename>

to save the memory as a binary image. From then on you can use

<from> <to> **BINLOAD** <filename>

to load the program into memory.

3. Download Intel hex format programs from another computer system with **HEXRCV**, if your PC has two serial ports. The sending computer must support the XON/XOFF protocol.

After you type this command, your PC will accept hex code through its second port until you press a key or the PC receives an end of file message.

While this method is useful for interfacing with existing systems, it makes more sense to use your personal computer for program development and avoid the bottleneck of program downloading. See Appendix B for a partial list of assemblers and compilers for the personal computer.

4. If your assembler or compiler can assemble directly into memory at a specified location other than the origin, you can instruct it to leave the object code in some unused area of host memory (C000 to E000 is free in most systems).

Then when you enter the UniLab program you can download from your host's memory to the UniLab's emulation ROM with

‹fromadr› ‹toadr› ‹targadr› **MLOADN.**

Note that the first two addresses refer to RAM in your host machine, the third address is in UniLab emulation ROM.

Hand enter Code

You can also hand enter a program, poking machine language instructions into memory. We recommend this only to those suffering from computer nostalgia.

You hand-enter a short program by using the memory patching commands of the UniLab system. Type in ‹address› **ORG**, where the address is the start of the target program, then enter ‹byte› **M** or ‹word› **MM** for each byte or word of the program. See Section Three of this chapter for more ways to alter and examine emulation memory.

Read a program from ROM

The UniLab software also allows you to read a program from ROM. We support all of the most popular EPROMs-- see Appendix G: EPROMs Supported.

Read a program from a ROM by first placing the chip into the UniLab's programming socket. Hit function key 10 to get the main menu, and then function key 9 to get the PROM reader menu.

picture of chip in a socket....

The PROM READER menu tells you which personality module to use:

PROM READER MENU

```
 F1  READ 2716/48016 - use PM16
 F2  READ 2532       - use PM16
 F3  READ 2732       - use PM32
 F4  READ 2764       - use PM64
 F5  READ 27128      - use PM64 (PM56 for 27128A)
 F6  READ 27256      - use PM56
 F7  READ 27512      - use PM512
 F9  Go to Prom Programmer Menu
F10  RETURN TO MAIN MENU
```

Avoid leaving any PROM in the socket after you read it or program it.

2764s and up will sometimes erase location zero when you turn on the UniLab.

Load the sample program

You can also load a simple test program, with **LTARG** (Load **TARG**et memory).

LTARG enables emulation memory and loads a simple test program. The Target Application Note for your Disassembler/DEBUG software shows trace and breakpoint displays generated with the **LTARG** test program.

2.5 Saving Programs

You can save a program for later use with **BINSAVE** as described below. There are at least five situations in which you will want to save a program:

1) You have changed the program since you loaded it in, by moving sections of memory or poking in an opcode.
2) You have loaded a program using **HEXLOAD**, and want to be able to use **BINLOAD** instead.
3) You have loaded a program from a ROM.
4) You have "hand assembled" a program.
5) You want to make a macro that tests equipment by loading and running a test program.

When you have completed your design, you can "save" a program to ROM with the PROM programmer menu. See section 7.

Saving with BINSAVE

Any area of emulation memory can be saved to disk as a named file. Type in

‹from address› ‹to address› **BINSAVE** ‹file name›.

If you leave off the filename, you will be prompted for it.

3. Examine and Alter Memory

Introduction

After you load a program into emulation memory, you can immediately run it. However, you often want to look at the program first, to refresh your understanding of the code, or to verify that you have loaded in the correct file.

And as you work on the program, you will want to look at portions of your code, and perhaps alter it. You will often want to disassemble from memory to help you decide where to set triggers or breakpoints.

You can also examine and alter RAM (see subsection 3.2 and section 6).

Contents

3.1 Feature Summary

All memory access commands work both on emulated ROM and on target RAM. However, you need DEBUG control to access RAM. See section 3.2.

Summary of Most Common Commands

Feature	Menu	Command
Display and modify memory	Yes	<addr> **MODIFY**
Disassemble from Memory		
Disassemble a range of memory	Yes	<start> <# of lines> **DM**
Disassemble into right hand window	NO	<start> **DN**
Assemble		
Line-by-line assembler	NO	<addr> **ASM**
Assemble code from FORTH file	NO	<addr> <from scr> <to scr> **ASM-FILE**
Block-oriented commands		
Fill a range of memory with one byte value	Yes	<start> <end> <byte> **MFILL**
Move a range of memory to a different place	Yes	<start> <end> <new start> **MMOVE**
Compare two ranges of memory	Yes	<start> <end> <comparison addr> **MCOMP**
Dump a range of memory	Yes	<start> <end addr> **MDUMP**
Set context		
Addresses refer to RAM	NO	**TRAM**
Addresses refer to ROM (default)	NO	**TRAM'**

Byte and Word Oriented Memory Access Commands

Since the **MODIFY** command makes it very easy to alter memory while you examine it, you will rarely have to use the commands on this page. They will be most useful within macros, where **MODIFY** is inappropriate.

Feature	Menu	Command
Display Memory		
Look at one byte	NO	<addr> **M?**
Look at one word	NO	<addr> **MM?**
Alter Memory		
Alter a single byte	Yes	<value> <addr> **M!**
Alter a single word	Yes	<value> <addr> **MM!**
Set up the address for subsequent **M** and **MM** commands	NO	<addr> **ORG**
Store one byte and update **ORG**	NO	<byte> **M**
Store a word and update **ORG**	NO	<word> **MM**

Memory Access Menu

Command:	**Menu:**	
		EXAMINE OR CHANGE PROGRAM MEMORY MENU
MODIFY	F1	EXAMINE MEMORY, ALTER IF DESIRED
DM	F2	DISASSEMBLE FROM MEMORY
M!	F3	CHANGE ONE BYTE
MM!	F4	CHANGE ONE WORD
MFILL	F5	FILL A RANGE OF MEMORY WITH ONE VALUE
MMOVE	F6	MOVE AN AREA OF MEMORY
MCOMP	F7	COMPARE TWO AREAS OF MEMORY
MDUMP	F8	DUMP MEMORY
	F10	RETURN TO MAIN MENU

3.2 Memory Access Complications

Three factors can cause complications when you access memory with UniLab commands:

1) When you access emulation ROM, you will cause the program to stop.
2) You cannot access RAM unless you have establish DEBUG control.
3) Some DDB packages require something more than a 16-bit address to distinguish between RAM and ROM. With these software packages you will need to switch context back and forth with two context setting words, **TRAM** (for Target RAM) and **TRAM'** (for ROM).

Access to emulation ROM

When you read from or change emulation ROM, the UniLab has to take control of the memory chips that emulate ROM. Unless, you first

establish DEBUG control, or
use the command **RES-**,

your access to emulation ROM will cause your target program to crash. These two approaches are explained on the following page.

Emulation ROM not available to target system

Reading and writing emulation memory with UniLab commands has the same effect as removing the ROM chips from the target system circuitry. So it is not surprising that the target system stops executing the program-- when it tries to fetch a byte of opcode, your processor will see FFs (all the data lines float high).

Generally, after you read or write emulation ROM, you must restart the target program.

Sometimes, when the target processor gets bad data from emulation memory, it will write into various addresses. These random writes can overwrite battery backup RAM or send instructions to port addresses that control peripheral devices.

Crash free access to emulation ROM ...

... with DEBUG control

While you have DEBUG control, your processor does not attempt to access ROM. It is held in an "idle loop" by the UniLab hardware.

This approach will work with any DDB software package. See the DEBUG Control section of this chapter to learn more about how to establish DEBUG control.

... with RES-

Alternatively, you can use the command **RES-** to pull low and hold low the output of the UniLab which resets the target processor (labeled RES-). This will, with some target systems, hold the target processor in a reset state. In this state the target system does not execute any code, so you can read and write to emulation memory without causing any harm.

The RES- output will stay low until you start the analyzer.

Of course, when you do start the analyzer, your target system will start to execute the program from the reset vector address, regardless of whether you have reset enabled or disabled.

RES- will not work if your target system has a "one-shot" in the reset circuit.

Access to RAM

All the commands that read and write memory can perform their work on any memory chips in your target system, as well as on emulation ROM.

You can examine or alter target RAM (or non-emulated ROM) with the DEBUG features of the UniLab. You establish DEBUG control with commands detailed in section 6 of this chapter.

If you do not have DEBUG control when you attempt to access target system memory, the UniLab can establish it for you, as detailed below.

Access to RAM with automatic DEBUG control

If you try to read or write RAM when you do not have DEBUG control, the UniLab program will try to establish debug control for you, by issuing an NMI (Non-Maskable interrupt) signal. You will get two messages:

1) a "not enabled" message, informing you that you are trying to access an address that the UniLab is not emulating,
2) an "-nmi-," to let you know that the UniLab is trying to gain DEBUG control.

The UniLab will then perform the action you requested, and afterward allow your target program to resume execution.

If the attempt to gain DEBUG control fails, the UniLab will either wait for you to press the carriage return, or give you the message "Debug Control not established."

You will find additional information on DEBUG control and the NMI feature in section 6 of this chapter.

Access to RAM with manual DEBUG control

You will usually first gain DEBUG control yourself and then perform the access to RAM.

You will still get the "not enabled" message, to remind you that you are working on RAM, not on emulated ROM.

Distinguish between RAM and ROM

Many processors require more than a 16-bit address to uniquely identify a memory address. Some of these have RAM and ROM mapped onto the same memory space, while a growing number of processors use a 20 or 24 bit address to address memory.

In either case, the same 16-bit physical address can be used to refer to either a RAM address or a ROM address.

With these processors the UniLab will, as a default, assume that you are trying to access ROM. When you want to access RAM, you will have to issue the command **TRAM** (Target **RAM**) before giving a memory command.

After you are done with the RAM, you will want to issue the command **TRAM'** to toggle back to working with emulation ROM. You might run into problems with DEBUG control and with starting the analyzer if you neglect to do this.

Processors with RAM and ROM in the Same Address Space

Some processors allow you to have RAM and ROM at the same addresses at the same time, such as:

the **8051** family,
the **Z8** family,
and the **64180.**

Processors that can address more than 64K of memory

A different set of processors, including the **68000** and the **8088/86** family, can address more than 64K of memory.

3.3 Memory Display and Modify

The new, screen oriented memory access command, **MODIFY**, will make it easier for you to poke bytes into program and data memory locations. This new feature replaces several of the older memory commands.

MODIFY and **DM** give you more than enough power for almost all of your memory display needs. And the combination of **MODIFY** and **ASM** will be enough to satisfy most of your memory modification needs. Other UniLab memory commands are available to do the few jobs for which **MODIFY, ASM,** and **DM** are inappropriate.

Invoke Memory Modify

MODIFY uses most of the current window to display up to 100 hex bytes at a time (256 decimal). The window size determines how much you see.

You invoke the command with a single parameter-- the address from which to start:

‹addr› **MODIFY**

The **MODIFY** display will then fill up the window. When you exit normally from **MODIFY**, the previous window will be restored.

Examine Memory with Modify

Once you are within **MODIFY**, the cursor keys will be reassigned, as indicated by the prompt line.

The **Page Up** and **Page Dn** keys bring new blocks of memory into the modify screen. Any changes that you make to the old block are saved before the new one is brought in.

You use the up and down arrows both to move around in the the screen and to scroll memory onto the display screen 10 (hex) bytes at a time.

Alter Memory with Modify

You can use **MODIFY** to alter the value stored in any memory address. Use the arrow keys to place the cursor on either the hexadecimal digits or the ascii character, then type in the new value.

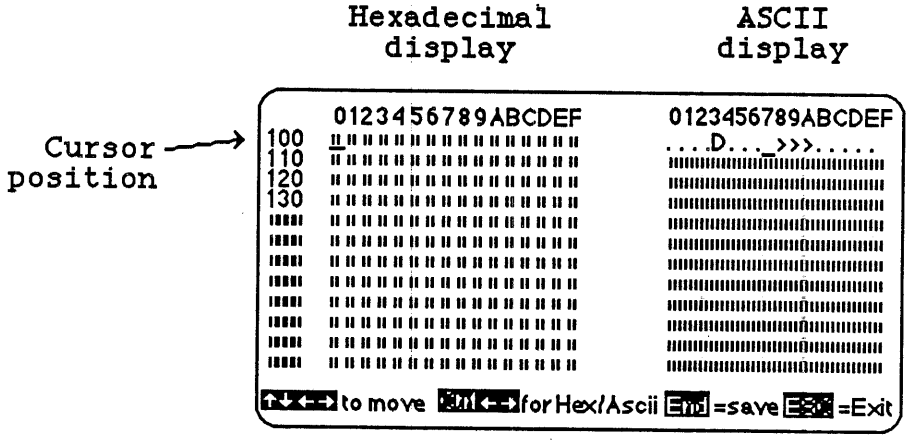
The new value will be saved if you exit with the **END** key, or if you use the cursor keys to scroll the altered location off the screen. You can exit without saving your most recent changes by pressing the **ESC** key instead.

As noted in the prompt line, you move the cursor from the hexadecimal dump to the ASCII representation by holding down **CTRL** and the **Right Arrow** keys at the same time. **CTRL-Left Arrow** moves the cursor back to the hexadecimal dump area.

Non-emulated Memory and MODIFY

You examine and alter target system RAM with **MODIFY**, but only if you are able to achieve DEBUG control. (See subsection 3.2 and section 6 of this chapter).

You will be dumped out of **MODIFY** with an error message if you attempt to alter memory and the UniLab is unable to achieve DEBUG control.



3.4 Disassemble from memory

Two commands allow you to disassemble from memory:

<addr> <# of lines> **DM**
<addr> **DN**

Both commands need to know what address you want to start disassembly from.

DM also has a second parameter-- the number of lines of disassembled code to display.

DN writes the disassembled instructions into a dedicated window on the right hand side of the screen. It always disassembles enough lines to fill the window, and so does not need a second parameter.

Specify address that points to start of instruction

The disassembler will try to interpret the hexadecimal code at the address you specify as the first byte of an opcode. If that address is not the first byte of an instruction, you might see some incorrectly decoded instructions.

Once the disassembler gets back "in sync," it will decode properly.

Example: Good disassembly

```
                    0 7 DM
                    0000  310019   LD SP,1900
                    0003  3E12     LD A,12
                    0005  015634   LD BC,3456
                    0008  119A78   LD DE,789A
                    000B  21DEBC   LD HL,BCDE
                    000E  C5       PUSH BC
                    000F  C1       POP BC
```

Example: "Out of Sync" disassembly

```
                    1 7 DM
                    0001  00       NOP
                    0002  19       ADD HL,DE
Back in sync --->   0003  3E12     LD A,12
                    0005  015634   LD BC,3456
                    0008  119A78   LD DE,789A
                    000B  21DEBC   LD HL,BCDE
                    000E  C5       PUSH BC
                    000F  C1       POP BC
```

3.5 Line-by-line Assembler

The processor specific line-by-line assemblers, **ASM** and **ASM-FILE**, allow you to write assembly language patches to your target program from within the UniLab program.

To invoke the assembler when you want to enter assembly language instructions from the keyboard use

<addr> **ASM.**

You can also invoke the assembler on code that has already been written into a FORTH file, with

<addr> <from screen #> <to screen #> **ASM-FILE.**

Assembler replaces previous opcodes

Both commands process assembly code instructions and write machine language codes into memory. You overwrite-- and therefore lose-- the data already in memory.

You can keep a record of the changes you have made by turning on the LOG TO PRINT option in the mode panel **(F8)**.

Choose the starting address

If you do not include the address, the assembler will start from wherever the assembler last left off (or, if assembler has not been used yet, the last address stored by the **ORG** command).

Conventions

The line-by-line assembler will only accept assembly language instructions, not ORIGIN statements or EQU statements. (You can use the UniLab command **IS** to define symbols.)

Only one instruction per line.

A complete list of assembly language commands accepted by the assembler appears in the Instruction Set section of the Target Application Note for your DDB package.

The normal conventions of assembly language apply. For example, there must be at least one space between an instruction and its operands.

Use of ASM

When you use **ASM** you can include an assembly language instruction on the command line, and assemble only that one instruction:

```
1200 ASM  INC A
```

You can enter multiple lines if you do not include an assembly language instruction on the command line:

```
1100 ASM
```

ASM will give you as a prompt the address to which it is assembling, and wait for you to give it an instruction followed by a carriage return.

The assembler will prompt you with a new address each time you enter an instruction (and put assembled code into memory) until you feed a blank line (press return without entering an instruction).

If you enter an instruction that is not recognized by the assembler, it will type an error message and then prompt you with the same address again.

Use of ASM-FILE

Enter instructions from the MEMO pad section of FORTH file

If you only have a few lines of code, you can use the screen that **MEMO** puts you into, and the two following (screens 1D through 1F). See the command reference entry for **MEMO** to get a few pointers on using the FORTH screen editor. You might also want to look at Appendix F.

Put code in its own FORTH file

You will want to put the code into a file of its own if you have many lines of code, or if you want a more convenient way to archive the code. First, close the current file **(UniLab.SCR)** with the command:

CLOSE

Then create a new file with:

OPEN-NEW <file name>

and give it a size with:

<# of screens> **SCREENS**

1K is allocated per screen. Always specify at least 2 screens (numbered 0 and 1). NEVER try to assemble from screen zero. Use the command:

<screen #> **EDIT**

to get into the file.

Only use **OPEN-NEW** when you want to create a new, blank file. After that, when you open the file, use the command:

OPEN <file name>.

Assemble code from FORTH screens

After you write and save the assembly language code, you will use **ASM-FILE** to assemble the code stored in your new file. For example, to assemble screens one through four into emulation ROM, starting at address 1200:

1200 1 4 ASM-FILE

You can include comments on a screen by putting a semicolon (;) on a line. The assembler will ignore everything after the semicolon on that line. The semicolon must either be the first character on the line, or be preceded by at least one space.

When you are done assembling, use the command **UDL.SCR**, which closes your file and re-opens the UniLab.SCR file. If you don't do this, then some of the on-line help facilities and error messages will not work.

3.6 Variable Size Block Memory Commands

Four UniLab memory commands operate on variable sized blocks of memory-- perform on a range of memory that you specify with a lower and upper address limit, rather than on a byte, word, opcode or screenful at a time:

MFILL
MMOVE
MCOMP
MDUMP.

Common uses for block commands

You generally fill blocks only to test the target system-- put a long series of identical one-byte opcodes into memory.

You usually move blocks only for patching purpose-- push a block of memory up or down to make room for an extra instruction or series of instructions.

You will probably only compare blocks to find the differences between two versions of a program, or to compare two blocks of data memory.

There are many situations where you will want to dump out a range of memory. However, you will usually use **MODIFY** rather than **MDUMP**, since **MODIFY** allows you to alter the memory while displaying it. **MDUMP** will be most useful when you only want to display a small area of memory, or when you want to dump huge block to the printer.

Limitations on block commands

The maximum **MMOVE** block size is 32K.

For all other commands, the maximum block size is 64K.

Fill memory

You fill a range of memory with the **MFILL** command. It sets every byte in the range to the same value:

<from address> <to address> <byte value> **MFILL**

Use MFILL to test a target system

This is a handy way to test the data and address lines of your processor board:

1) Fill a range of emulation memory with the opcode for NOOP, or other simple instruction, starting at the reset address.
2) Start up the processor and capture a trace, using the command **STARTUP**.
3) Examine the trace and verify that the address lines and data lines work properly.

Example

To fill 80 bytes of emulation ROM with FA, starting at address 00:

0 100 FA MFILL

Move memory

You copy information from one range of memory to another with:

‹start address› ‹end address› ‹copy starting at address› **MMOVE**

You will want to do this to make room for extra instructions when patching code, or to move large chunks of code or data for other reasons.

Overlapping ranges

This command is smart enough to decide whether to start moving from the front or the back when moving into an overlapping range of addresses.

Example

To copy the code at 100 through 152 into 105 through 157:

100 152 105 MMOVE

To copy from 230 through 370, starting at 220:

230 370 220 MMOVE

Limitation

You cannot move more than 32K at a time.

Comparing

This command compares two ranges of memory, and reports any discrepancies it finds:

<start> <end addr> <comparison addr> **MCOMP**

MCOMP will start comparing from whatever address you specify. It compares the range that you specify.

You will find this command especially useful for comparing the contents of a PROM to the expected contents:

1) Put the expected contents in one range of memory,
2) move the actual contents to another range (with the PROM reader menu), and
3) use **MCOMP** to compare the two.

Example

Notice how, in this example, **MCOMP** starts finding bytes that don't match after comparing five of them. It would continue to compare bytes until it had compared the data at 120 (hex) to that at 820 (hex)-- or you can terminate the display by hitting any key.

```
105 120 805 MCOMP
Data is 16 at addr 0110 ..but is  5 at addr 0810
Data is 90 at addr 0112 ..but is 80 at addr 0812
Data is 27 at addr 0116 ..but is 23 at addr 0816
```

Dumping

One command allows you to see the hexadecimal contents of a range of memory:

<start> <end addr> **MDUMP**

MDUMP will start dumping from whatever address you specify, showing 10 (hex) bytes of memory on each line. It always displays a full line, so that the second address will get rounded up, if necessary. The right-hand side of the display shows what ASCII characters, if any, the hexadecimal codes correspond to.

```
0 14 MDUMP
 0   31 00 19 3E 12 01 56 34  11 9A 78 21 DE BC C5 C1   1..>..V4..x!....
10   3C 3C 3C 3C 3C 3C 3C 3C  3C 3C 3C 3C 3C 3C 3C 3C   <<<<<<<<<<<<<<<<
```

3.7 Byte and Word Oriented Memory Commands

Look at Memory

MODIFY always shows you a full screen display of memory-- so you will sometimes prefer the byte and word display commands since they display the hexadecimal number stored at a single memory address.

Write to Memory

When you want to alter a few bytes of memory you will most often use **MODIFY**-- but sometimes you will find it appropriate to use one of the byte or word write commands.

Display a byte or word

You "peek" at memory with:

<addr> **M?**
<addr> **MM?**

The first command looks at a byte, the second at a 16-bit word.

Peeking example

13 M? 3C ok

C MM? BCDE ok

Alter memory, by the byte and word

There are three ways to alter memory on a small scale:

poke bytes into specific addresses,
<byte> <addr> **M!**
poke words into specific addresses,
<word> <addr> **MM!**
set up an origin address, then store bytes and words at sequential addresses.
<addr> **ORG** <byte> **M** <word> **MM**

Poking bytes

You poke bytes into specific addresses with:

<byte> <address> **M!**

Poking words

You poke words into specific addresses with:

<word> <address> **MM!**

If you have a disassembler, then the UniLab program knows which order to store bytes into memory.

Setting up an origin and storing bytes and words

You set up the origin address with:

<address> **ORG**

and then can store bytes and words with:

<byte> **M**
and
<word> **MM**

These commands store the value and increment the "origin" address (which is also used by **ASM**).

You will want to use this method whenever you need to store several opcodes at sequential addresses, and do not want to use the screen-oriented **MODIFY**.

4. Set a Trigger (Generate a Trace)

Introduction

This section tells you how to describe a bus state with the UniLab command language. This information is also presented in the Command Section of Chapter Three.

A properly executed trigger description will capture a trace buffer which includes only the bus activity you want to see.

The power of the UniLab comes from this ability to capture and display the program activity that you specify.

Contents

4.1 Feature Summary

Pre-set Triggers

Feature	Menu	Command
Start the target program and show first cycles	Yes	**STARTUP**
Show what the program is doing right now	Yes	**NOW?**
Sample address lines, twice/second	Yes	**ADR?**
Sample all lines, once each second	Yes	**SAMP**

Tools for Building a Trigger

Feature	Menu	Command
Most commonly used trigger-- on a single address		
Clear out previous trigger spec, and set one on a single address. This command has the same effect as **NORMT** <addr> **ADR S**, but requires a lot less typing.	NO	<addr> **AS**
Clear out previous trigger spec, and setup for trigger to appear...		
... at top of trace buffer	NO	**NORMT**
... in the middle of trace buffer	NO	**NORMM**
... at bottom of trace buffer	NO	**NORMB**
Specify a subsection of target system bus		
Set trigger on (16-bit) address	Yes	**ADR**
Set trigger on (8-bit) CONTrol inputs	NO	**CONT**
Set trigger on (8-bit) data value	Yes	**DATA**
Set trigger on high byte of data	NO	**HDATA**
Set trigger on high byte of address	NO	**HADR**
Set trigger on low byte of address	NO	**LADR**
Set trigger on (8) MISCellaneous inputs	NO	**MISC**
Specify the value to search for on section of target bus		
Specify a single value for **ADR**		<16-bit value>
Specify a single value for any other input		<byte>
Set trigger on Range of values	Yes	<val1> **TO** <val2>
Invert following	Yes	**NOT**
Add following trigger to current	NO	**ALSO**
Specify masked value for 8-bit channel	NO	<mask byte> **MASK** <value>
Specify a 20 bit address	NO	<20-bit number>**. ADR**
Start searching for trigger specification		
Startup Analyzer	Yes	**S**
Startup Analyzer, capture new trace that starts where current trace ends	NO	**S+**
Reset the Target System when Analyzer starts		
Don't restart target program when Analyzer starts	Yes	**RESET**
Do restart target program when Analyzer starts	Yes	**RESET'**
Change one of two variables, to specify a delay or repetition		
Specify number of cycles to wait after trigger before freezing the trace buffer (default: zero)	NO	<number> **DCYCLES**
Specify number of repetitions of trigger event (default of one)	NO	<number> **DEVENTS**

Filtered Traces

Clear out previous trigger and produce a trace showing only bus cycles that match ...

...trigger	Yes	**ONLY**
...trigger and first cycle that follows	NO	**1AFTER**
...trigger and two cycles that follow	NO	**2AFTER**
...trigger and three cycles that follow	NO	**3AFTER**

Qualifiers

You can specify up to three qualifier bus states. These qualifiers must occur during sequential bus cycles to satisfy the qualifying sequence. The trigger event does not need to immediately follow.

Basic qualifier command

Add a qualifier to the current trigger	NO	**AFTER** <spec>

Change one of two variables, to specify a delay or repetition

Specify number of cycles to wait after qualifying sequence (default of zero)	NO	<number> **PCYCLES**
Specify number of repetitions of qualifying sequence (default of one)	NO	<number> **PEVENTS**

Advanced qualifier commands

You can also specify the number of qualifiers, then change context before trigger spec commands to set the triggers and the qualifiers.

Set between 0 and 3 qualifiers	NO	<number> **QUALIFIERS**

Change context for any trigger setting commands that follow

Change to trigger	NO	**TRIG**
Change to Qualifier One	NO	**Q1**
Change to Qualifier Two	NO	**Q2**
Change to Qualifier Three	NO	**Q3**

Command: **Menu:**

ANALYZER MENU

STARTUP	F1	RESET AND TRACE FIRST CYCLES
NOW?	F2	TRACE IMMEDIATELY
NORMT <addr> **ADR S**	F3	TRACE FROM A SPECIFIC ADDRESS
<from> <to> **CYCLES?**	F4	COUNT CYCLES BETWEEN TWO ADDRESSES
SAMP	F5	SAMPLE THE BUS CONTINUOUSLY
ADR?	F6	SAMPLE ADDRESS ACTIVITY
RESET RESET'	F7	TURN RESET OFF
	F10	RETURN TO MAIN MENU

ANALYZER TRIGGER MENU

NORMT <addr> **ADR S**	F1	TRIGGER ON AN ADDRESS
NORMT <from> **TO** <to> **ADR S**	F2	TRIGGER ON A RANGE OF ADDRESSES
NORMT <f> **TO** <t> **ADR** <byte> **DATA S**	F3	TRIGGER ON A RANGE OF ADDRESSES AND A DATA VALUE
NORMT NOT <f> **TO** <t> **ADR S**	F4	TRIGGER OUTSIDE A RANGE ADDRESSES
ONLY NOT <f> **TO** <t> **ADR AFTER** <addr> **ADR S**	F5	FILTER, EXCLUDING A RANGE OF ADDRESSES AFTER AN ADDRESS
RESET RESET'	F6	TURN RESET OFF OR ON (reset is now xxx)
	F10	RETURN TO MAIN MENU

4.2 Overview

In this section of the **Operations In Detail** chapter, all the examples show traces of a Z80 program with display of the CONT and MISC columns disabled.

The first subsection introduces triggers with a simple example. The simplest trigger, and the most commonly used, is a trigger on a program address.

The next two sections discuss two issues that arise when writing triggers:

how to clear out the trigger spec,
and whether or not to reset the target processor.

The next part covers general purpose triggers. You can trigger on sets of values and on ranges of values. You can tell the analyzer to look at the control lines, the address lines, the data lines, the miscellaneous inputs, or any combination of them.

The real-life examples show how you can put trigger specs commands together to solve specific problems.

Filtered traces, introduced in the following part, allow you to look at only the cycles that interest you. You use qualifiers to set up preconditions-- the trigger will not occur until after the preconditions are met.

4.3 A Simple Example

When you use the UniLab, you will most often want to look at a trace of the bus activity that follows a certain instruction.

For example, if you have a jump instruction at address 29 of your program and want to see the trace of what it does after it jumps type in the command:

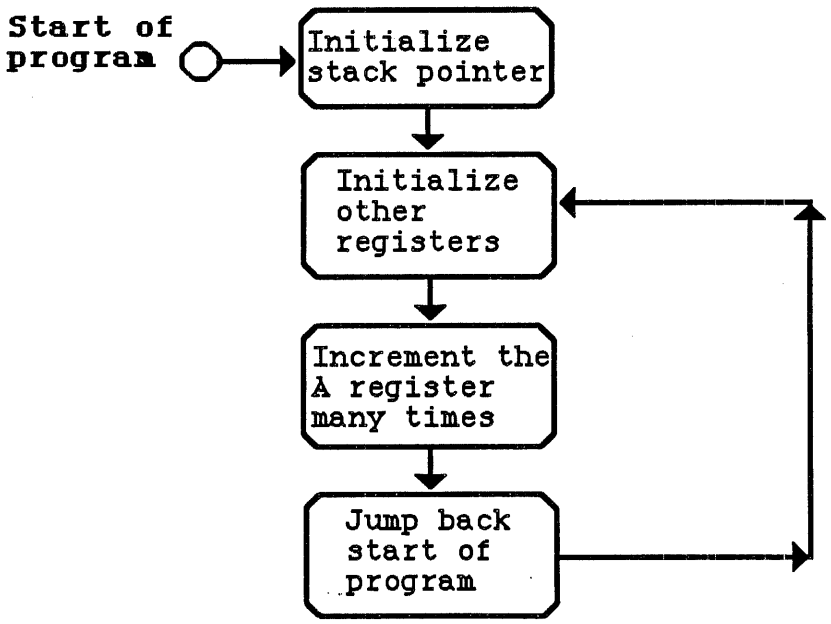
29 AS

After you hit a carriage return, the UniLab will start searching for address 510 on the target system's bus. The first time it sees that address, it will "trigger," and then freeze the trace buffer 165 bus cycles later.

While your target program continues, the UniLab sends that trace buffer to the host computer. The top of the trace fills your screen, showing the five bus cycles that preceded address 510 (labeled -5 to -1), the trigger cycle (labeled 0), and some of the cycles that follow.

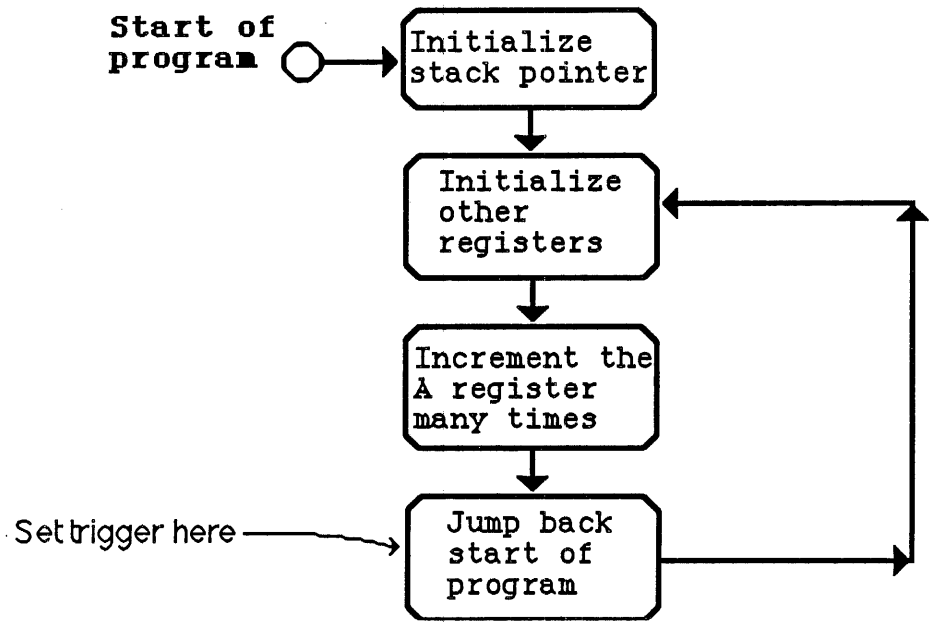
Simple Z80 example

It's easy to understand the test program loaded into the UniLab's memory by the **LTARG** command. Each DDB software package loads a different processor-specific program, but they all do about the same thing: initialize some registers, and then go into an infinite loop.



The program that you work on will, of course, be more complicated. But no matter how complicated or simple your program, you can always tell the UniLab to trigger when the some particular address appears on the bus.

There is only one mildly interesting point in the test program for the Z80. That is the unconditional jump at address 29, that jumps back to address 3.



To get a trace starting at that address, type in:

29 AS

AS will clear out any previous trigger spec, set a trigger for an address of 29, and start the analyzer.

Or, if you like to type more characters:

NORMT 29 ADR S

NORMT clears out all previous trigger specifications, and tells the UniLab that you want the trigger event at the Top of the trace.

29 ADR is the trigger specification

S starts the analyzer

Which results in the following display (with MISC, HDATA and CONT columns removed for the sake of simplicity):

```
 resetting

 cy# ADR DATA
  -5 0024 3C        INC A
  -4 0025 3C        INC A
  -3 0026 3C        INC A
  -2 0027 3C        INC A
  -1 0028 3C        INC A
   0 0029 C30300    JP 3      <----<---- Here is the trigger
   3 0003 3E12      LD A,12
   5 0005 015634    LD BC,3456
   8 0008 119A78    LD DE,789A
   B 000B 21DEBC    LD HL,BCDE
   E 000E C5        PUSH BC
   F 18FF 34 write
  10 18FE 56 write
  11 000F C1        POP BC
  12 18FE 56 read
  13 18FF 34 read
  14 0010 3C        INC A
  15 0011 3C        INC A
  16 0012 3C        INC A
  17 0013 3C        INC A
  18 0014 3C        INC A
```

Cycle numbers

The analyzer found the trigger event, and then sent to the host computer a record of bus activity starting nine cycles before the trigger. The trigger event is labeled as cycle 0, the cycles before it have negative numbers.

Explanation

The rest of the trace is fairly simple-- and very similar to the display that results from **STARTUP** with the Z80 test program.

There are only two mysteries to clear up, before continuing the discussion of trigger specifications:

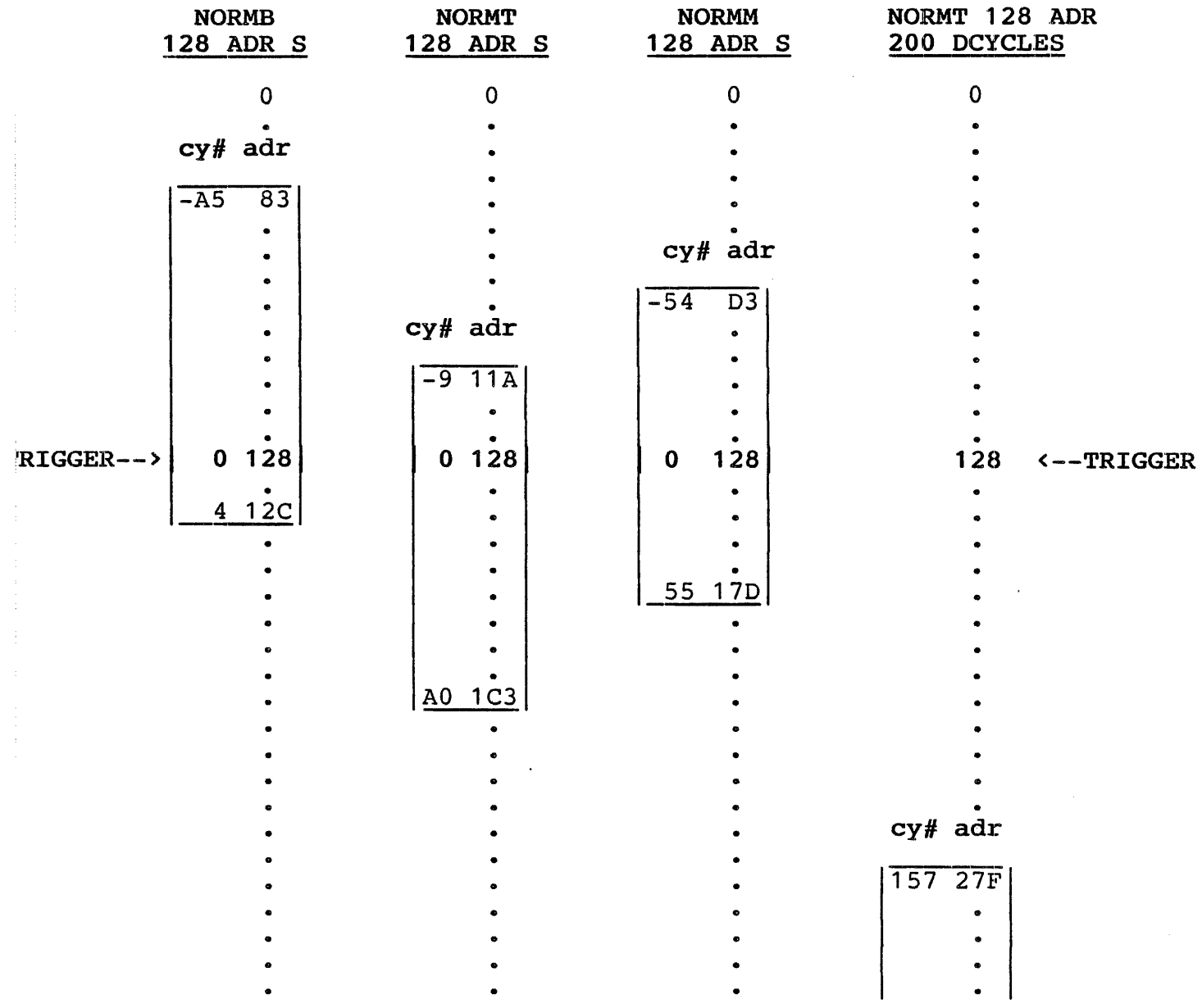
1) why **NORMT** is important,

2) the meaning of the "resetting" message that appears just before the trigger display

4.4 The NORMx Words

The three **NORMx** commands, **NORMT**, **NORMM**, and **NORMB**, first clear out all previous trigger settings. They wipe the slate clean.

And then each one sets up the "display window" to show a different time portion of the program's execution. The diagram below shows the effect of each instruction on a program that, rather boringly, executes instructions starting at address 0 without any jumps or calls or branches:



All four trigger specs are watching for the same event, and all of them number the trigger cycle as cycle zero.

They vary only in the value of **DCYCLES.** The Delay **CYCLES** is the number of cycles that the UniLab will let pass between when it sees the trigger and when the buffer is frozen. If this value is small (as happens when you use **NORMB**), then most of the trace buffer will show what happened before the trigger.

The fourth example shows how you can manually set the delay count. Here the delay is so large that the trigger is not even in the window. This example uses the **NORMT** command to clear out the previous trigger spec, but then uses **DCYCLES** to change the delay count.

Notice how the **NORMx** commands change the value of **DCYCLES** in the following example:

```
NORMT
TSTAT
 Analyzer Trigger Status:
RESET
 A0 DCYCLES   0 QUALIFIERS

NORMM
TSTAT
 Analyzer Trigger Status:
RESET
 55 DCYCLES   0 QUALIFIERS

NORMB
TSTAT
 Analyzer Trigger Status:
RESET
 4 DCYCLES   0 QUALIFIERS
```

Summary

The first address you see on the trace display after you start the analyzer with the S command depends on two things:

1) The trigger address you selected with **ADR**

2) The delay you selected with **NORMT, NORMM, NORMB,** or **DCYCLES.**

4.5 RESETting-- Restarting the target program

Whenever you start the analyzer with **S** or **AS**, the analyzer will either watch the program in progress or will start the program over. In other words, you can choose whether or not to reset the target processor.

The UniLab software gives the message "resetting" when you start the analyzer with reset enabled. This message lets you know that the UniLab has sent a reset signal to your processor.

Whenever you start up the analyzer with **S**, the UniLab will either

start watching the bus for the trigger spec **RESET'**

OR

send a reset strobe, then start watching the bus. **RESET**

Enable and disable reset

You can enable and disable resetting of the target processor with the commands **RESET** and **RESET'**, or with the mode panel (function key 8).

Mode Panel:

```
1. ANALYZER modes
DISASSEMBLER on
SYMBOLS      off
RESET        enabled
```

Other commands that affect the state of reset

RESET is also enabled by **STARTUP**, while **RB** disables it.

4.6 General Purpose Trigger Definitions

While the previous examples were limited to address triggers for simplicity, the UniLab allows much more complex triggers to be defined, using all 48 analyzer inputs.

Each of the groupings of inputs can be referred to using the same descriptive name used to label it on the trace display:

CONT **ADR** **DATA** **HDATA** **MISC**

Each of these names labels one byte of the inputs into the UniLab, except for **ADR**, which labels 2 bytes. **LADR** and **HADR** each label one byte of the address inputs.

Just as we used

‹16 bit value› **ADR**

to define a single address trigger, we can define triggers for the other input bytes:

‹8 bit value› **CONT**	to trigger on cycle type and on A19-A16.
‹8 bit value› **DATA**	to trigger on the data byte.
‹8 bit value› **HDATA**	to trigger on the upper byte of data on 16-bit processors, or on anything you like with an 8-bit processor.
‹8 bit value› **MISC**	to trigger on whatever signals you connect the miscellaneous lines to (usually target system inputs and outputs).

Modifying the input group words

All of the input group words can be altered in several different ways, by preceding them with keywords. You can also combine several input group words, as detailed on the next page.

Trigger on a single value

enter a single number to trigger on a single value,

12 DATA

Trigger on any collection of values

Use **ALSO** to add to the trigger values for a given set of inputs. **12 DATA ALSO 19 DATA**

You can use **ALSO** a number of times, to add a number of items to a set. **12 DATA ALSO 19 DATA ALSO F0 DATA ALSO E0 DATA**

Trigger on a range of values

Enter a range separated by **TO**, to trigger on a range of values. **12 TO 34 DATA**

Trigger on anything except a value

Enter the command **NOT** to trigger on anything but the value that follows. **NOT 10 DATA**

Trigger on any thing outside a range of values

Use both **NOT** and **TO** to trigger outside of a range of values

NOT 10 TO 13 DATA

Trigger on a subset of the lines in an input group

Use the **MASK** command to ignore certain input lines while watching for a given state on the other lines. The following is identical to **10 TO 13 DATA:**

FC MASK 10 DATA

since it triggers when data bit 4 is set to 1 and the all the other data bits except 0 and 1 are reset to 0-- in other words, on the values 10, 11, 12 and 13.

Or, in binary:

B# 11111100 MASK B# 00010000 DATA

Which tells the UniLab that we are only interested in the values of the first six wires

1111 1100

and that on those wires we want to see the signals

0001 00

Since we don't care what the lowest two bits are, they can be any value-- 00 or 01 or 10 or 11.

In other words, we will trigger on the values

0001 0000,	10
0001 0001,	11
0001 0010 and	12
0001 0011.	13

A Note on scope

These three keywords, **NOT**, **TO**, and **MASK**, only affect the first input group word which follows.

For example:

NORMT NOT 12 DATA 400 ADR S

will trigger when the data is <u>not</u> 12 and the address <u>is</u> 400.

Triggering on combinations

You can set a trigger on several different input bytes, and the UniLab will search for a bus cycle that satisfies all the conditions you describe. If you want to search for 12 on the data lines AND for 100 on the address lines, all you have to do is type in the command:

12 DATA 100 ADR

More on ALSO

However, when you declare a trigger for a group, you automatically clear out the previous setting for that group, unless you use **ALSO.**

If you want to search for 12 on the data lines or 15 on the data lines, you have to use **ALSO:**

12 DATA ALSO 15 DATA

which tells the UniLab to trigger on either 12 or 15.

If you had just entered

12 DATA 15 DATA

then the UniLab would watch the data lines for only one value--15, the last number specified.

When you make a new description for an input group without **ALSO**, you clear out the previous trigger for that group, without affecting the other groups. For example, if you enter

NORM 123 ADR 45 DATA S

trigger will occur only when the data is 45 during a bus cycle with 123 address. If you then enter

60 TO 71 DATA S

the analyzer will restart and trigger will occur when data is between 60 and 71 during a bus cycle with 123 address. You have altered the **DATA** specification, but not the **ADR** spec.

4.7 Real-life Examples:

Catching the program when it goes outside of program memory

One of the nastiest problems you encounter while checking out hardware or software is when your program "blows up" and begins executing data or garbage.

These errors not only are troublesome to recover from, but the mistake that caused the blow up is almost impossible to find-- until now. Trapping these problems is a pleasure with the UniLab.

If, for example, your program is supposed to be limited to addresses 0 to 1234, you can enter

RESET NORMB NOT 0 TO 1234 ADR S

The UniLab will reset the target system and wait for the target program to access an address outside of the specified range. You can then look back through the trace memory for the abnormal operation which caused the program to "blow up."

Whether it is a hardware malfunction or a software bug you will have trapped it effortlessly.

You can add **FETCH** to the above example, so that the UniLab doesn't trigger on reads and writes outside of memory. Some processors lack the fetch indicator.

Catching garbage values being written to a single memory location

Another common bug you encounter is when some location in RAM gets accidentally overwritten.

For example, a variable called STRING_LEN gets written with the length of a string. But when your program reads the value, it isn't the same as the value written into it.

One way to catch this bug is to produce a filtered trace that shows every access to this variable, and the cycles that immediately follow the access. You can then examine the trace, and find the region of the program that causes the overwrite:

2AFTER STRING_LEN ADR S

You can then trigger on the address of the bad instruction:

NORMM <address> **ADR S**

Catching a stack overflow

You can set the UniLab to trigger when your stack grows too large.

For example, a target board has ROM at locations 0 to 1FFF, and RAM at 2000 to 3FFF. The program sets the stack pointer to address 20FF in RAM. This means the stack can grow to FF bytes before running into ROM.

You can tell the UniLab to trigger when the program makes reference to some address that the stack will write to when it grows "too large"-- whatever too large means to you.

Some might want to wait until the stack is about to run into hardware limitations:

NORMB 2001 ADR S

Others will want to trigger when the stack holds more than 2F bytes:

NORMB 20D0 ADR S

Either way, you get to see what the program was doing just before the stack grew too large.

Catching bad data going into a string

You can use a combination of a range of data and a range of addresses to catch the trace of a bug that causes an inappropriate character to be written into a string.

Of course, you don't want to look at every access to the memory locations-- you just want to see when the bad data comes in.

Suppose the string sits at a location with the symbolic name STRING1 and has a length of 50 (hex) characters. The string should only contain characters between A (41 hex) and z (7A hex).

The instruction:

NORMM NOT 41 TO 7A DATA STRING1 TO STRING1 4F + ADR S

will cause the UniLab to trigger when any data outside the range 41 to 7A gets written to any of the 50 data locations starting at STRING1.

4.8 The Limits of Trigger Complexity

Since the UniLab trigger logic uses high speed truth tables instead of comparators, there is no limit to the complexity of triggers within byte groups. For example:

12 DATA ALSO 34 DATA ALSO C0 TO C5 DATA ALSO FF DATA

is perfectly acceptable.

Another way to state the same thing is by entering:

12 34 C0 C1 C2 C3 C4 C5 FF 9 NDATA

Note that the 9 is the number of terms listed.

ALSO with ADR

You can run into problems with **ADR**, since that word actually describes two bytes. If the high byte of several addresses that you are using **ALSO** on don't match, you can produce unanticipated cross products. For example:

1200 ADR ALSO 1535 ADR

would cause the UniLab to trigger on either **1200** or 1535-- and also on either **12**35 or 1**500**

These cross products usually are not a problem, but you should be aware of them.

MASKing

You can also specify triggers with a MASK format. For example,

80 MASK 0 DATA

requires the MSBit of the data bus to be 0, but doesn't care about the other 7 bits. It is identical to entering **0 TO 7F DATA** or **NOT 80 TO FF DATA.** All three commands give the same result so you should simply use the format that seems most natural to you.

Triggering on 20-bit addresses

If your system uses more than 16 bits for addressing, you can set triggers on 5-digit hex addresses by ending the address with a period. For example, **12345. ADR** will actually set a trigger on 1 in the right digit of the CONT column (which is connected to address bits A16-A19) and 2345 on the ADR inputs.

4.9 Filtered Traces

The UniLab's trace buffer stores 170 48-bit samples of bus activity. Other analyzers need gigantic trace buffers because they lack the sophisticated triggering and filtering logic of the UniLab.

Often the majority of bus cycles are not of interest-- for example when most of the time is spent in a status loop or a delay loop.

The sledgehammer solution: have a huge trace buffer. Then you get to look through that buffer, hunting for the relevant information.

The UniLab approach: have the computer throw away the boring parts of the program.

With the UniLab you never have to look through thousands of uninteresting cycles. The UniLab will filter the trace, and record only the cycles that interest you.

An introduction to ONLY

If you enter:

ONLY 1234 ADR S

the UniLab will record only cycles that address location 1234. If the instruction at 1234 is the one that reads input samples, you will end up with a trace recording of nothing but input samples. Note that **ONLY** clears out the previous trigger spec-- much like **NORMB**.

Filter, excluding addresses

More practically, suppose that a boring status loop occupies program memory from 1020 to 1060. You want to get a trace that does not include the trace of the opcodes in those addresses. The command is:

ONLY NOT 1020 TO 1060 ADR S

Filter the trace, but don't start until AFTER . . .

You can make a filtered trace even more useful by setting up a separate trigger that tells the UniLab when to start checking cycles against the filter specification. For example, the program might not get interesting until after 30 gets written to address 3000 of RAM:

ONLY NOT 1020 TO 1060 ADR **AFTER 30 DATA 3000 ADR** S

Further discussion of **AFTER** is deferred to the following subsection 4.10 on **Qualifying Events.**

The rest of the filter commands

ONLY is most useful when you want to exclude some type of operation or some section of the program.

But when you filter to include cycles, you usually want to see at least one cycle after the trigger.

For example, if you are looking at all the writes to RAM, you can find out which section of the program performed the write with

3AFTER WRITE S

which will show you every write along with the three cycles that follow it.

2AFTER captures the two cycles that follow each trigger, and **1AFTER** captures only one cycle after each trigger event. All of these words clear out the previous trigger spec and then set up for a filtered trigger spec.

Filtering and disassembly

Since filtering will produce a trace with partial opcodes, the disassembler will not be able to interpret the sequence of cycles properly. You will probably want to turn off the disassembler when producing a filtered trace. Use **DASM'** or the mode panel **(F8)**.

Mode Panel:

1. ANALYZER modes
DISASSEMBLER on
SYMBOLS off
RESET enabled

Filtering and the MISC inputs

The filtering logic of the UniLab does not look at the MISC inputs. This lets you tell the UniLab to filter a trace while waiting for a trigger condition to appear on the MISC inputs.

This is not the same as using **AFTER** with the filter commands-- with **AFTER** you get a filtered trace starting at some bus event. With the use of the MISC lines, you can get a trace that shows the bus activity before some event.

For example, if you want a trace with the delay subroutine at A0-B0 removed, but you want to trigger on an active high error signal, you connect the error signal to one of the MISC inputs and enter:

ONLY NOT A0 TO B0 ADR FF MISC

The filtered trace will exclude cycles accessing addresses A0 to B0, but trigger will not occur until the error input goes true, thus causing FF on the MISC inputs.

4.10 Qualifying Events

The UniLab can trigger on sequences of events, instead of just when it sees a single trigger event. For example,

NORMT 78 DATA AFTER 56 DATA S

will not trigger until first 56 appears on the data bus and then, anytime later, 78 appears.

The 56 is the qualifier, and 78 the trigger.

Up to three qualifiers

You can specify up to three sequential qualifying events. Use **AFTER** when you want to start the description of the next qualifier. For example:

NORMT 10 DATA AFTER 250 ADR AFTER 300 ADR S

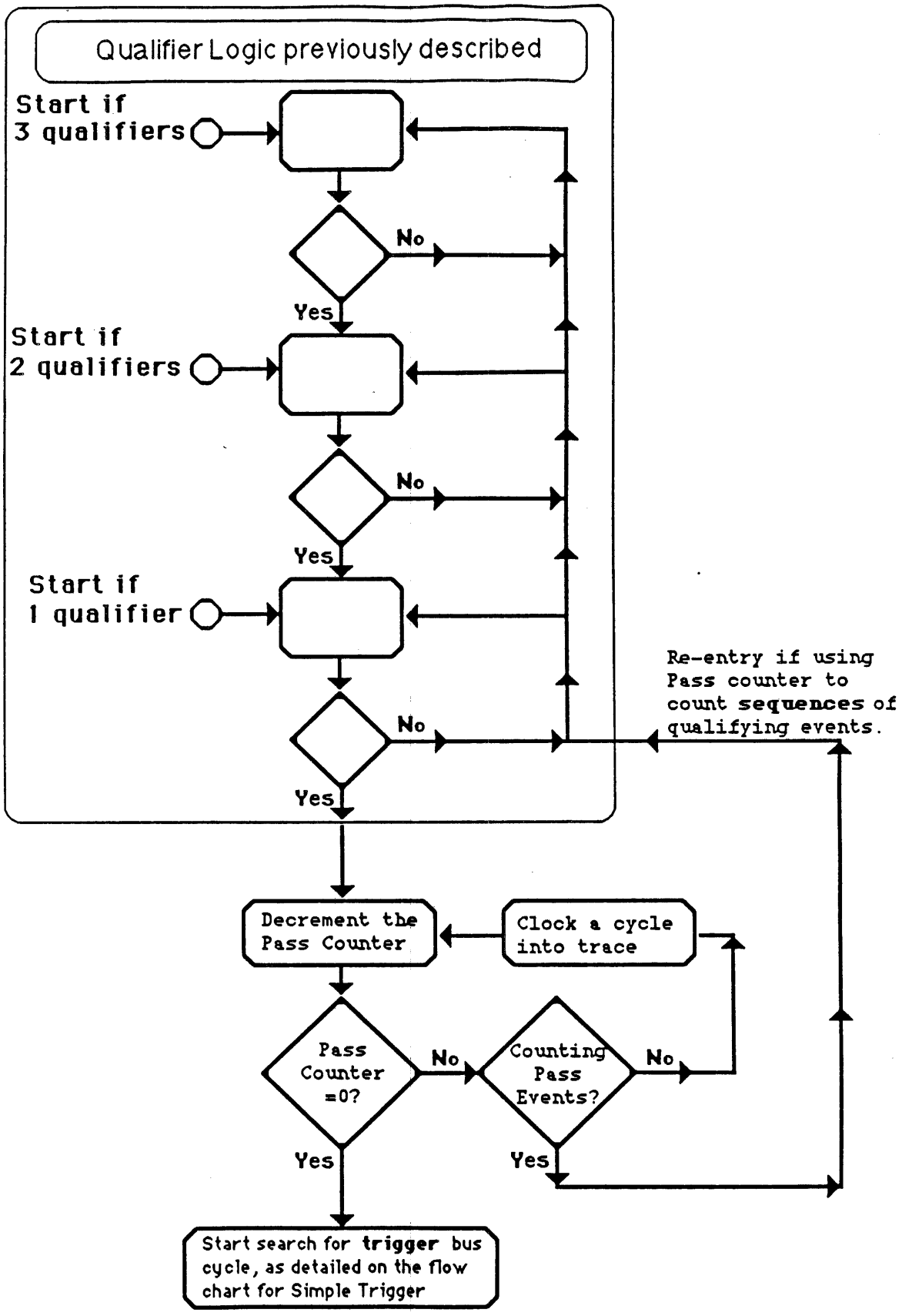
will trigger on 10 data, anytime after 300 is immediately followed by 250 on the address bus.

The UniLab will not start to search for the trigger itself until after it sees the qualifiers.

The qualifiers must appear on the bus without any intervening bus cycles. If the sequence does not appear, then the UniLab goes back to searching for the top-most qualifier. However, once all the qualifiers have shown up, the trigger does not have to occur immediately.

You can specify a minimum number of bus cycles after the time the last qualifier is seen, before the UniLab starts looking for the trigger. The default is **0 PCYCLES.** Or, you can specify a number of complete repetitions of the sequence of qualifiers. The default is **1 PEVENTS.**

See the flowchart on the next page, reproduced from section three of Chapter Three. Turn to that section for more explanation of the flow chart.


Qualifier Logic previously described
Start if 3 qualifiers
No
Yes
Start if 2 qualifiers
No
Yes
Start if 1 qualifier
No
Yes
Re-entry if using Pass counter to count sequences of qualifying events.
Decrement the Pass Counter
Clock a cycle into trace
Pass Counter =0?
No
Counting Pass Events?
No
Yes
Yes
Start search for trigger bus cycle, as detailed on the flow chart for Simple Trigger

Triggering for a filtered trace

Qualifiers also allow you to set up a trigger that is different from the filter specification. That way you can produce a filtered trace that starts after the qualifiers:

ONLY NOT 200 TO 250 ADR AFTER 368 ADR S

This trigger specification will make a filtered trace that excludes addresses 200 through 250. The UniLab will not start making the filtered trace until it sees address 368 on the bus.

Triggering on sequential events

You can use qualifiers to trigger on a consecutive sequence. Suppose there is a three byte conditional branch instruction starting at address 1010, which will jump to address 250. Other instructions also cause a jump to 250, but you are not interested in those. You want to see what happened before the branch at address 1010 is taken-- so you want to trigger when 250 follows immediately after address 1012 (the address of the last byte of the three byte opcode).

Using address 1012 as the qualifier and then 250 as the trigger will not work, because the UniLab would trigger on address 250 even if it occurred hours after address 1012. Instead, you want to have both addresses as qualifiers, and no trigger event:

NORMB AFTER 250 ADR AFTER 1012 ADR S

Watch out

Note that the qualifiers must always appear one immediately after another on the target system bus:

NORMT AFTER 250 ADR AFTER 305 ADR S

In this example, the UniLab will look for address 250 immediately after address 305. If 250 does not appear on the address bus immediately after 305, then the UniLab will go back to searching for 300.

That particular trigger only makes sense if there is a jump, call or conditional jump that could cause the next address to be 250 whose last byte is at 305.

Delay between qualifiers and trigger

Though the qualifiers must follow one after another, the trigger can come anytime after the qualifiers.

In fact, you can specify a minimum length to the delay between the qualifiers and the trigger. This is useful for avoiding trigger immediately after the qualifiers are seen.

If you want to keep the trigger disabled for 200 cycles after the qualifying sequence you can simply enter

200 PCYCLES

This is the pass count.

For example:

NORMB 10 DATA AFTER 250 ADR 200 PCYCLES S

tells the UniLab to trigger when the data is ten. The UniLab will not start to search for data 10 until 200 bus cycles after the address appears on the bus.

Repetition of the qualifying events

You can also specify to the UniLab that the sequence of qualifying events be repeated. This is useful for looking at the nth pass through a loop, or the nth call to some routine.

If you want the UniLab to wait for 150 complete repetitions of the qualifiers before starting to search for the trigger enter

150 PEVENTS

For example:

NORMT AFTER 1100 ADR 150 PEVENTS S

causes the UniLab to trigger after address 1100 has appeared on the bus 150 times.

4.11 Stepwise refinement

The UniLab allows you to build on existing trigger definitions.

Trigger definitions can be gradually expanded in complexity as you find limitations in your original idea. If you are trying to see a subroutine at address 1200, that gets called from a certain section of code, you might first enter

NORMB 1200 ADR S

only to find that the trace shows a call to the subroutine from a section of the program that you are not interested in.

You can add a qualifier and restart the analyzer by entering

AFTER 5670 ADR S

This time the UniLab will search for 1200 only after address 5670 (an address in the desired calling routine) has been detected.

If you had thought of the need for a qualifier in the first place, you could have entered

NORM 1200 ADR AFTER 5600 ADR S

This ability to polish trigger definitions makes your interaction with the UniLab conversational. You ask questions about what the system is doing and receive immediate answers -- all from the same keyboard you use to write and change the programs.

5. Save Information

Introduction

The UniLab software lets you save transcripts of your sessions and screen images as DOS text files, and also lets you save specific information as encoded DOS files.

Contents

5.1 Feature Summary

Features	Mode Panel	Commands
Print out commands that alter memory	Yes	**LOG LOG'**
Send all screen display to DOS file	Yes	**TOFILE TOFILE'**
Print out everything	Yes	**PRINT PRINT'**
Save the current screen image	NO	**SSAVE** <file>
Save a trace to a file	NO	**TSAVE** <file>
Compare current trace to one saved as a file	NO	<n> **TSAVE** <file>
Save symbol table to a file	NO	**SYMSAVE** <file>
Save current state of UniLab program	NO	**SAVE-SYS** <file>
Save program or data memory to a file	NO	<from> <to> **BINSAVE** <file>
Save a histogram setup as a file	NO	**HSAVE** <file>
Look at one line of "screen history"	NO	**Up Arrow key**
Look at one page of history	NO	**PgUp key**

Mode Panel:

3. LOG modes
LOG TO PRINT
LOG TO FILE
PRINTER

Commands:

LOG LOG'
TOFILE TOFILE'
PRINT PRINT'

5.2 Overview

While using the UniLab software, you can preserve any information you want about your session.

The software always preserves a history of your screen. You can save up to 60K in this history, which starts up every time you begin a session with the UniLab. After the history buffer fills, you start losing the oldest information.

You can also turn on features that will save all screen displays

to a text file	**TOFILE**
or to your printer.	**PRINT**

You can also log only memory changes to the printer. **LOG**

You can save the current screen image as a textfile. **SSAVE**

Other commands save, as encoded DOS files:

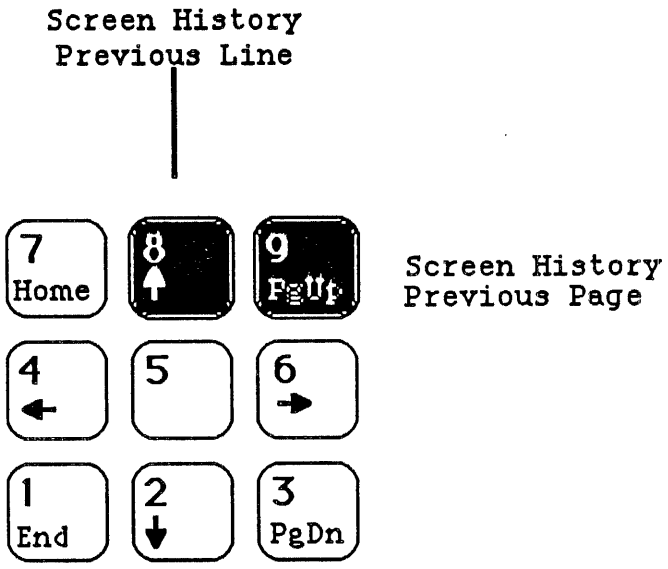
the current trace display,	**TSAVE**
the current symbol table,	**SYMSAVE**
any range of memory,	**BINSAVE**
the state of the histogram (PPA),	**HSAVE**
or the current state of the system.	**SAVE-SYS**

5.3 Screen History

The screen history always preserves the last 20 to 60K of screen display.

The information that scrolls off the top of either the the full screen or the lower window gets saved.

You look at the history with the Up Arrow and Pg Up keys, numbers 8 and 9 on the numeric key pad.



Setting the size of screen history

The size of the history is set by giving the command:

<hex # of Kbytes> **=HISTORY**

then saving the newly altered UniLab software with **SAVE-SYS.** You must then exit the program with **BYE** and reenter it, since the history buffer is allocated when the program starts up, and cannot be changed on the fly.

The maximum number of kilobytes that you can allocate to history is 3C (decimal 60K).

Use the command **?FREE** to find out how many bytes are allocated to history and to symbols. That display appears in decimal base.

5.4 Save Record of Session to a Text File

You can save the record of a session with a text log file. You can only save to one text file per session, but once you have created a log file you can turn the logging on and off at will, with the mode panel or with **TOFILE** and **TOFILE'**:

Mode Panel:

```
 3. LOG modes
LOG TO PRINT inactive
LOG TO FILE  off
PRINTER      off
NMI VECTOR   active
SWI VECTOR   active
```

Creating the log file

You cannot create the file from the mode panel-- you must use the command

TOFILE <file name>

to create the file in the first place.

This command can be used as a "command tail" when you call up the UniLab software from DOS:

A> **ULZ80 TOFILE JUNE3**

will call up the UniLab program with "june3" as the log file.

You can also name the log file from within the UniLab program, with **TOFILE** <file name>.

You will not be able to turn on logging to a file until you have named a file.

5.5 Save Record of Session to a Printer

You can save all screen output to a printer with the Mode Panel or with the **PRINT** and **PRINT'** commands.

Mode Panel:

```
  3. LOG modes
LOG TO PRINT inactive
LOG TO FILE  off
PRINTER      off
NMI VECTOR   active
SWI VECTOR   active
```

5.6 Save Only Memory Changes to Printer

This feature helps make certain that you don't forget any patches that you make to your program. It keeps a record on your printer of all commands that alter memory.

You turn it on and off with the Mode Panel, or with the commands **LOG** and **LOG'**.

Mode Panels:

```
  3. LOG modes
LOG TO PRINT inactive
LOG TO FILE  off
PRINTER      off
NMI VECTOR   active
SWI VECTOR   active
```

5.7 Save and Compare Trace

You can save the trace as an encoded file, which can later be retrieved with **TSHOW** or compared to the current trace with **TCOMP**.

You save the current trace with the command:

TSAVE <file name>

This is very useful for production checkout of a system. You can save the trace of a program on a known good system, and then use **TCOMP** to compare the known good trace to the trace of the hardware you want to check.

Comparing traces

Enter **AA TCOMP** <file name> to compare the last AA cycles of the trace (which is the whole trace) currently in the trace buffer with the trace previously saved as a file.

If the two traces are identical, the UniLab will respond with an "OK" message. Otherwise it will display 14 lines of the trace on disk including the first non-matching bus cycle, followed by the non-matching cycle in the current trace.

You can completely test a system with a few such trace comparisons, using programs that exercise the hardware of the system. The UniLab's macro capability allows you to write a macro which completely tests a system, automatically. (UniLabs are given their final test at the factory with just such a macro.)

Example: Trace compare

This example shows the result of performing **TCOMP** on a faulty trace produced by a Z80 board running the simple target program (**LTARG**). One of the address lines of the board was grounded, which pulled it low.

AA TCOMP TESTZ80.TRC

cy#	CONT	ADR	DATA		HDATA	MISC
3	B7	0003	3E12	LD A,12	11111111	11111111
5	B7	0005	015634	LD BC,3456	11111111	11111111
8	B7	0008	119A78	LD DE,789A	11111111	11111111
B	B7	000B	21DEBC	LD HL,BCDE	11111111	11111111
E	B7	000E	C5	PUSH BC	11111111	11111111
F	D7	18FF	34 write		11111111	11111111
10	D7	18FE	56 write		11111111	11111111
11	B7	000F	C1	POP BC	11111111	11111111
12	F7	18FE	56 read		11111111	11111111
13	F7	18FF	34 read		11111111	11111111
14	B7	0010	3C	INC A	11111111	11111111
15	B7	0011	3C	INC A	11111111	11111111
16	B7	0012	3C	INC A	11111111	11111111

No Good! (Above is correct.) Was:

cy#	CONT	ADR	DATA	HDATA	MISC
F	D7	18DF	34 write	11111111	11111111

TCOMP reports a discrepancy to you by showing the relevant section of the trace on disk, and then showing the non-matching line from the current trace.

You then have to perform a visual comparison of the two cycles that don't match up.

In this case, you can see that in cycle F of the trace on disk, the Z80 wrote to address 18**FF**. In that same cycle of the new trace, the Z80 wrote to 18**DF**. Since the program is the same in both cases, the difference is in the hardware.

```
          ADDRESS LINE:     7654 3210
          FF hexadecimal is 1111 1111 binary,
while     DF hexadecimal is 1101 1111 binary.
```

So, obviously, address line A5 has been accidentally grounded.

5.8 Save Symbol Table as DOS File

You can save the symbol table as an encoded file, which can later be retrieved with **SYMLOAD.**

You save the current symbol table with the command:

SYMSAVE <file name>

5.9 Save a Range of Memory as DOS file

You can save any range of emulation ROM as an encoded file, which can later be retrieved with **BINLOAD.** You can use this command to save the program you are working on.

You can also save from and load to RAM if you have first established debug control. See section 6 on Breakpoints and The DEBUG.

You save a range of memory with the command:

<from address> <to address> **BINSAVE** <file name>

5.10 Save the State of the UniLab Software

You can save the current state of the UniLab program to a command file. This allows you to save the software with a certain range of memory enabled, and with other variables set up to your preference. Saving the system will also preserve the current trace.

You can save to a file with the same name as the current **.EXE** file, or to a different one.

To save the current state of the system, use the command:

SAVE-SYS <file name>

Unless you specify a different path, the file will get saved to the Orion directory.

6. Breakpoints and the DEBUG features

Introduction

The UniLab emulator includes special hardware that makes possible virtually all of the traditional processor-pod development system features. The basic UniLab software includes all of the processor-independent debug features.

Processor-specific software packages add more features, such as the ability to change specific registers, or take advantage of special functions of the processor.

Consult the Target Application Note for your Disassembler/DEBUG software to get more information on DEBUG features.

Contents

6.1 Feature Summary

Feature	Menu	Command
To enter DEBUG:		
Establish debug control by setting a breakpoint	Yes	**RESET** <addr> **RB**
Gain debug control immediately with a hardware interrupt	NO	**NMI**
Watch the bus for some trigger spec, then issue a hardware interrupt.	NO	**RI** <trigger spec> **SI**
Within DEBUG:		
All commands for reading and altering memory work on RAM.		
Set breakpoint and let program run to the new breakpoint	Yes	<addr> **RB**
Set breakpoint at next code address	Yes	**N**
Show the "breakpoint display" again	NO	**R**
Execute the next instruction-- use when single stepping for jumps and branches.	NO	**NMI**
Alter Program Counter, then resume to breakpoint	Yes	<New PC> <addr> **GB**
Set Multiple Breakpoints	NO	<addr> <bp #> **SMBP**
Clear one multiple breakpoint	NO	<bp #> **RMBP**
Clear all multiple breakpoints	NO	**CLRMBP**
Trigger style breakpoints		
Set up a trigger for DEBUG	NO	**RI** <trigger spec>
Start program and gain debug control when trigger seen on bus	NO	**SI**
To exit DEBUG:		
Exit immediately-- set program running again	NO	**RZ**
Alter Program Counter, then exit from debug control	NO	<New PC> **G**
Alter Program Counter, then wait for the analyzer to start	NO	<New PC> **GW**

Additional (target specific) DEBUG commands alter register contents, output values to ports, etc.

Command:	**Menu:**	DEBUG MENU
RESET <addr> **RB**	F1	SET A BREAKPOINT TO ESTABLISH DEBUG CONTROL
<addr> **RB**	F2	RESUME EXECUTION TO A BREAKPOINT
N	F3	EXECUTE THE NEXT STEP (WON'T FOLLOW JUMPS)
<New PC> <addr> **GB**	F4	GO TO AN ADDRESS WITH A BREAKPOINT SET
<New PC> **G**	F5	GO TO AN ADDRESS AND EXIT THE DEBUG
	F10	RETURN TO MAIN MENU

6.2 Overview

The DEBUG commands of the UniLab software provide you with the tools traditionally associated with development systems.

DEBUG capabilities

The DEBUG features:

set single or multiple breakpoints,
interrupt the processor to gain immediate DEBUG control,
single step through code,
read and alter internal registers, and
read and alter RAM.

The UniLab's powerful bus state analyzer replaces most of the functions of the traditional debug tools. But when you have tracked a bug down to a small segment of code, it is very useful to be able to set a breakpoint and single step through the program.

DEBUG characteristics

All Orion DEBUGs have certain common requirements and characteristics. The specifics of each DEBUG are discussed in the Target Application Note for the DDB software.

This page and the next two describe the common requirements. This section of Chapter Six then goes on to describe the common DEBUG commands.

Use of target system resources

All Orion DEBUGs make use of some of your processor's resources. All of them require a several byte "reserved area" in emulation ROM that you cannot put code into, and a larger "overlay area" that you can use. These areas are movable. See the next page.

All DEBUGs use your stack.

Most DEBUGs also make use of a trap vector or an internal register of the target processor.

To support the **NMI** dependent commands, they use a hardware interrupt vector.

Movable reserved area

Both the reserved area and the overlay area are relocatable. Use **CTRL-F3** to see the current location. Use <addr> **=OVERLAY** to move them to a new starting address.

Do not put these areas too close to a 2K boundary. A good rule of thumb-- if the overlay area is already close to a 2K boundary, do not move it any closer. For example, if the default location is 0FD8, do not move it to 0FF0. However, 07D8 would be fine.

You will run into problems if you do not enable the 2K block that contains the reserved and overlay areas.

How the DEBUGs work

All DEBUGs work by downloading Orion software routines to your emulation ROM (in the overlay area) and using your processor to execute routines that display target registers, alter target RAM, etc.

While you can put code into the overlay area, most of the DEBUG routines will not work on that area of memory.

Required Code in User Program

All the routines that are downloaded to your processor require a working stack. If your processor does not have a default stack pointer, your target system code must initialize the stack pointer. You cannot set a breakpoint in your code until the stack pointer is initialized.

A few DEBUGs, such as that for the 8088 and 8086 processors, require that your code initialize other resources, such as interrupt vectors. Consult the Target Application Note for more information.

Some DEBUGs require you to issue a UniLab command that tells it what areas of RAM or ROM you want the DEBUG to use.

Connections

Each section of this chapter includes a diagram that shows you how to connect your UniLab to your processor. The command **PINOUT** shows a connection diagram on-line. Some software packages that support more than one processor require different cable connections for different processors.

Consult Appendix C to double-check your wiring.

Access to RAM, to internal registers, etc.

You cannot look at or alter RAM or internal registers until you have first established debug control. You establish debug control by setting a breakpoint (**RESET** <addr> **RB**) or by asserting a non-maskable interrupt while your target program is running (**NMI**).

For more information on debug control, check the entries for **NMI** and **RB** in the command reference chapter, or read on.

6.3 Establish Debug Control

Most of the DEBUG commands will not work until after the UniLab's special DEBUG hardware has taken control of your processor.

You can establish debug control with either a software interrupt

RESET ‹addr› **RB,**

or with a hardware interrupt

NMI,

or with a trigger spec which will cause a hardware interrupt

RI ‹trigger spec› **SI.**

You cannot invoke the DEBUG until after your program initializes the stack pointer. The DEBUG actually runs code on your processor, and then uses the RETurn instruction to resume execution of your program.

If the stack pointer is not initialized, you will not be able to establish debug control at all, or will get an obviously incorrect breakpoint-- for example, with all registers displayed as FF.

DEBUG terminology

When your program reaches a breakpoint or you issue a hardware interrupt, the UniLab will take control of your processor. Your program stops executing. You have "established DEBUG control."

Your processor is then held in an "idle loop."

When you use one of the DEBUG commands, the UniLab will release your processor from the idle loop to execute a routine that has been downloaded into the "overlay area."

Any code that you have in the overlay area gets saved before the DEBUG routines are downloaded, and then restored.

The reserved area does not get saved and restored. This area varies in size from one byte on some DDB packages to six bytes at the most.

Establish control with a software interrupt

To establish DEBUG control, you must enable reset of the target system, then set a breakpoint at an opcode address.

RESET
‹address› **RB**

The address you give must be the first address of an opcode. In the fragment of Z80 code below, you could set a breakpoint at any of the addresses that appear in the adr column. But you would not be able to set a breakpoint on address 131 or 132, for example.

Adr	**Opcode**	**Instruction**
012B	012C00	LD BC,2C
012E	7C	LD A,H
012F	BA	CP D
0130	C23801	JP NZ,138
0133	7D	LD A,L
0134	BB	CP E
0135	CA4201	JP Z,142
0138	7E	LD A,(HL)

Use a trace display, or disassemble from memory (with **DN** or **DM**) to determine where you can set breakpoints.

Example: Establish debug control with RB

In this example, we set a breakpoint on address 12F in the Z80 code fragment above. The transcript below shows the command, followed by a "resetting" message, and then the "breakpoint display."

The "resetting" message shows that the UniLab has issued a restart strobe to the target board. When the processor reaches the breakpoint address, the UniLab downloads a DEBUG routine to the overlay area and then displays the values of the internal registers.

```
RESET 12F RB resetting

AF=02A0 (Sz-a-pnc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
012F  BA       CP D                                (next step)  ok
```

Establish control with a hardware interrupt

You can either issue a non-maskable interrupt manually,
NMI,

or you can set up the UniLab to issue the hardware interrupt when a certain bus state is reached,
RI ‹trigger spec› **SI.**

Either way, the UniLab sends a signal to the hardware interrupt pin of your processor, which immediately interrupts your processor.

These commands will only work if the target system does not need to make use of your processor's hardware interrupt pin. The UniLab DEBUG makes use of the Non-Maskable Interrupt (NMI) pin. If the processor does not have an NMI pin, the DEBUG uses the maskable interrupt request (IRQ) pin of the target processor.

If your target system does make use of your processor's hardware interrupt, you should disable the UniLab's NMI features. See subsection 6.8, **Disable DEBUG**.

NMI for DEBUG control and for single-step

The "state-smart" **NMI** command will also single-step through code, following program flow.

When you use **NMI**, the command first checks whether you have your processor under DEBUG control. Then **NMI** will establish DEBUG control if you don't have it, or will single-step the processor if you do have it.

When **NMI** has to establish DEBUG control, it will print "-nmi-" on the screen. When single-stepping, it works silently.

RI and SI for trigger-style hardware interrupts

RI and **SI** produce a hardware interrupt which achieves DEBUG control a cycle or two after the bus conditions appear.

You can use **RI** and **SI** either to establish debug control in the first place, or to continue after you have already established control.

RI clears out the old trigger spec and sets up for a hardware interrupt trigger spec, and **SI** starts the analyzer. Like this:

RI <trigger spec> **SI**

Qualifiers should not be used in the trigger spec. If you do use them, the result will be that the qualifier and trigger must occur one immediately after another.

Example: Trigger style breakpoints

The example shows the setting of two breakpoints using analyzer style commands. Note that the breakpoint occurs one cycle after the trigger event occurs. The code in which we are setting breakpoints appears below, for convenience.

```
0005  015634    LD BC,3456
0008  119A78    LD DE,789A
000B  21DEBC    LD HL,BCDE
000E  C5        PUSH BC
000F  C1        POP BC
0010  3C        INC A
0011  3C        INC A
(... jump back to address 3 ... )
```

```
RI 10 ADR 3C DATA SI resetting
AF=1300 (sz-a-pnc) BC=3456 DE=789A HL=BCDE IX=E5C1 IY=E5C1 SP=1900
0011  3C        INC A                              (next step)  ok

RI 18FF ADR AFTER 0E ADR SI
AF=1228 (sz-a-pnc) BC=3456 DE=789A HL=BCDE IX=E5C1 IY=E5C1 SP=18FE
000F  C1        POP BC                             (next step)  ok
```

Common DEBUG pitfalls

1. Watchdog Timer:
Your microprocessor stops executing your program when you are at a breakpoint.

If you have a watchdog timer, it will then try to restart your target board. The watchdog thinks that something has gone wrong with your program.

You must disable the watchdog timer to use the DEBUG.

2. Stack Pointer:
The Orion overlay routines make use of your processor's stack. You cannot set a breakpoint until after your program has established a functional stack.

Most programs initialize the stack pointer as one of the first few steps. Some processors have a default value for the stack pointer, and so do not need to initialize the stack pointer.

Check whether your stack is working properly if you cannot establish DEBUG control, or if you are always getting obviously bad data, such as all registers reported as value 0FF.

3. Opcode Address:
You can only set a breakpoint on the first address of an instruction, as explained earlier.

4. Reserved bytes and the Overlay Area:
Your program cannot make use of the reserved bytes, and you cannot set a breakpoint in the overlay area.

The addresses of the reserved bytes and the overlay area appear in Appendix H, or on-line when you press **CTRL-F3**. The reserved area is between one and six bytes of ROM, and the overlay area is the area of 30 to 70 bytes above the reserved bytes.

Debug commands which refer to addresses in the overlay area may produce strange results.

6.4 Interpreting the Breakpoint Display

The breakpoint display varies from processor to processor, but always contains the same two basic parts:

1) the register display, and

```
AF=02A0 (Sz-a-pnc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
```

2) the display of the next step.

```
012F  BA       CP D                                (next step)  ok
```

Register Display

You can see the breakpoint display again with the command **R**.

The register display varies from processor to processor, but always includes the stack pointer (**SP**) and the flags register (**F**), along with the standard internal registers.

```
AF=02A0 (Sz-a-pnc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
```

Often the automatic display shows only some of internal registers -- but each DDB package has commands that will allow you to examine any internal registers or RAM while at a breakpoint.

All registers are displayed in hexadecimal, and a single letter abbreviation for each flag bit is also displayed.

```
AF=02A0 (Sz-a-pnc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
```

Notice how the flags display changes as the value stored in the F register changes-- a capital letter indicates that the bit is set high, a lowercase letter indicates that it is low:

AF=02**A0** (Sz-a-pnc)

AF=02**42** (sZ-a-pNc)

Next Step

The display of the next step shows you the **address** of the opcode which will execute next (which is the same as the program counter or instruction counter),

```
AF=02A0 (Sz-a-pnc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
012F  BA       CP D                                (next step)  ok
```

---

and the **opcode** stored at that address,

```
012F  BA       CP D                                (next step)  ok
```

---

and the **disassembled instruction** that the processor is about to execute:

```
012F  BA       CP D                                (next step)  ok
```

6.5 Within DEBUG

After you have established DEBUG control, you can:

run to another breakpoint,
single step,
follow jumps while single stepping,
change the program counter and then run to a bp,
set multiple breakpoints,
examine and alter internal registers,
examine and alter RAM or ROM,
and exit from DEBUG control.

You can also exit from the DEBUG, and start using analyzer commands again.

While your processor is under DEBUG control you can safely look at or change emulated ROM without crashing the program.

On some processors, you can also send data to a port, and examine the contents of a port.

Missed breakpoints

You will lose DEBUG control if you use **RB** or **GB** to set a breakpoint which your program never reaches.

When this occurs you can press any key and **NMI** will be executed, regaining DEBUG control. Or, you can reset the processor with **RESET** <addr> **RB**.

If you accidentally set a breakpoint in the middle of an opcode (instead of at the start) your program might crash, and will at the least have a corrupted opcode.

Run to another breakpoint

After establishing debug control, you can let the program run to another breakpoint:

<address> **RB**

You must enable **RESET** to establish debug control, but must leave it disabled when running to subsequent breakpoints. If you re-enabled **RESET** each time before you used **RB**, your program would start running again from the beginning, rather than continuing from the breakpoint where it had last stopped.

In the transcript below, the **RB** command was used, with reset enabled, to first establish debug control. **RB** automatically disabled reset. **RB** was then used to run to a second and then a third breakpoint.

```
RESET 14C RB resetting

AF=786A (sZ-a-pNc) BC=040E DE=0200 HL=1801 IX=FFFF IY=FDFF SP=1C00
014C  0B       DEC BC                              (next step)  ok

15E RB
AF=0044 (sZ-a-Pnc) BC=0086 DE=FFFE HL=0000 IX=FFFF IY=FDFF SP=1BFE
015E  39       ADD HL,SP                           (next step)  ok

177 RB
AF=0044 (sZ-a-Pnc) BC=0086 DE=1BEE HL=0000 IX=FFFF IY=FDFF SP=1BF0
0177  C9       RET                                 (next step)  ok
```

Single-Step

After you have established debug control, you can step through your program, one opcode at a time.

When you want to single-step through code you have your choice of two UniLab commands. One, **NMI**, uses a hardware interrupt to step through your code. This command will follow all jumps, branches and calls, just as you would expect. The other command, **N**, has a different but still highly useful mode of action. See below for more information.

NMI single step

As explained in the section on establishing DEBUG control, **NMI** is a "state-smart" command. It either establishes DEBUG control for you, or single-steps the processor if you already have DEBUG control.

N single step

N always sets a software breakpoint on the address just after the "next step" instruction. **N** is not appropriate when you want to single-step through a command that changes the program counter-- such as a jump, call or branch. When you are, for example, stopped at a CALL instruction, you can use **N** to set a breakpoint that will not be reached until the program returns from the CALL.

N will step through much of your code without a problem (and without using the hardware interrupt).

Example: NMI and N

In this example **N** and **NMI** are used to single-step up to a jump and then to follow the jump.

```
RESET 134 RB resetting

AF=7842 (sZ-a-pNc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
0134  BB       CP E                              (next step)  ok

N
AF=786A (sZ-a-pNc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
0135  CA4201   JP Z,142                          (next step)  ok
```

The program executes the jump to 142.

```
NMI
AF=786A (sZ-a-pNc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
0142  210018   LD HL,1800                        (next step)  ok
```

Example: N only

In the transcript below, we first establish debug control with **RB**, and single-step through a series of stack and register manipulations.

You can see the effects of the register manipulations in this code. The registers that are about to change are in **bold** text, and the ones that have just changed are underlined.

```
RESET 170 RB resetting
AF=0040 (sZ-a-pnc) BC=00DE DE=0002 HL=0F83 IX=FFFF IY=FDFF SP=1BEA
0170  EB       EX DE,HL                          (next step)  ok

N
AF=0040 (sZ-a-pnc) BC=00DE DE=0F83 HL=0002 IX=FFFF IY=FDFF SP=1BEA
0171  E1       POP HL                            (next step)  ok
N
AF=0040 (sZ-a-pnc) BC=00DE DE=0F83 HL=1BEE IX=FFFF IY=FDFF SP=1BEC
0172  F9       LD SP,HL                          (next step)  ok
N
AF=0040 (sZ-a-pnc) BC=00DE DE=0F83 HL=1BEE IX=FFFF IY=FDFF SP=1BEE
0173  C1       POP BC                            (next step)  ok
N
AF=0040 (sZ-a-pnc) BC=0086 DE=0F83 HL=1BEE IX=FFFF IY=FDFF SP=1BF0
0174  EB       EX DE,HL                          (next step)  ok
```

Change the program counter and then run to breakpoint

To resume the program at a different address than the one you are stopped at, use

<New PC> <bp address> **GB**

This command takes two arguments. It puts the first value into the program counter, and sets a breakpoint at the second value. Then the UniLab releases the processor, so the program starts at the new code address pointed to by the program counter.

Note that this can have some unexpected results-- you are interfering with the program flow.

See the example on the next page.

Example: Change PC, then run to a breakpoint

First, while stopped at a breakpoint, reset the PC and set a breakpoint on the very next opcode address.

```
RESET 160 RB resetting

AF=004C (sZ-a-Pnc) BC=0086 DE=1BFE HL=FFFE IX=FFFF IY=FDFF SP=1BFE
0160  39       ADD HL,SP                          (next step)  ok

170 171 GB
AF=004C (sZ-a-Pnc) BC=0086 DE=FFFE HL=1BFE IX=FFFF IY=FDFF SP=1BFE
0171  E1       POP HL                             (next step)  ok
```

Of course, you can set the breakpoint and the new PC to the same address.

```
RESET 160 RB resetting

AF=004C (sZ-a-Pnc) BC=0086 DE=1BFE HL=FFFE IX=FFFF IY=FDFF SP=1BFE
0160  39       ADD HL,SP                          (next step)  ok

171 171 GB
AF=004C (sZ-a-Pnc) BC=0086 DE=1BFE HL=FFFE IX=FFFF IY=FDFF SP=1BFE
0171  E1       POP HL                             (next step)  ok
```

Notice the difference in the **HL** register when you actually run the program to the next breakpoint, instead of changing the PC. This is an example of the unexpected results that come from interfering with program flow.

```
RESET 160 RB resetting

AF=004C (sZ-a-Pnc) BC=0086 DE=1BFE HL=FFFE IX=FFFF IY=FDFF SP=1BFE
0160  39       ADD HL,SP                          (next step)  ok

171 RB
AF=0040 (sZ-a-pnc) BC=00DE DE=0000 HL=0002 IX=FFFF IY=FDFF SP=1BEA
0171  E1       POP HL                             (next step)  ok
```

Set multiple breakpoints

Traditionally, multiple breakpoints are used when you do not know where the program was going to go next. You would try to "block all exits" by setting a breakpoint at every place the program could go.

The UniLab's ability to show you program flow makes multiple breakpoints obsolete. But, if you want to use them, here's how:

After establishing debug control, use

<address> <breakpoint #> **SMBP**

to set one of the eight numbered breakpoints.

You should set all but one of your breakpoints with **SMBP**, and then use

<address> **RB**

OR

<New PC> <address> **GB**

to set the last breakpoint and set the processor running again.

Establish debug control

You can also use **SMBP** before a

RESET <addr> **RB**

to establish debug control in the first place.

Clear multiple breakpoints

If you want to clear out all multiple breakpoints, use **CLRMBP.** The command <breakpoint #> **RMBP** will clear one of the breakpoints.

Example: Set multiple breakpoints

The transcript below shows an example of the use of the **SMBP** command while checking out the following code:

```
014D  79       LD A,C
014E  B0       OR B
014F  C24A01   JP NZ,14A
0152  C38000   JP 80
```

The problem to be solved: where does the program go after executing the code at address 014E. It might jump back to 014A, it might continue beyond that instruction and jump to 80. So you have to set two breakpoints:

```
RESET 14E RB resetting

AF=0D6A (sZ-a-pNc) BC=040D DE=0200 HL=1801 IX=FFFF IY=FDFF SP=1C00
014E  B0       OR B                                (next step)  ok

14A 1 SMBP
1 $014A  2 $----  3 $----  4 $----  5 $----  6 $----  7 $----  8 $----

)0 RB
AF=0D08 (sz-a-pnc) BC=040D DE=0200 HL=1801 IX=FFFF IY=FDFF SP=1C00
014A  73       LD (HL),E                           (next step)  ok
```

We find, not surprisingly, that the program jumps back to address 014A. But to find out about how the program flows, you will probably prefer to use the analyzer command <addr> **AS** as illustrated below. The analyzer trace shows you what happens each time the program reaches the code at 014E.

14E AS resetting

cy#	CONT	ADR	DATA		HDATA	MISC
-1	B7	014D	79	LD A,C	11111111	11111111
0	B7	014E	B0	OR B	11111111	11111111
1	B7	014F	C24A01	JP NZ,14A	11111111	11111111
4	B7	014A	73	LD (HL),E	11111111	11111111
5	D7	1801	00 write		11111111	11111111
6	B7	014B	23	INC HL	11111111	11111111
7	B7	014C	0B	DEC BC	11111111	11111111
8	B7	014D	79	LD A,C	11111111	11111111
9	B7	014E	B0	OR B	11111111	11111111
A	B7	014F	C24A01	JP NZ,14A	11111111	11111111
D	B7	014A	73	LD (HL),E	11111111	11111111
E	D7	1802	00 write		11111111	11111111

Examine and alter internal registers

The breakpoint display shows your internal registers. Of course, this display varies from processor to processor.

You can display all registers again with **R**.

And you can alter them with commands that follow this pattern:

<value> **=Name_of_Register**

The commands for altering registers are processor specific. For example, the Z80 package includes:

=AF **=BC** **=DE** **=HL** **=IX** **=IY**

Check the glossary section in the Disassembler/DEBUG Target Application Note for your processor, or press **CTRL-F3**.

Example: Alter the flags register

Notice how, in the example below, we change the flow of the program by altering the "Zero" Flag.

R shows you the register display again-- very handy for verification after you've changed a register.

After we alter the flag, we single step, and see that the program takes the jump, because of the change to the flag.

```
RESET 12F RB resetting

AF=02A8 (Sz-a-pnc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
012F  BA       CP D                                (next step)  ok

N
AF=0242 (sZ-a-pNc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
0130  C23801   JP NZ,138                           (next step)  ok

0202 =AF  ok

R
AF=0202 (sz-a-pNc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
0130  C23801   JP NZ,138                           (next step)  ok

NMI
AF=0202 (sz-a-pNc) BC=002C DE=0278 HL=0278 IX=FFFF IY=FDFF SP=1C00
0138  7E       LD A,(HL)                           (next step)  ok
```

Examine and alter RAM

While stopped at a breakpoint, you can use all the memory access commands to examine and alter either RAM or ROM.

If you have not established debug control, the UniLab will try to establish DEBUG control for you when you attempt to access target RAM.

Remember that access to emulation ROM while the target system is running will cause your target program to crash.

For details on the memory commands, see section 3: Examining and Altering Memory.

6.7 Exit from DEBUG

There are four ways to exit from the DEBUG:

1) **RZ** immediately releases the program from debug control, so that it starts running again,
2) <addr> **G** releases the processor from debug control after changing the Program Counter,
3) <addr> **GW** changes the Program Counter, and then waits until you restart the analyzer, before it releases the processor,
4) or you can simply define a trigger spec, and start up the analyzer.

If you exit from DEBUG control by starting up the analyzer be sure to remember that the DEBUG has disabled reset. If you want the program to start over from the beginning, you have to enable automatic resetting with **RESET** or the mode panel (function key 8).

You will definitely want to use **NORMx** to clear out the special trigger specifications that the DEBUG commands use.

A simple alternative is to just use **STARTUP**, which clears out the previous trigger and starts the program over from the beginning.

Exit after a target system crash

If you crash the target system while you are in the DEBUG, you will need to start the target program over from the beginning.

Examples: Exiting from the DEBUG

If you want to watch the processor when it resumes execution of the program use:

<New PC> **GW**
<analyzer trigger spec>

to change the program counter, and then wait for the analyzer to start up. The analyzer trigger specification can appear on the same line or on a separate line.

For example:

8 GW 05 AS

```
cy#   ADR DATA
 -5  0027 3C       INC A
 -4  0028 3C       INC A
 -3  0029 C30300   JP 5
  0  0005 015634   LD BC,3456
  3  0008 119A78   LD DE,789A
  6  000B 21DEBC   LD HL,BCDE
  9  000E C5       PUSH BC
```

If you want to release the processor and set it running, without setting any analyzer trigger spec, use:

<New PC> **G**
or
RZ.

6.8 Disable the DEBUG: How and Why

Why

You might need to disable the DEBUG features if your target needs the resources utilized by DEBUG, or if you want to look at a program running in ROM chips.

Resources used

The Orion DEBUG software packages use between one and six bytes of emulation ROM as a reserve area, and overlay code into another 30 to 70 bytes.

In addition, the breakpoint command often needs exclusive use of a software interrupt vector, and the NMI features always require a hardware interrupt vector.

If your hardware does make use of your processor's hardware interrupt, then you will want to disable the UniLab software's use of that feature.

You can put your own code in the overlay area, but never in the reserved area (**CTRL-F3** tells you where the reserved area is on your processor). If you disable the DEBUG, then you can make use of these small areas of memory, and of the vectors used by the DEBUG features.

Special note: running program in ROM chips

If you want to run a program from a ROM chip on your target board, you must first clear out emulation memory enables with **EMCLR**. This automatically disables the DEBUG. You will, of course, still be able to use the analyzer and disassembler. If you later re-enable emulation memory, you will need to manually re-enable the DEBUG features.

How

Disable NMI features only

Use either the Mode Panel option "NMI VECTOR" or the command **NMIVEC'.**

Whichever you use, the result is the same-- **NMI** and the features dependent upon it **(RI,SI)** no longer work, but the rest of the DEBUG commands, such as **RB** and **GW**, work fine.

Disable all DEBUG features

You turn off all DEBUG commands, including **NMI**, with either the mode panel option "SWI VECTOR" or the command **RSP'.**

Mode panel

```
 3. LOG modes
LOG TO PRINT inactive
LOG TO FILE  off
PRINTER      off
NMI VECTOR   active
SWI VECTOR   active
```

7. Program EPROMs

Introduction

You can do all your EPROM programming from the menu system. The menus allow you to program any EPROM with just a few key strokes. The menus also tell which personality module to use.

When your program is working perfectly under emulation, you can use the EPROM programmer to copy it into virtually any single-supply EPROM or EEPROM, directly from the emulation memory. To program a 2716, for example, from target locations 800 to FFF, you just put an erased 2716 in the socket and choose the appropriate menu option. You will be prompted for the starting and ending addresses, then the ROM will be programmed. Make certain the addresses you program from are enabled.

Erase check, programming, and verification will immediately begin, and the LED to the right of the socket will light. The light goes out when the PROM has been programmed (usually just a few seconds).

Contents

7.1 Feature Summary

Though there are commands for burning programs into each type of EPROM we support, we recommend that you use the EPROM burning Menu whenever possible. These menus are always just a few keystrokes away, and include reminders about which PM (personality module) you need for each EPROM.

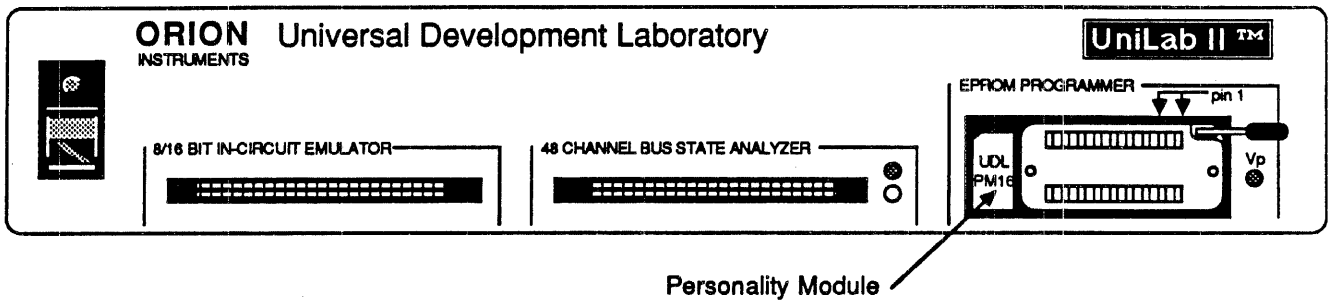
The only time you would really need the command rather than the menu item, is when you want to burn a EPROM from within a macro. See also Appendix G for information on EPROMs.

All the EPROM programming commands are covered by the menus:

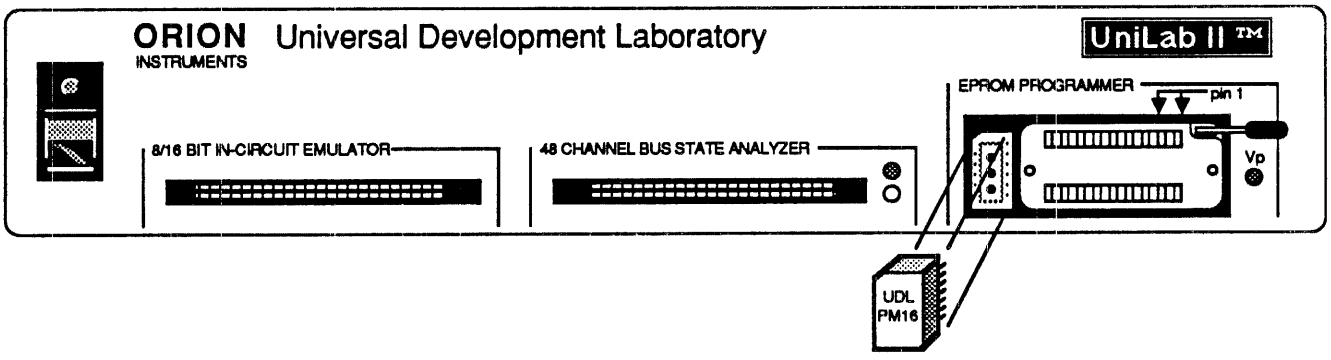
	Menu:		**Command:**
	PROM PROGRAMMING MENU #1		
F1	PROGRAM A 2716	(use PM16 personality module)	**P2716**
F2	PROGRAM A 2532	(use PM16 personality module)	**P2532**
F3	PROGRAM A 2732A	(use PM32 personality module)	**P2732A**
F4	PROGRAM A 2764A	(use PM64 personality module)	**P2764**
F5	PROGRAM A 27128A	(use PM56 for A version)	**P2764**
F6	PROGRAM A 27256A	(use PM56 personality module)	**P27256**
F7	PROGRAM A 27512	(use PM512 personality module)	**P27512**
F9	Next page of Prom Programming Menu		
F10	RETURN TO MAIN MENU		
	PROM PROGRAMMING MENU #2		
F1	PROGRAM A 27C16	(use PM16 personality module)	**PD2716**
F2	PROGRAM A 48016	(use PM16 personality module)	**P48016**
F3	PROGRAM A 27C32	(use PM16 personality module)	**P27C32**
F4	PROGRAM A 2764	(use PM64 personality module)	**PD2764**
F5	PROGRAM A 27128	(use PM64 personality module)	**PD2764**
F6	PROGRAM A 27256	(use PM56 personality module)	**PD27256**
F9	RETURN TO PROM READER MENU		
F10	RETURN TO MAIN MENU		

7.2 Personality Modules

Whether reading or burning an EPROM or EEPROM, you must have the correct Personality Module in the 16 pin socket just to the left of the EPROM PROGRAMMER socket.



Because control signals vary from one type of EPROM to another, the Orion UniLab needs the personality module to alter the voltage and pin location of control signals. The change in personality module makes it possible to program all the most popular EPROMs.



See Appendix G for full information on EPROMs and Personality Modules.

The UniLab is shipped with:

For 21 volt EPROMs:

PM16 for programming 2716, 27C16 and 2532 EPROMs.

PM32 for programming 2732 and 2732A EPROMs.

PM64 for programming 2764, 27C64 and 27128 EPROMs.

For 12.5 volt EPROMs:

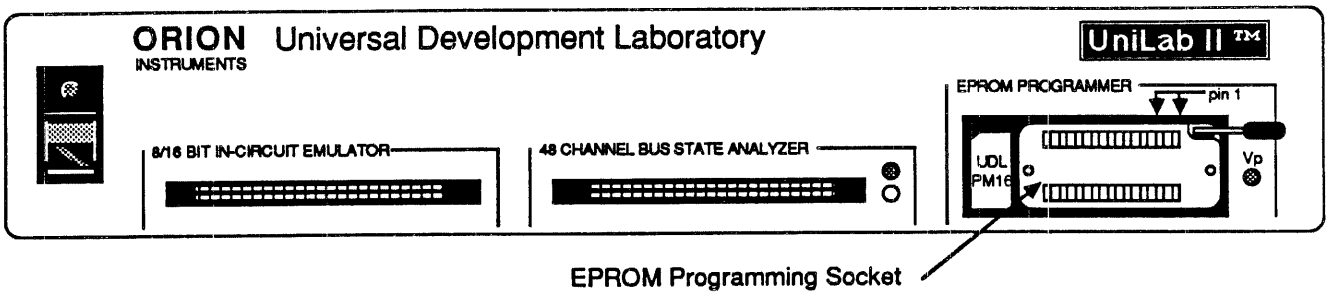
PM56 for programming 12.5 volt programmed EPROMs such as 2764A, 27C64, 27128A, 27C128, and 27256.

You can also purchase:

PM512 for programming 12.5 volt 27512 EPROMs.

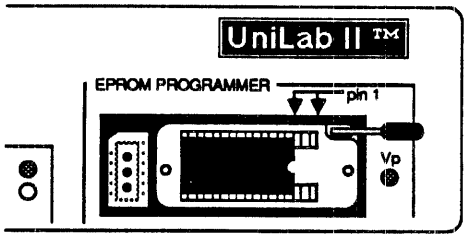
PM56-21 for programming 21 volt 27256 EPROMs.

7.3 Plug in PROM

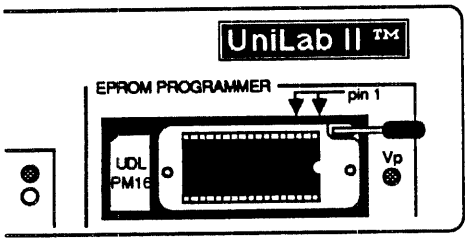


Be sure to plug PROMs into the socket with the notch to the right.

24-pin EPROMs should be inserted into the socket shifted as far to the left as possible:



24 Pin EPROM in Programming Socket



28 Pin EPROM in Programming Socket

28-pin EPROMs will fill up the whole socket.

Never turn power on and off with a PROM in the socket-- this could erase location 0. The same warning applies to changing the personality module with the PROM in the socket.

In general, don't leave the PROM in the socket any longer than necessary to read or program it.

The UniLab's smart programming algorithm guarantees a 4:1 margin on stored charge while taking a minimum amount of programming time. An erase check is done before programming starts and all locations are verified during programming.

7.4 Program the EPROM

Make certain that you have enabled the range of memory from which you wish to program. Use **ESTAT** if you want to check the status of emulation memory.

With your EPROM in the programming socket, the correct personality module in place, and your program residing in emulation ROM, you are ready to program the EPROM.

Program with the menu

Get into one of the two PROM PROGRAMMING MENUs and press the correct function key for your EPROM. You will be prompted for the start and end addresses of the memory range that you wish to program into the PROM.

Program with a command

If you wish, you can program your EPROM with a command rather than the menu. All these commands need two parameters:

‹start addr› ‹end addr› PROM-PROGRAMMING-COMMAND.

The list of commands for programming each type of EPROM appears both in Appendix G and in the feature summary at the start of this section.

7.5 Calculate Checksums

If you want to put checksums in your EPROMs, you should calculate the checksum value before programming the EPROM, and store that value in emulation ROM.

The **CKSUM** command computes the value for you. Enter:

adr toadr **CKSUM**

to calculate the sum.

Then use the **MM!** command to put the checksum in the desired location before burning the PROM.

Be sure to have a known value (such as 0 or 1) in the location you plan to put the checksum in (usually the top or bottom of memory) before executing **CKSUM.**

7.6 Verify Your PROM

PROMs are verified during the programming process.

However, if you want to separately verify a PROM, you can read it into another area of memory and use the **MCOMP** command to compare to the original data.

7.7 Program PROMs for 16-bit Processors

When the 16-bit mode has been selected (by entering **16BIT**) the prom programmer will automatically select either all odd or all even bytes, depending on the first address you specify.

```
1 TO FFF RPROM
    or
1 TO FFF P2716
```

will thus read or write odd bytes only, while

```
0 TO FFE P2716
```

will write even bytes only.

The same rule applies when programming or reading using the PROM menus (**F9** under the menu system).

7.8 Standalone PROM Programming

Programming a large EPROM can take a significant amount of time. You can use the standalone PROM programming ability of the UniLab, and use your host computer for some other task while the UniLab burns the EPROM.

To do this, you type the command **STANDALONE** and then either type an EPROM programming command use one of the EPROM programming menus to start the programming operation going.

This will cause the UniLab to do its work without needing to be in contact with the host computer. You can exit the UniLab program, even disconnect the UniLab from the host machine. When the UniLab is finished programming, the red light next to the EPROM PROGRAMMING socket will go out.

You can then call up the UniLab software again, being certain to press a key during the software initialization-- this will disable the automatic initialization signal to the UniLab hardware. Then use the command **PROMMSG** to get the message that tells you the completion status of the programming operation.

If you initialized the UniLab since the **STANDALONE** command, or you turn it off, you will lose the completion status message.

7.9 Sample Macro for Production of EPROMs

It can be tiring, when programming many identical EPROMs, to keep typing in the same series of instructions to the menu.

Fortunately, you can make a simple macro that will take care of the EPROM programming for you. For example, if you are burning a 2764 with the code that starts at 0 and goes to 1300:

```
: BURN  0 1300 P2764 ;
```

After that, all you have to do to program the ROM is type **BURN**.

Appendix F tells you more about macros.

If you choose to not use the menus, check Appendix G to find out more about what commands and **PM**s you'll need for each EPROM.

8. Generate Stimuli

Introduction

Often in system checkout it is useful to build a switch panel to allow system inputs to be changed easily. The UniLab stimulus outputs make this unnecessary by providing eight latched output bits that are controlled from your keyboard.

Contents

8.1 Feature Summary

Feature	Menu	Command
Generate a high signal on one wire	Yes	<wire #> **SET**
Generate a low signal on one wire	Yes	<wire #> **RESET**
Define bit pattern of all 8 wires	Yes	<hex byte> **STIMULUS**

Command:	**Menu:** STIMULUS MENU	
<wire #> **SET**	F1	SET A STIMULUS BIT
<wire #> **RESET**	F2	RESET A STIMULUS BIT
<hex byte> **STIMULUS**	F3	DEFINE ALL 8 STIMULUS BITS
	F10	RETURN TO MAIN MENU

8.2 How to Do It

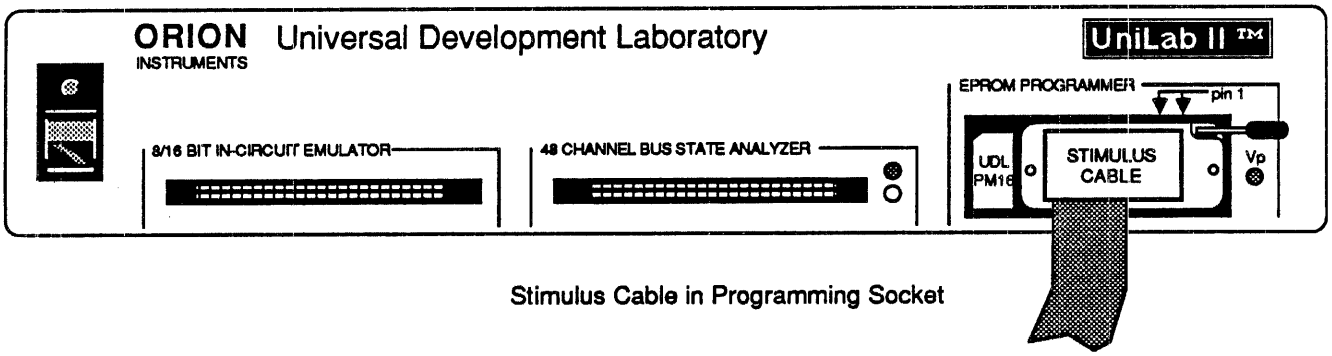
The eight stimulus generator signals, asserted at the EPROM programmer socket, can be individually set or reset from the keyboard, set and reset as a group, or programmed to produce a repeating pattern.

A ninth output (ST-) gives a 4-microsecond low pulse whenever any of the other wires are changed.

Connecting Stimulus Cable

The stimulus cable actually plugs into the EPROM programming socket and brings the signals out to .025" receptacles, like the ones used on the analyzer cable. These output signals of the UniLab can be plugged into wire wrap pins or DIP-CLIPs.

You can connect the signals to the inputs of your target system, and use the stimulus generator as a "control panel" during system checkout.



Stimulus Cable in Programming Socket

Specify an 8 bit stimulus

You can specify the 8 bits of the stimulus signal with <value> **STIMULUS.** For example, to make bits 7 and 2 high, while all other bits are low, type in:

84 STIMULUS

The number 84 hex is, of course, 1000 0100 binary.
bit # 7654 3210

Change one bit at a time

You can also set or reset the bits individually with <bit #> **SET** and <bit #> **RES.** For example,

1 SET

will set bit # 1 high.

You can also use these two commands to "pulse" target system inputs. For example, to pulse an "active high" signal with stimulus wire 3:

3 SET 3 RES

Stimulus generator and macros

You can assign a convenient name to any stimulus configuration by simply preceding the name with a colon, and ending the definition with a semicolon. For example,

```
: START1  0 SET  1 RES  3 RES  ;
```

will define a word **START1**, which causes the UniLab to perform three operations: set stimulus #0 high, then set stimulus #1 low, last set stimulus #3 low.

You could also define **START** this way:

```
: START2  01 STIMULUS ;
```

though this will have a slightly different effect than the first definition-- it sets bit 0 high, bits 1 through 7 low, and it does this all at the same moment.

The stimulus generator commands are very useful in test programs. You can use the generator to change the inputs to the target system in sequence, and then compare the resulting traces. See Appendix F for more info on test macros.

9. Special Keys

Introduction

The diagrams here are repeated from section Four of Chapter Three. See that chapter for more information about cursor and function keys.

Contents

9.1 Feature Summary

When you are in command mode, some UniLab features can be accessed through the function keys and cursor keys.

In the UniLab software, the cursor keys are always used by themselves or with the **Ctrl** key. The function keys can be used by themselves, or while you are holding down any one of:

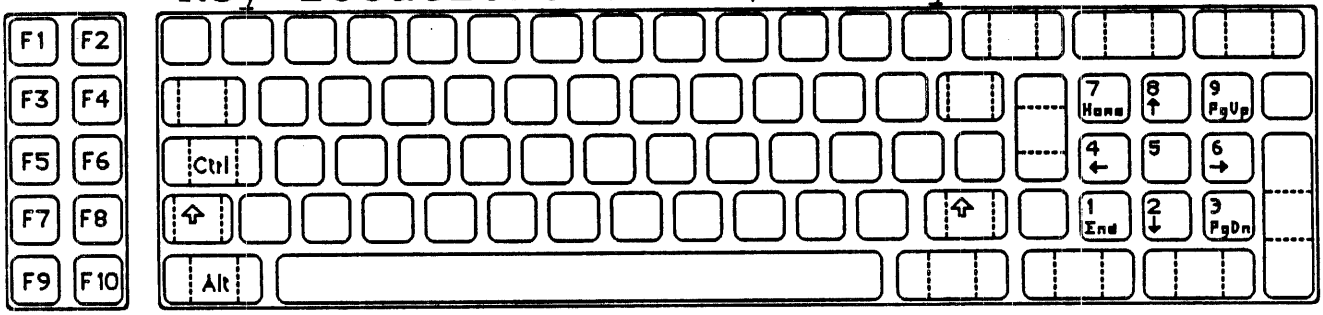
the **ALT** key,
the **SHIFT** key,
the **CTRL** key.

This means that you really have access to forty function keys.

Often used commands and help screens have been pre-assigned to many of the 40 function keys. You can change the function associated with any of the function keys.

You use the cursor keys with
the trace display,
the screen history,
the split screen,
the command line editor, and
textfiles.

Key locations on PC/AT keyboard



Function keys

Cursor keys

9.2 Function Keys

All the commands that you call up with the function keys can also be executed by typing in the command-- but it is usually more convenient to use the function key. The only time you would need to use the command is within a macro definition.

See the commands assigned to the function keys

Function key one tells you the current assignments of the function keys.

Hit function key one (**F1**) to find out what commands have been assigned to the "bare" function keys.

Hit **F1** while holding down **ALT** to see the current assignments of the ALTered function keys.

Hit **F1** while holding down **SHIFT** to see the current assignments of the SHIFTed function keys.

Hit **F1** while holding down **CTRL** to get a display of the help screens assigned to the CONTROLled function keys.

Change the commands assigned to the function keys

You can easily change the command that gets executed by any function key. You use one of four commands to do this. All of them take the same parameters:

<# of key> **FKEY** <command>
<# of key> **ALT-FKEY** <command>
<# of key> **SHIFT-FKEY** <command>
<# of key> **CTRL-FKEY** <command>

Any command that does not take parameters can be reassigned to a function key-- including any macros that you write.

Function Key assignments when no other key held down

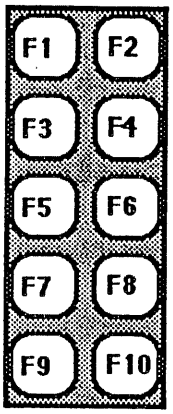
HELP with general instructions for using glossary. Also Function Key assignments.

Next Step - Execute next instruction. Will not follow jumps or branches.

Restore window split to Default sizes.

TSTAT - Display current trigger spec.

STARTUP - Issue reset pulse to target and trace first cycles of target operation.



SPLIT mode - Enter/Exit split screen mode.

NMI - Issue pulse on NMI- line to target, to gain DEBUG control or to single step through code.

MODE - Bring up pop-up mode panels for changing display or system modes.

MENU - Enter/Exit menu mode.

Function Key assignments when Ctrl key held down

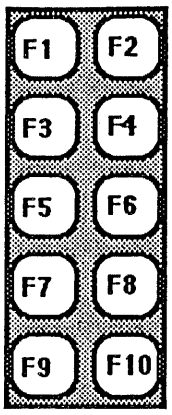
Help for using on-line displays

Help for Debuggers

Help for Emulation memory functions

Help for loading/ saving programs

Help for displaying/ altering memory



Help for using windows

Help for simple analyzer triggers

More help for analyzer triggers

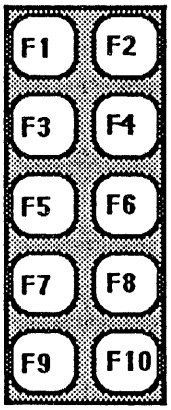
Help for mode panel switches

Help for trace display

Function Key assignments when ⇧ key held down

List Function Key assignments for Shift

RES- - Pulls RES- output line low, and holds it low



MEMO - Bring up system editor for use as custom memo pad

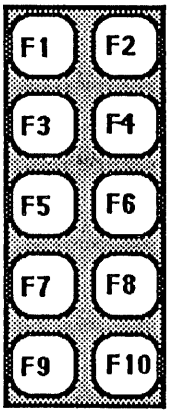
ASC - Show ASCII characters and hexadecimal code

WSIZE - Set new window split size

Function Key assignments when Alt key held down

List Function Key assignments for Alt

SSAVE - Save the screen image as a text file



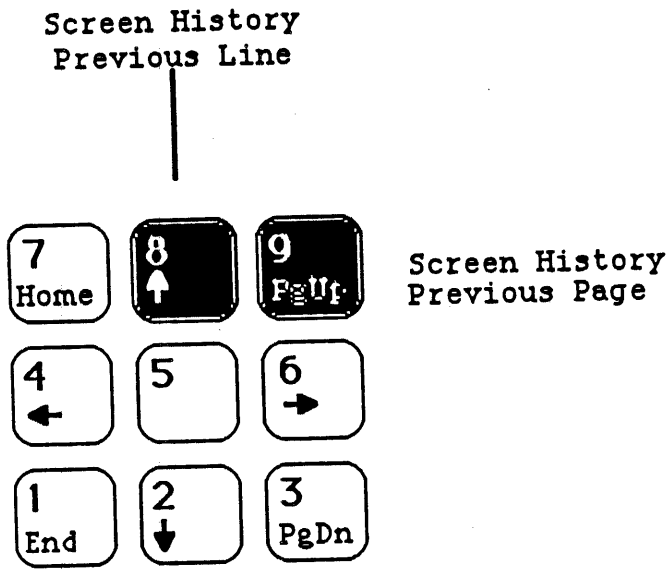
Call up the Program Performance Analyzer Menu

9.3 Cursor keys

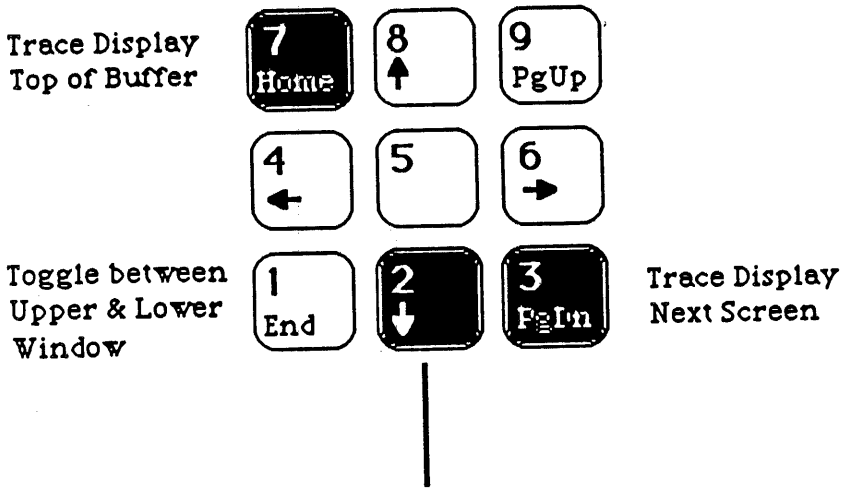
You use the cursor keys on the numeric key pad to move through various displays. The functions of the keys changes as you change the task you are working on.

This page and the two following summarize the different purposes of the cursor keys.

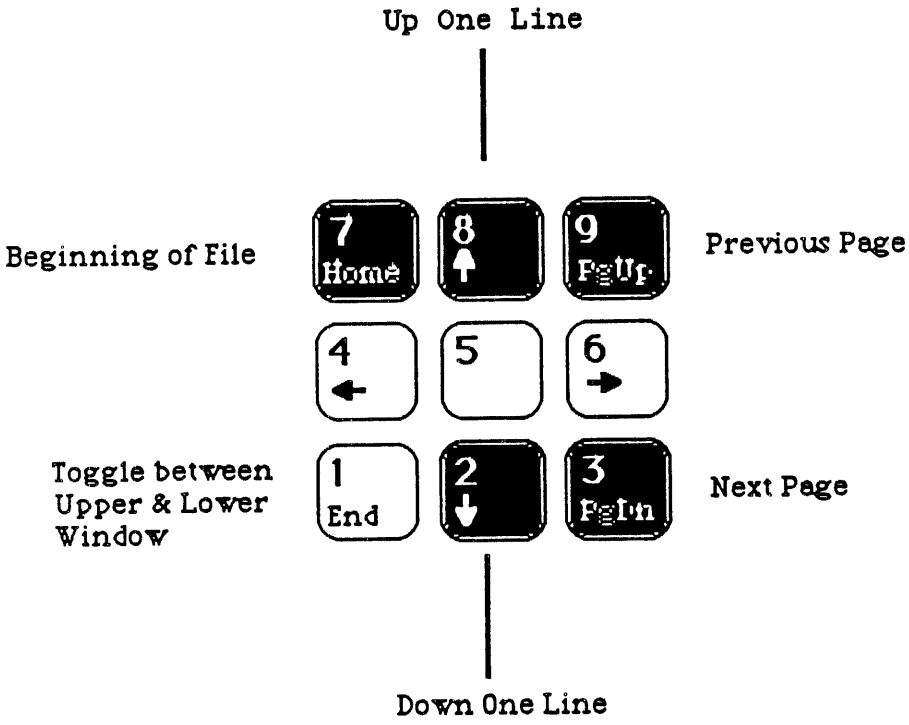
Cursor Keys and the Screen History



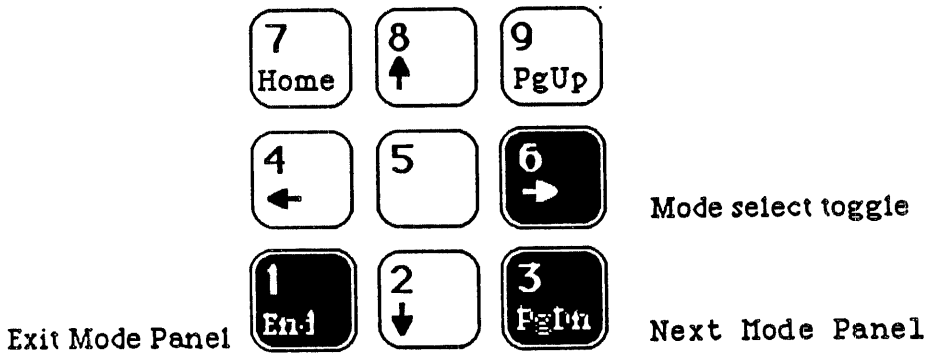
Cursor Keys and the Trace Buffer Display



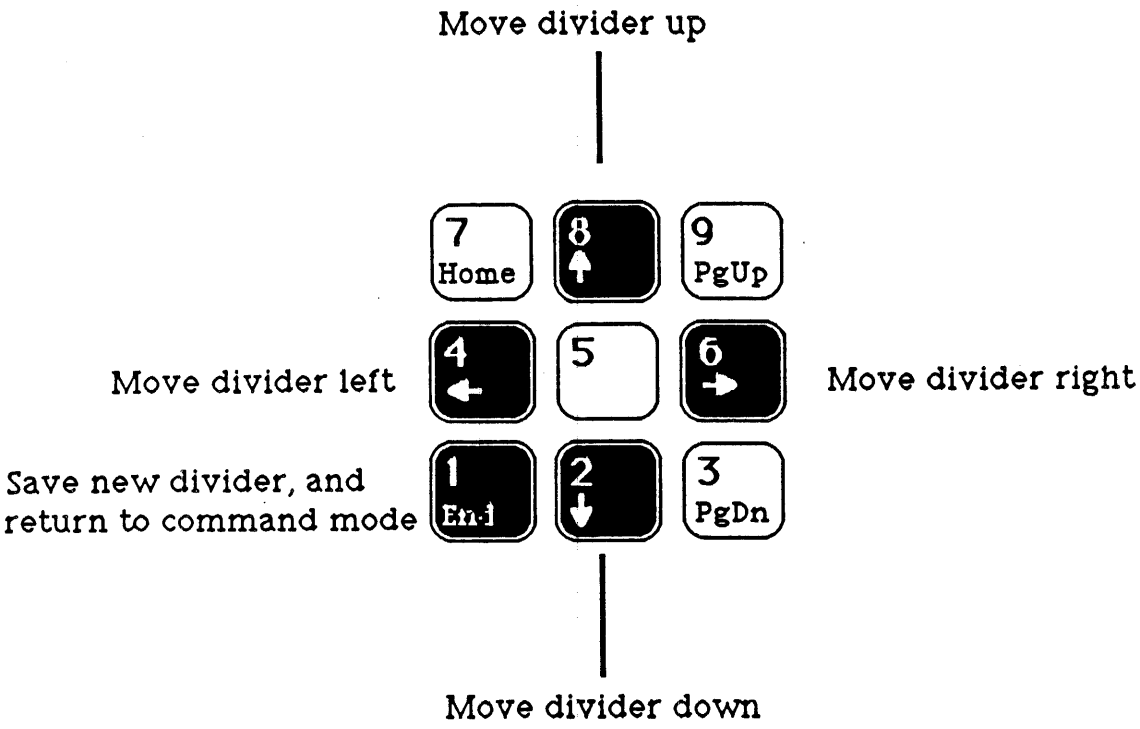
Cursor Keys and Textfiles



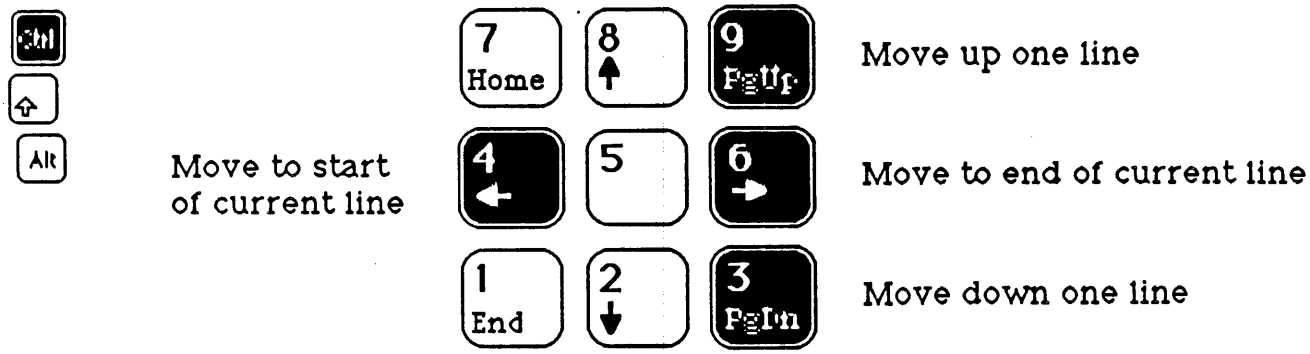
Cursor Keys and the mode panel
(enter the mode panel with **F8**)



Cursor Keys and split screen setup
(enter the setup screen with **Shift-F8**)



Cursor Keys and the command line editor
Press the cursor keys while holding down **Ctrl** key.



10. Mode Panels

Introduction

The mode panels, available at the press of a button (**F8**), let you toggle features on and off without any need to remember commands.

The Mode Panels are very simple to use, including on-line help.

Contents

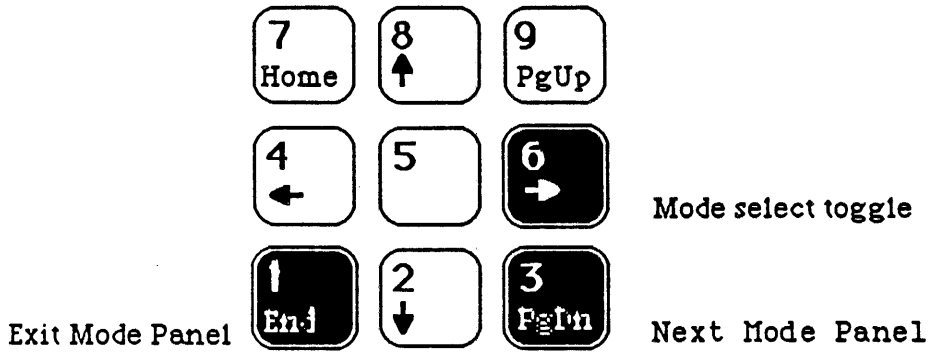
10.1 Feature Summary

Press **F8** to get into the Mode Panels. Once you have a mode panel on the screen, you can run through all of them, by hitting **F8** repeatedly.

Use the **UpArrow** and **DownArrow** keys to move around within each mode panel, from option to option.

The **RightArrow** toggles the current option. Press **F1** to get help screen for the current option.

Press **END** to exit from the Mode Panels.



Equivalent commands:	Mode Panel:
	1. ANALYZER modes
DASM DASM'	DISASSEMBLER
SYMB SYMB'	SYMBOLS
RESET RESET'	RESET
	2. DISPLAY modes
SHOWM SHOWM'	MISC COLUMN
SHOWC SHOWC'	CONT COLUMN
<value> **=MBASE**	MISC # BASE
PAGINATE PAGINATE'	PAGINATE
HDG HDG'	FIXED HEADER
	3. LOG modes
LOG LOG'	LOG TO PRINT
TOFILE	LOG TO FILE
PRINT PRINT'	PRINTER
NMIVEC NMIVEC'	NMI VECTOR
RSP RSP'	SWI VECTOR

10.2 Analyzer Modes

Disassembler

Mode Panel:	Command:	
1. ANALYZER modes		
DISASSEMBLER	**DASM**	**DASM'**
SYMBOLS		
RESET		

What it does

When you don't want or don't need to see the assembly language instructions that each opcode represents, you can turn off the trace disassembler and then look at the same trace again. The trace disassembler remains off until you enable it again.

When to use

You will need to disable the disassembler only when looking at heavily filtered traces or when trying to solve a target board hardware problem.

Example

The display below shows the first fourteen cycles of the Z80 test program, with the disassembler on and with the disassembler off.

The LD A,12 instruction is underlined and the **PUSH BC** instruction is highlighted in both traces. The disassembled display has been extended so the cycle numbers in the two displays match up.

DISASSEMBLER ON				DISASSEMBLER OFF		
cy#	**ADR**	**DATA**		**cy#**	**ADR**	**DATA**
0	0000	310019	LD SP,1900	0	0000	31
				1	0001	00
				2	0002	19
3	0003	3E12	LD A,12	3	0003	3E
				4	0004	12
5	0005	015634	LD BC,3456	5	0005	01
				6	0006	56
				7	0007	34
8	0008	119A78	LD DE,789A	8	0008	11
				9	0009	9A
				A	000A	78
B	000B	21DEBC	LD HL,BCDE	B	000B	21
				C	000C	DE
				D	000D	BC
E	**000E**	**C5**	**PUSH BC**	**E**	**000E**	**C5**
F	18FF	34 write		F	18FF	34
10	18FE	56 write		10	18FE	56
11	000F	C1	POP BC	11	000F	C1
12	18FE	56 read		12	18FE	56
13	18FF	34 read		13	18FF	34
14	0010	3C	INC A	14	0010	3C

Symbols

Mode Panel:	Commands:	
1. ANALYZER modes		
DISASSEMBLER		
SYMBOLS	**SYMB**	**SYMB'**
RESET		

What it does

When symbols are enabled, the trace disassembler will search the symbol table for the symbolic equivalent of any number. You will also be able to use a symbol anywhere that you can use a number.

You will need to use either **SYMLOAD** to load an ORION symbol table format, or **SYMFILE** to load the symbol table produced by your linker.

When to use

When you load a symbol table or define a symbol, symbol translation gets turned on. However, you may want to turn symbol translation off and see only the numeric values.

Reset

Mode Panel:	Commands:
1. ANALYZER modes	
DISASSEMBLER	
SYMBOLS	
RESET	**RESET RESET'**

What it does

After reset is enabled, the UniLab will issue a reset strobe to the target board whenever you use **S, AS, SI** or **RB.**

STARTUP enables reset.

RB and **NMI** disable reset.

When to use

When generating a trace, enable reset when you want to start the target program from the beginning.

When using **RB,** enable reset to gain DEBUG control. Keep it disabled after that.

10.3 Display Modes

MISC column

Mode Panel:	Commands:
2. DISPLAY modes	
MISC COLUMN	**SHOWM SHOWM'**
CONT COLUMN	
MISC # BASE	
PAGINATE	
FIXED HEADER	

What it does

Toggle this option on to display the eight miscellaneous inputs (and, on an 8-bit data bus processor, the eight high data inputs). Toggle off to hide these inputs.

When to use

When your MISC wires (M0 through M7) are connected to signals on your board, you will want to see the signals displayed. Otherwise that display just clutters up the screen.

CONTrol column

Mode Panel:	Commands:
2. DISPLAY modes	
MISC COLUMN	
CONT COLUMN	**SHOWC SHOWC'**
MISC # BASE	
PAGINATE	
FIXED HEADER	

What it does

Toggle on to display the "control" column inputs. These signals connect to the bus control pins and, in some cases, the address pins of your microprocessor. Processors with a greater than 16-bit external address bus will have the low nibble (bits 0 through 3) of the control column connected to bits 16 to 19 of the address bus.

When to use

When you are troubleshooting your target board, you will need to see the CONT column.

Most of the time it simply clutters the screen-- unless you need to routinely see the 20-bit address. Remember that the control column data is always gathered, even when you choose not to display it.

MISCellaneous Number Base

Mode Panel:	Command:
2. DISPLAY modes	
MISC COLUMN	
CONT COLUMN	
MISC # BASE	<n> **=MBASE**
PAGINATE	
FIXED HEADER	

What it does

Toggle this option to change the number base of the miscellaneous column display. The default is binary display.

When to use

When you have the MISC inputs connected to a port or a register, you will probably want to display that column in octal or hexadecimal, rather than in binary.

This feature alters the display base of the HDATA column at the same time.

Paginate

Mode Panel:	Commands:	
2. DISPLAY modes		
MISC COLUMN		
CONT COLUMN		
MISC # BASE		
PAGINATE	**PAGINATE**	**PAGINATE'**
FIXED HEADER		

What it does

Toggle this option off to see the entire trace display scroll to the end without stopping. Toggle it on to have the display stop after each screenful of display.

When to use

You usually want the display to stop after each screenful of display. But sometimes, when you are sending data to a file or a printer, you might want to have the whole trace scroll on by.

Fixed Header

Mode Panel:	Commands:
2. DISPLAY modes	
MISC COLUMN	
CONT COLUMN	
MISC # BASE	
PAGINATE	
FIXED HEADER	**HDG HDG'**

What it does

Toggle this on to have a fixed header on the trace display in your lower display window. Toggle off to have the usual header, which scrolls off the screen with the trace display.

When to use

The choice of fixed or scrolling header is a personal aesthetic decision. Fixed headers speed up the display a little bit. Use whichever you prefer.

10.4 Log Modes

Log to Print

Mode Panel:	Commands:
3. LOG modes	
LOG TO PRINT	**LOG LOG'**
LOG TO FILE	
PRINTER	
NMI VECTOR	
SWI VECTOR	

What it does

Memory writes are recorded on the printer when you toggle this option on. Use option PRINTER to record everything on the printer.

LOG TO PRINT will record all operations that change memory--including the assembly language instructions that you give to the line-by-line assembler. You will even record the UniLab writing into the reserved area when you start the analyzer.

When to use

This option allows you to record all the experimental changes you make while tracking down defects or modifying algorithms in your software.

Log to File

Mode Panel:	Commands:	
3. LOG modes		
LOG TO PRINT		
LOG TO FILE	**TOFILE**	**TOFILE'**
PRINTER		
NMI VECTOR		
SWI VECTOR		

What it does

Toggle on to resume logging of all screen output to a DOS text file. You must use the command **TOFILE** <filename> to create the file and start logging to it in the first place.

After the recording has been started, you can use this option to suspend and then resume recording.

When to use

When you want a record of a UniLab session. Logging to a file will slow down all output-- but much less than logging to a printer.

Logging to a file has other advantages over logging to a printer. You can later review the file with the DOS command sequence:

TYPE <filename> **|** **MORE**

You can edit the file with any text editor that handles or "imports" text files. The file can always be printed later, in the original or edited form.

Printer

Mode Panel:	Commands:
3. LOG modes	
LOG TO PRINT	
LOG TO FILE	
PRINTER	**PRINT PRINT'**
NMI VECTOR	
SWI VECTOR	

What it does

Toggle this option on to send all screen output to your printer.

When to use

Most useful when you want to immediately record a single trace buffer, or a few breakpoint displays. Otherwise, logging to a file will be a better option. See the previous page.

NMI vector

Mode Panel:	Commands:
3. LOG modes	
LOG TO PRINT	
LOG TO FILE	
PRINTER	
NMI VECTOR	**NMIVEC NMIVEC'**
SWI VECTOR	

What it does

Enables and disables the NMI features. The NMI features make use of the hardware interrupt request feature of the target processor-- either the non-maskable interrupt (NMI), or the interrupt request (IRQ) if the processor lacks an NMI. You can also disable all DEBUG features. See the next page.

With this option disabled, the following features will no longer function:

NMI,
RI ‹trigger spec› **SI,**
and auto-DEBUG control for RAM read and write.

With this option enabled, the UniLab will write into the hardware interrupt vector location of your processor whenever you start up the analyzer with reset enabled.

When to use

When the UniLab has the use of the hardware interrupt vector, you can achieve DEBUG control at any time, with the **NMI** command. See section 6 of this chapter for more information on **NMI**.

However, you should toggle this option off if your target hardware/software system uses the hardware interrupt of your processor.

SWI vector

Mode Panel:	Commands:
3. LOG modes	
LOG TO PRINT	
LOG TO FILE	
PRINTER	
NMI VECTOR	
SWI VECTOR	**RSP RSP'**

What it does

Toggle this option off to disable all the DEBUG features of the UniLab system. See section 6 of this chapter for information on DEBUG features.

With this option disabled, you will not be able to set breakpoints, single-step, alter registers, or read and write RAM.

With this option enabled, the UniLab will write into the "reserved area" of your emulation memory every time you start up the analyzer with reset enabled. Press **CTRL-F3** to get the current location of the movable reserved area for your processor-specific DEBUG software.

When to use

Disable this option for completely transparent emulation. You will still be able to set triggers, examine traces, read and write emulation ROM, etc.

This option is disabled by **EMCLR**.

The explicit use of any DEBUG command, such as **RB** or **NMI**, will automatically enable this option.

Chapter Seven:
UniLab Command Reference

Contents

Overview

This chapter contains the reference material for the UniLab command language, a rich, flexible language that you use to work on your microprocessor control board and software.

The brief first section of this chapter classifies the commands as one of six types:

1. Beginner-- the minimal vocabulary you need to talk to the software.
2. Common-- commands that you will quickly learn and use often.
3. Advanced-- useful commands which might not be necessary for your work.
4. Special key and mode panel-- commands that perform the same function as one of the mode panel switches or one of the function or cursor keys.
5. Rarely used-- commands that you will probably be able to live without 98% of the time, but will appreciate during the remaining 2%.
6. Macro only-- commands that are only available in a macro system. See the **MACRO** and **OPERATOR** entries for more information.

The second of the two sections makes up the bulk of this chapter. It contains anywhere from a paragraph to a page or more of reference material for each command. The first page of this section explains the format of the entries, then the entries follow.

Display the entry for a command by typing:

HELP <command>

Appendix A is an alphabetical listing of all commands.

Processor specific commands

Every disassembler/DEBUG software package includes commands that are specific to that package. These words are not documented in this chapter. To learn more about target-specific words, consult the Target Application Note for your software package.

The Categories

The Orion software provides you with access to commands that let you:

set triggers on any input or combination of inputs,
alter the display and logging features,
set breakpoints and alter registers.

You will be able to do most of your work with just a few commands: the beginner and common words, with occasional use of advanced words.

The Special Key and Mode Panel words are listed mainly to help you find out more about the features that you would usually call up with a function key. Of course you can, if you want to, enter the command instead of using the special keys.

The rarely used commands are all very useful words, which cause rarely needed effects. But when you do need to do anything from see the binary equivalent of a hex number to filter only on the control column values, you will find them helpful.

Lastly, the "macro only" commands are not available until you use the command **MACRO.** The macro system also has access to some of the internal variables of the UniLab control program.

Consult Appendix F and the UniLab Programmer's Guide for more information.

The command **MACRO** converts the software into a macro system.

1. Beginner-- the minimum vocabulary you need to talk to the software, and learn more with the on-line help.

On-Line Help

HELP	MENU	MESSAGE
PINOUT	WORDS	

Load the Simple Target Program

LTARG

Predefined Trigger

STARTUP

Set Trigger on Address and Start Analyzer

AS

Exit the Program

BYE

Memory Enable

EMENABLE

Memory Reading

DN	MODIFY

Status Enquiry

ESTAT	TSTAT

DEBUG Words

N	RB	NMI

2. Common-- additional commands you will quickly learn and use often.

Predefined Triggers

ADR?	CYCLES?	EVENTS?
NOW?	SAMP	

Clear Out Previous Trigger

NORMB	NORMM	NORMT

Clear Out Previous Trigger, Prepare for Filtered Trace

ONLY

Set Up Trigger Spec

ADR	ALSO	ANY
DATA	NOT	TO

Set Up Trigger Spec-- Not supported on all processors

(Consult the Target Application Note.)

FETCH	READ	

Start Analyzer

S	S+	

Call DOS DOS

Trace Display Commands

TCOMP	TD	TMASK
TN	TSAVE	TSHOW

DEBUG Commands

GW	RI	SI
TRAM	TRAM'	

Stimulus Generator Commands

RES	SET	STIMULUS

Memory Reading

DM	MCOMP	

Line-by-line assembler

ASM	ASM-FILE	

Memory Writing

MFILL	M	M!
MM	MM!	ORG

Symbols

IS	SYMFILE	SYMLIST
SYMLOAD		

Save Information

BINSAVE	TOFILE	SAVE-SYS
SSAVE	SYMSAVE	TSAVE

Load Program From File

BINLOAD	HEXLOAD	

Examine Text File TEXTFILE TX

Initialize Instrument INIT

3. Advanced words-- commands that you will sometimes find useful, but can live without.

Create macro system MACRO

Summon menu of processors supported PATCH

On-Line Displays
CATALOG ASC

Program Performance Analyzer (optional feature)
AHIST HLOAD HSAVE
MHIST SOFT THIST

Trigger Commands
CONT DCYCLES HADR
HDATA LADR MASK
MISC

Start the Analyzer Repeatedly SR

Filter the Trace
1AFTER 2AFTER 3AFTER

Define a Pre-Qualifier
AFTER

Trace Display Commands TNT
DEBUG Commands
=OVERLAY G GB
RZ
Multiple Breakpoints
CLRMBP DMBP RMBP
SMBP
Symbols
CLRSYM SYMDEL SYMFILE+
SYMTYPE
Disable Emulation Memory EMCLR
Memory reading
M? MM? MDUMP
MMOVE RES-
Assign a Function to a Function Key
ALT-FKEY CTRL-FKEY FKEY
SHIFT-FKEY
Show Source File on Screen in Trace
MAPSYM MAPSYM+ SOURCE
SOURCE'
Display and Change RAM Allocated to Screen History and to Symbols
?FREE =HISTORY =SYMBOLS
Calculate a CheckSum CKSUM

4. Special key and mode panel words-- commands that perform the same function as one of the mode panel switches, or the same as the cursor keys or a function key.

Mode Panel Access
MODE

Mode Panel 1

DASM	DASM'
SYMB	SYMB'
RESET	RESET'

Mode Panel 2

SHOWM	SHOWM'
SHOWC	SHOWC'
=MBASE	
PAGINATE	PAGINATE'
HDG	HDG'

Mode Panel 3

LOG	LOG'
TOFILE	TOFILE'
PRINT	PRINT'
NMIVEC	NMIVEC'
RSP	RSP'

Function Keys

ALT-FKEY?	CTRL-FKEY?	DEFW
FKEY?	MEMO	SHIFT-FKEY?
SSAVE	WSIZE	

Cursor Keys
TOP/BOT

5. Rarely used words-- commands that you will probably not often use.

Trigger Commands

ASEG	INT	INT'
NDATA	SC	

Standalone Trigger Search

SST	TS

PreQualifier Commands

INFINITE	PCYCLES	PEVENTS
Q1	Q2	Q3
QUALIFIERS	TRIG	

Filter Manipulation Commands

CONTROL	FILTER	HDAT
MISC'	NO	

Emulation Memory Segment Enable =EMSEG

Temporary Number Base Change

B#	B.	D#
H>D		

Special Display Characteristic Commands

CLEAR	CLEAR'	COLOR
SET-COLOR		

Serial Port Setting

19.2K	9.6K	AUX1
AUX2		

Receive HEX format file from another system
HEXRCV

Symbol File Format SYMFIX

PROM Burning Mode

8BIT	16BIT

Loading from Host RAM MLOADN

Timing commands

=WAIT	MS

128K UniLab Only-- bank switch between 64K banks

PAGE0	PAGE1

Burn PROMs in Standalone Mode

STANDALONE	PROMMSG

6. Macro only -- commands that are only available after you use the configuration word **MACRO.**

In addition, other commands and variables are available to the user of the macro system. See Appendix F and the UniLab Programmer's Guide for details.

Convert to an Operator system
MAKE-OPERATOR OPERATOR

Macro Definition
: ; BPEX
BPEX2

For use with auto-test systems
‹TST›

THE UniLab COMMANDS

Entry format

The First Line

The first item on the first line of each entry is the command itself, always printed in bold capital letters. The rest of the line always contains either the phrase "no parameters" or the command repeated along with its parameters.

The parameters always appear inside <pointy brackets>.

Some words will have a last entry on the first line, which indicates that it belongs to some special category of commands. There are four types of commands that are marked this way:

1) commands which correspond to a function key (category four in the previous section) are marked with the function key number.
2) commands in the rarely used category (category five in the previous section) are marked "RARELY USED."
3) commands only available in macro system (category six in the previous section) are marked "MACRO SYS."
4) commands associated with the optional Program Performance Analyzer are marked with "PPA."

The Definition

The first block of text tells you what the command does.

Usage

The next block of text tells you how and when you use the command-- sometimes warning you that you only want to use the word in extraordinary circumstances.

Example

Almost every command includes a section showing examples of how to use the word.

Comments

This optional section includes warnings, historical notes, and various other bits and pieces of information.

16BIT no parameters RARELY USED

Selects 16-bit mode for memory emulation and for trace display and for PROM burning and reading.

USAGE

You will probably not use this command. It sets up the UniLab to work with processors that have a 16 bit data bus. If you have purchased a disassembler, then either this command or **8BIT** has been "built-in" to your software.

COMMENTS

16BIT is one word with no space after the 16. The 16BIT command changes both the signals put onto the target system's bus by the UniLab and the way the UniLab displays the trace display. That means you need a 28 pin ROM emulation cable, or the 16 bit emulation will not work.

The **HL** and **LH** commands determine the order in which the trace displays the bytes. The byte order has already been set for you by your UniLab DDB software.

19.2K no parameters RARELY USED

Changes bald rate of serial port to 19.2 K baud. This is the default.

USAGE

Use after **9.6K** to restore the default condition.

You must **SAVE-SYS** to make the change permanent.

1AFTER 1AFTER <trigger spec>

Clears out previous trigger spec and enables trace filtering. Only the bus cycle that satisfies the trigger spec and one cycle immediately after will be kept.

USAGE

The UniLab stores the trigger cycle and the one immediately after, every time it sees conditions that match the trigger specification. The "trigger status display line" shows how many cycles have been stored away.

You have to use **S** to start the analyzer after setting this trigger spec.

The UniLab automatically displays the trace after the entire trace buffer has been filled.

The disassembler will not work properly on fragments of code. The disassembler should be disabled with **DASM'** while you are looking at the results of any of the **xAFTER** commands.

CHECKING THE TRACE

If you want to see the trace before the buffer has been completely filled, then press any key to stop the cycle recording. Then type in **TD** to dump the trace, and display part of it on the screen.

The trace buffer fills from the bottom, and each new cycle pushes up the already recorded data. If you end up with a partially filled buffer, then the cycles you want to see are in the last part of the buffer.

EXAMPLES

1AFTER 1200 ADR S
shows only those cycles with adr =1200 and one cycle following.

1AFTER 235 TO 560 ADR S
shows 2 consecutive cycles each time a cycle has an address between 235 and 560.

(continued on next page)

(continued from previous page)

COMMENTS

Do not put a space between the number and **AFTER.** **1AFTER** is a single word, not a word preceded by a parameter. This command can be used when seeking the cause of a memory cycle error. It will show the program address of the cycle after the one that caused the memory access. **xAFTER** initializes all trigger features, so **NORM**x is unnecessary with these commands.

2AFTER 2AFTER <trigger spec>

Same as **1AFTER** except that two cycles are kept immediately following each trigger cycle.

USAGE

Enables a filtered trace that gives you a little more information than **1AFTER** does.

COMMENTS

See **1AFTER.**

3AFTER 3AFTER <trigger spec>

Same as **1AFTER,** except that the three cycles after the trigger cycle get stored.

USAGE

Enables a filtered trace that gives you a little more information than **2AFTER** does.

COMMENTS

See **1AFTER.** And notice that this filtered trace will contain enough information to make a disassembled trace sensible-- sometimes.

8BIT no parameters RARELY USED

Selects 8-bit mode for trace display and memory emulation and for PROM burning and reading.

USAGE

You will probably not use this command. It sets up the UniLab to work with processors that make use of 8 bit data. If you have purchased a disassembler, then either this command or **16BIT** has been "built-in" to your software.

COMMENTS

Use the 24 pin ROM cable with this command. **8BIT** is one word, with no space between the number 8 and the rest of the command.

9.6K no parameters RARELY USED

Changes baud rate of serial port to 9.6 K baud. The default is 19.2 K baud.

USAGE

Use to lower the baud rate for communication to the UniLab. Be certain to toggle the switch in the UniLab as well. The switch is inside the UniLab case, toward the back of the board, next to the plug-in for the connection to the host computer.

See also **19.2K**, for restoring the default.

You must **SAVE-SYS** to make the change permanent.

: no parameters Macro Sys

The colon character starts a macro definition. The word that follows the colon is the name of the macro.

USAGE

Once a macro has been defined, you can execute any lengthy series of commands with a single word. See Appendix F for further information. See also **BPEX.**

You will need to create a macro UniLab system (use the command **MACRO**) before you can define a macro.

WHAT A MACRO IS

A macro is a command that you create out of previously defined commands. For example,

: LOADUP 0 TO 3FFF BINLOAD A:MYPROG ;

creates a macro called **LOADUP**, which uses the previously defined UniLab command **BINLOAD.**

LOADUP will always load from a file on drive A: called myprog. You can see how this would be easier than using **BINLOAD** every time you wanted to load this file.

HOW TO WRITE MACROS

A macro definition begins with a colon and ends with a semicolon (**;**). The first word after the **:** is the name of the macro, and all the other words are the definition of it.

There must be at least one space between the colon and the name of the macro, and at least one space between the last word and the semicolon. Like this:

: NAME FIRSTWORD SECONDWORD VALUE THIRDWORD ;

FORTH

When you define a macro, you are actually making use of the programming language FORTH. With this powerful language you can define new words that make use of conditional statements, looping, and more. The best introduction to the language is Leo Brodie's Starting FORTH.

(continued on next page)

(continued from previous page)

WHY MACROS

The example below defines a macro called READRAM. After the new word has been defined, you would just type in READRAM every time you want to set up the trigger specification that shows only the cycles that read from the address range 1000 to 1FFF. This will save you a lot of keystrokes.

EXAMPLE

```
: READRAM  ONLY READ 1000 TO 1FFF ADR S ;
```

defines a macro called READRAM.

COMMENTS

Whenever the word immediately following : is entered the result is the same as if the rest of the words up to ; were entered. After typing in the example above, the word READRAM will have the same effect as entering " ONLY READ 1000 TO 1FFF ADR S ." To preserve the macro definition, you must **SAVE-SYS** before leaving the UniLab program.

See also appendix F.

; no parameters Macro Sys

Ends a macro definition started by : .

<TST> <value> ' <TST> ! Macro Sys

Setting this constant to 1 will turn off the output of some messages and will leave on the stack many results which are, normally, printed out.

USAGE

Use during testing procedures, or for sophisticated macros. When <TST> is set to one, such words as MM? will leave their results on the stack, rather than printing them out.

Remember to set <TST> back to zero when you are through.

EXAMPLE

: NEWMM? (addr--val) 1 ' <TST> ! MM? 0 ' <TST> ! ;
This macro will act the same as MM?, except it will leave the word value it finds on the stack rather than printing it out.

=BC <word> =BC

Changes the contents of the BC register to n.

USAGE

An example of the type of register control command available with a DEBUG package. This command addresses the Z-80 internal register BC. Consult the Target Application Note for your processor-specific software.

EXAMPLE

1234 =BC
puts 1234 in the BC register.

COMMENTS

You can use the register commands only after DEBUG has gained control of your microprocessor. See **NMI** or **RB** for more information on debug control.

This is a typical register changing instruction format. A similar command is provided for each of the processors internal registers (except SP). No space appears between the = and the register name.

=EMSEG <hex digit> =EMSEG RARELY USED

Sets A16-A19 context for subsequent **EMENABLE** statement(s). Determines which 64K "bank" of memory the emulated ROM will be in.

YOU PROBABLY DON'T NEED TO BOTHER

This value **must** be set properly, or the UniLab will not put the program opcodes onto the target system bus. This variable is already set properly for each disassembler/DEBUG software package.

WHY IT MIGHT MATTER

Though the upper 4 bits of our 20-bit address bus are meaningful only with processors that can address more than 64K of memory, **=EMSEG** must always be set.

On some microprocessors, those four lines are floating high, on other mp's several of the lines are pulled low.

HOW IT WORKS

This command only sets a variable. **EMENABLE** is the command that actually enables memory.

WHEN IT MATTERS

The UniLab looks at the upper 4 bits of address (A16 through A19) during fetch and read cycles, to determine whether your microprocessor wants to fetch an instruction from emulation ROM. If the upper 4 bits that the UniLab sees don't match the **=EMSEG** specification, then the UniLab will not respond to the mp's request.

Use **ESTAT** to see how this command effects the settings of emulated memory.

EXAMPLES

7 =EMSEG
sets A19 to 0 and A16, A17, and A18 to one.

(continued on next page)

(continued from previous page)

F =EMSEG 0 TO 1FFF EMENABLE
enables addresses F0000 to F1FFF.

E =EMSEG 0 EMENABLE ALSO F =EMSEG 0 EMENABLE
enables emulation of addresses
E0000 - E07FF and F0000 - F07FF.

COMMENTS

The 4 most significant bits of the 20 bit UniLab enable addressing are selected with =EMSEG so that subsequent statements only refer to 16-bit addresses. EMENABLE commands enable emulation memory in blocks of 2K.

A read or fetch command from the target microprocessor will reference emulation memory only when the A16-A19 inputs agree with an =EMSEG statement and A11-A15 indicate an enabled 2K block of emulation ROM.

Inputs A16-A19, displayed on the trace display as the right-hand digit of the CONT column, are the values seen by the emulation enable logic. If the inputs are not connected then they will "float," and appear as all 1s (hex F).

=EMSEG itself has no effect on the UniLab until an **EMENABLE** or **INIT** sends the data to the UniLab.

=HISTORY <hex# of Kbytes> =HISTORY

Selects the size of the screen history saved during each session with the UniLab.

USAGE

Allows you to change the amount of host RAM dedicated to saving information that scrolls off the top of the screen. The maximum is hexadecimal 3C Kbytes (decimal 60).

The new setting will not take effect until you **SAVE-SYS**, exit from the UniLab software, and start it up again.

Use **?FREE** to find out how much is allocated right now.

WHY CHANGE

You might want to have a longer history, or you might want to free up some of the host RAM for other purposes.

EXAMPLE

3C =HISTORY
allocates the maximum space to the line history.

=MBASE <n> =MBASE F8

Selects number base for the trace display of the MISC inputs to the UniLab, M0 through M7.

USAGE

The miscellaneous inputs (MISC) to the UniLab usually get displayed in binary format. This format allows you to easily tell which MISC inputs are receiving a high signal, and which are receiving a low.

This command also changes the number base for the HDATA column for 8 bit processors.

However, you might have an application for these inputs, such as reading the data from onboard RAM, where a hex or decimal display would be more useful.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

The panel only toggles between binary and hex.

EXAMPLES

10 =MBASE
hexadecimal display, the most space efficient

8 =MBASE
selects octal display mode.

A =MBASE
selects decimal display mode.

2 =MBASE
returns to binary display mode.

COMMENTS

The MISC inputs can be connected to any signals you like.

Notice that **A**, not **10**, must be used to specify decimal ten.

=OVERLAY <address> =OVERLAY

Changes the address of the area in emulation ROM used by the ORION DEBUG software.

USAGE

When your software uses the memory reserved by the ORION DEBUG software. This command changes the location of both the overlay and the reserved area. Use function key **CTRL-F3** to find the current address of the reserved and overlay area.

You can instead disable the DEBUG features for completely transparent operation. See **RSP'**.

You must **SAVE-SYS** to make the change permanent.

EXAMPLES

2310 =OVERLAY

Moves the reserved area to start at 2310, and puts the overlay area above there.

COMMENTS

The overlay area should not cross a 2K boundary. Be careful when changing its location.

=SYMBOLS <hex # of Kbytes> =SYMBOLS

Selects the amount of space allowed for symbol tables within the UniLab software.

USAGE

Allows you to change the amount of host RAM dedicated to storing the symbol table. The maximum is hexadecimal 80 Kbytes (decimal 128).

The new setting will not take effect until you **SAVE-SYS**, exit from the UniLab software and start it up again.

Use **?FREE** to find out how much is allocated right now.

WHY CHANGE

You might want to have a larger symbol table, or you might want to free up some of the host RAM for other purposes.

EXAMPLE

80 =SYMBOLS
make the symbol table the maximum possible size.

=WAIT <time> =WAIT

Changes the number of milliseconds that the UniLab software will wait between resetting the processor and checking for the processor clock.

USAGE

When you need a longer wait after reset. The default value is 140 (hexadecimal).

You must **SAVE-SYS** to make the change permanent.

EXAMPLE

280 =WAIT
Sets the wait time to double the default value.

?FREE no parameters

Displays the amount of host RAM allocated to the screen history and to the symbol table. Also shows how much host RAM is currently free.

USAGE

Find out how much you can increase the amount of space dedicated to history or symbol table, or whether you need to reduce it. See **=HISTORY** and **=SYMBOLS**.

ADR

<word> ADR
<word> TO <word> ADR

Sets up the trigger specification for analyzer inputs A0 through A15. (Sets trigger for A0 to A19 if five-digit address ends in a period.)

USAGE

Determines which 16 bit addresses the analyzer will trigger on. Can also trigger on 20-bit addresses.

With **TO** the trigger will occur on the address range from ADR1 to ADR2.

If **NOT** precedes the value(s) of the address, the UniLab will trigger outside of the specified address or range of addresses.

All previous entries to the address trigger spec are erased unless you precede this spec with the word **ALSO.**

You can inadvertently produce "cross products" when making use of **ALSO** with **ADR.** See the fourth example below.

EXAMPLES

NORMT 1023 ADR S
trigger on address 1023. **NORMT** causes the trigger to appear at the Top of the trace.

NOT 120 TO 455 ADR S
trigger if address outside 120-455 range.

12345. ADR S
trigger on 20-bit address 12345. The 1 will appear in right digit of the CONT column.

1200 ADR ALSO 8 ADR
sets the analyzer to trigger when the address is 1200 or 0008. Because of cross products, will also trigger on address 0000 and 1208.

(continued on next page)

(continued from previous page)

COMMENTS

ALSO must be used with caution with ADR. Generally you can use ALSO once, if the high-order byte of the previous spec and the new one match. To do more than that you should work with the two bytes of the address separately using **HADR** and **LADR.**

AS is a convenient abbreviation for **NORMT ADR S.**

You can define a 20-bit address trigger by ending the number in a period. See **ASEG** for another approach to 20-bit addresses.

ADR? no parameters

Displays random examples of the addresses seen on the bus--approximately two every second.

USAGE

This command displays two of the addresses that appear on the bus each second. A useful command for getting a rough-grained idea of how the program behaves.

Terminate the display by pressing any key.

EXAMPLE

ADR?

This command is never used in combination with anything else.

COMMENTS

Useful for monitoring program flow in a rough manner. For example, it will be obvious to you if the target program gets stuck in a loop. ADR? turns RESET mode off and sets up a trigger spec of its own. Be sure to use **NORMx** at the start of the first trigger spec after using this word.

AFTER AFTER <qualifier specification>

Sets the stage for the description of a qualifying event. Qualifying events are bus states that must be seen before the analyzer starts to search for the trigger.

USAGE

When you have specified qualifying events, the UniLab will not recognize the trigger until after the "qualifiers" have been seen.

You can set up to three qualifying events. Each qualifier spec must start with **AFTER.**

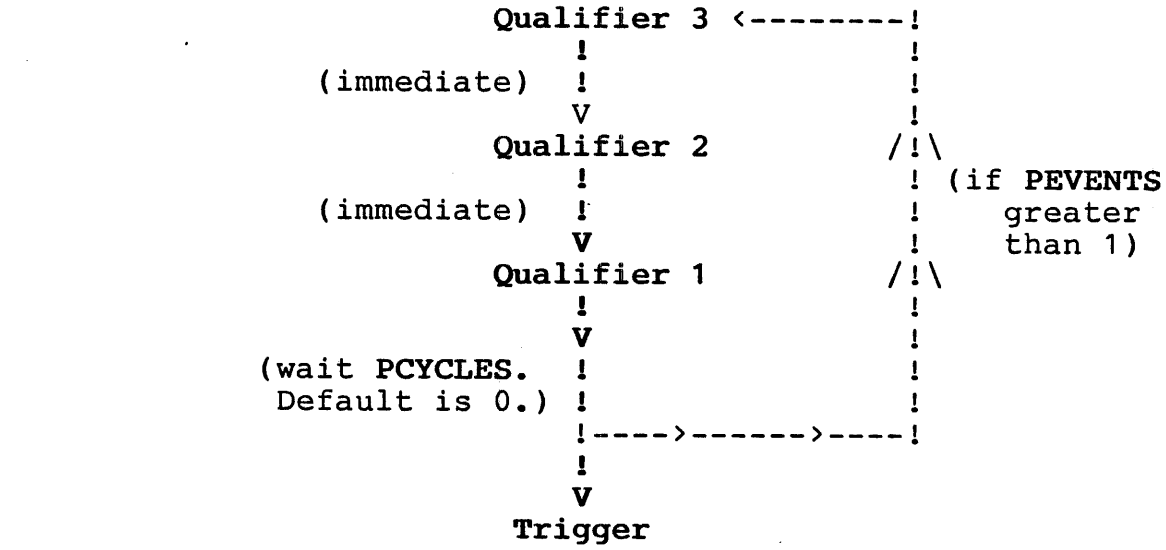
All the qualifiers must appear on the bus one immediately after another, without intervening bus cycles. However, the trigger itself can appear anytime after all the qualifiers have been satisfied.

You cannot use MISC inputs as qualifiers.

DELAYS AND REPETITIONS

You can specify a minimum number of bus cycles after the time the last qualifier is seen, before the UniLab starts looking for the trigger. See **PCYCLES.** The default is **0 PCYCLES.**

You can also specify a number of complete repetitions of the sequence of qualifiers. See **PEVENTS.** The default is **1 PEVENTS.**



(continued from previous page)

EXAMPLES

NORMT 100 ADR AFTER 535 ADR S
will trigger on address 100 only after address 535 gets seen on the bus.

AFTER 3F DATA S
You can add a second qualifying event-- which must occur earlier than the first. Now address 535 must be immediately preceded by data 3F hex before UniLab will look for address 100 on the bus.

NORMT 100 ADR AFTER 535 ADR AFTER 3F DATA S
a single statement with the same result as the two above.

NORMT AFTER NOT 345 ADR AFTER 344 ADR S
triggers if any address other than 345 follows immediately after 344. By starting with **AFTER** we are able to describe two events which must follow one another without intervening bus cycles.

COMMENTS

Equivalent results can be obtained by using <n> **QUALIFIERS** to set the number of qualifiers. The four related commands **TRIG, Q1, Q2,** and **Q3** can then be used to set the various triggers. But **AFTER** is the more natural way to do it.

You will find **Q1,** etc., handy when you want to "change context" to alter the description of an event that you though you had completed.

AHIST no parameters PPA

Address **HIST**ogram invokes the optional Program Performance Analyzer (PPA), which allows you to display the activity of your target program in each of up to 15 user-specified address ranges. See also **MHIST** and **THIST.**

USAGE

Allows you to examine the performance of your software. You can find out where your program is spending most of its time.

Press **F10** to exit from this menu-driven feature.

You must (only once) issue the command **SOFT** to enable this optional feature. **SOFT** performs a **SAVE-SYS**, and then causes an exit to DOS. The next time you call up the software, the PPA will be enabled.

MENU DRIVEN

You produce a histogram by first specifying the upper and lower limits of each address "bin" that you want displayed, then starting the display.

When you give the command **AHIST** you get the histogram screen with the cursor positioned at the first bin. You can then start typing in the lower and upper limits of each bin. Use return, tab or an arrow key after you enter each number, to move to the next entry field.

Press function key 1 **(F1)** to start displaying the histogram.

SAVE TO A FILE

You can save the setup of a histogram as a file with the **HSAVE** <file>. Issue this command after you exit from the histogram.

You load the histogram back in with **HLOAD** <file>. This command also invokes the histogram.

EXAMPLE

AHIST

This command is never used in combination with anything else.

ALSO no parameters

Used with both **EMENABLE** and with trigger specification commands. Prevents clearing of previous settings.

USAGE

The trigger spec commands, **CONT, ADR, DATA, HDATA, HADR, LADR** and **MISC**, normally cause the UniLab to trigger on the new conditions instead of the old conditions. By using **ALSO**, you can instruct the UniLab to trigger on the old conditions OR the new conditions.

The memory enable command, **EMENABLE**, normally enables only the new settings of memory. By using **ALSO**, you can enable both the old range of memory and the new.

You have to use **ALSO** for each new setting that you declare. See the second example below.

ALSO is not necessary when you want to trigger on several different categories. The UniLab will automatically AND together the specifications in different categories.

You can inadvertently produce "cross products" when making use of **ALSO** with **ADR**. See **ADR**.

EXAMPLES

12 DATA ALSO 34 DATA
sets the analyzer to trigger on either 12 or 34 data (without the ALSO only 34 data would remain set).

10 DATA ALSO 5 DATA ALSO 3 DATA 1200 ADR
sets the analyzer to trigger when the data is 10 or 5 or 3 and the address is 1200.

0 TO 7FF EMENABLE ALSO 2000 TO 2FFF EMENABLE
enables two ranges of emulation ROM.

COMMENTS

Applies only to the first **EMENABLE** or trigger spec command that follows.

ALT-FKEY <# of key> ALT-FKEY <command>

Assigns a command to an **ALT**ered function key.

USAGE

Reassign the function keys on PCs and PC look-alikes. Use **ALT-FKEY?** (or press F1 while holding down ALT) to find the current assignments.

The function keys allow you to execute any command or string of commands with a single keystroke. The initial assignments represent our best guess at what you will need. But you might want to change them.

To make your reassignments permanent, use **SAVE-SYS.**

EXAMPLE

2 ALT-FKEY WSIZE
assigns WSIZE to ALT-F2.

COMMENTS

To execute a string of commands, define a macro first (using :) and then assign the macro to the function key.

See also **FKEY, CTRL-FKEY,** and **SHIFT-FKEY.**

ALT-FKEY? no parameters ALT-F1

Displays the current assignments of the **ALT**ered function keys.

USAGE

Whenever you want to be reminded what command will be executed when you press a function key while holding down the ALT key.

See **ALT-FKEY** to reassign the keys.

ANY ANY <input group>

Sets a trigger spec that will trigger on any value on the input group.

USAGE

Provides a way to "clear out" the trigger on any selection of input groups. This can sometimes save you the trouble of re-entering a trigger spec.

This command is most appropriate after you have entered and used a trigger spec, but now want to use a broader trigger spec.

EXAMPLE

ANY CONT
trigger when any value appears on the CONT input lines. The rest of the trigger spec remains unchanged.

COMMENTS

The macro definition of this command:

```
: ANY     0 TO FFFF ;
```

AS <addr> AS

An abbreviation for **NORMT ADR S.**

USAGE

Defines an analyzer trigger spec, and starts the analyzer working. The trigger event appears near the top of the trace as cycle zero. A useful abbreviation-- saves you key strokes. When entering the most common trigger spec-- triggering on a code address.

Will not work on ranges of addresses (with **TO**) or with **NOT.**

EXAMPLE

1234 AS
triggers when address is 1234

COMMENTS

The macro definition of this command:
: AS NORMT ADR S ;

ASC no parameters SHIFT-F4

Displays the handy reference ASCII table.

USAGE

Shows each character, along with its decimal and hex value.

EXAMPLE

ASC

This command is never used in combination with anything else.

COMMENTS

This is a bonus feature provided to save you the trouble of hunting for a printed ASCII table.

ASEG <hex digit> ASEG RARELY USED

Sets a trigger spec on address bits A16-A19. **ASEG** cannot be used with NOT, ALSO, or TO.

USAGE

Normally, you set a trigger address with **ADR**, either a 16 bit or 20-bit address. This command allows you to set a trigger on the upper 4 bits of the 20 bit address. See **=EMSEG** for a longer discussion of the addressing scheme of the UniLab.

EXAMPLES

5 ASEG
requires a hex value of 5 on A16-A19 for trigger.

COMMENTS

Normally useful only if you have over 64K of memory in your target system. Even then, a better way to define a trigger on a 5-digit address is just to enter the 5-digit address ending in a period followed by ADR.

The command "n ASEG" has the same effect as "F MASK n CONT."

ASM <address> ASM <instruction>

Invokes the processor-specific line-by-line assembler.

USAGE

Patch assembly language code to the given address in emulation ROM. Allows you to overwrite locations in the copy of your target program residing in the UniLab's emulation ROM, so that you can quickly fix bugs when you find them. The assembler writes over memory-- it does not insert instructions.

If you do not include the address, **ASM** will use the current value stored by the **ORG** command.

ASSEMBLING MULTIPLE INSTRUCTIONS

If you do not include an assembly language instruction, then **ASM** will give you as a prompt the address to which it is assembling, and wait for you to give it an instruction followed by a carriage return.

The assembler will continue to prompt you with an address and patch assembled code into memory, until you feed a blank line (press return on an empty line).

CONVENTIONS

The line-by-line assembler will only accept assembly language instructions, not ORIGIN statements or EQU statements. (You should use the UniLab command **IS** to define symbols.)

Only one instruction per line.

The normal conventions of assembly language apply. For example, at least one space between the instruction and the operands.

The Target Application Note contains a section listing the instruction set recognized by the assembler.

(continued on next page)

(continued from previous page)

EXAMPLES

0 ASM LD SP,3000
alters the first instruction of the LTARG program of the Z80 package.

100 ASM
invokes the assembler, starting at address 100. The assembler will prompt you with that same address, and wait for you to enter an assembly language instruction.

ASM-FILE <addr> <start screen> <end screen> ASM-FILE

Invokes a version of the line-by-line assembler that assembles code contained on the screens of a FORTH file.

USAGE

A way to make large patches to your program, or to write prototype code without leaving the UniLab environment-- or just to write a few lines that you will want to be able to edit and re-enter.

ASM-FILE follows the same conventions as **ASM**.

You can include comments on a screen by putting a semicolon (;) on a line. The assembler will ignore everything after the semicolon on that line. The semicolon must be the first character on the line, or be preceded by at least one space.

FORTH FILES AND THE EDITOR

If you only have a few lines of code, you can use the screen that **MEMO** puts you into, and the two following (screens 1D through 1F). See the entry for **MEMO** to get a few pointers on using the FORTH screen editor.

OPENING A NEW FILE

You will want to put the code into a file of its own if you have many lines of code, or if you want a more convenient way to archive the code. You must make a MACRO system before you can use the file commands.

First close the current file (**UniLab.SCR**) with the command **CLOSE**.

Next create a new file with **OPEN-NEW** <file name>, and determine its size with <# of screens> **SCREENS** (1K allocated per screen). Use the command <screen #> **EDIT** to get into the file. Don't make use of screen zero.

You will then be able to use **ASM-FILE** to assemble the code stored in your new file.

When you are done with assembling, use **OPEN UNILAB.SCR** to close your file and re-open the UniLab.SCR file. If you don't do this, then some of the on-line help facilities and error messages will not work.

(continued on next page)

(continued from previous page)

EXAMPLES

1200 1D 1F ASM-FILE
loads assembly code, starting at address 1200, from screens 1D through 1F of the currently opened FORTH file.

1 4 ASM-FILE
loads code from screens 1 through 4, starting at the current value of **ORG**.

AUX1 no parameters RARELY USED

Tells the host computer to look for the UniLab on serial port 1. This is the normal default condition.

AUX2 no parameters RARELY USED

Tells the host computer to look for the UniLab on serial port 2. Only use this command if you have the UniLab connected to serial port 2.

B# B# <binary number> RARELY USED

Interprets the number following as a binary number.

USAGE

Useful when you want to input a number as a binary--saves time with pencil and paper. Quick, what is the hex value of a number with 1 at locations 0, 3, 7, 9 and 10? Let the computer do that work for you.

EXAMPLES

B# 0101010001001
has the same effect as entering 0A89H

NORMT B# 1111110 MISC S
will trigger when the MISC inputs are 11111110

COMMENTS

Changes the base to binary, just for the next number. Allows entering numbers in binary format, just as **D#** allows decimal format.

B. <hex number> B. RARELY USED

Displays the hex number as a binary number.

USAGE

When you want to find out the binary equivalent of a hex number, saves you time with pencil and paper.

EXAMPLE

A89 B.
displays the binary equivalent of A89, which is 0101010001001.

BINLOAD <from addr> <to addr> BINLOAD <filename>

Loads a binary file from disk into emulation memory. Prompts you for the name of the file if you don't include it on the command line.

USAGE

Starts loading a binary file into the **from addr.** Stops loading at the **to addr,** or when end of file is reached. The binary file should contain a program. Can be used to load the product of a cross compiler into emulation memory.

This command fully supports DOS pathnames.

You can save a program to a file with **BINSAVE.**

EXAMPLE

0 400 BINLOAD \ASM\MAIN.BIN
loads a binary DOS file, starting at location 0 and ending at location 400.

COMMENTS

Loads exact binary contents of file until DOS indicates end of file, or the "to address" is reached. If you don't know the ending address, you can just enter FFFF as toadr and loading will stop on end-of-file.

As with all memory writing commands, don't write into your stack area when loading into RAM.

Use with .COM,.BIN, or .TSK files. See **HEXLOAD** for Intel Hex files.

The Orion software can load to target RAM as well. See **NMI** and **RB.**

BINSAVE <1st addr> <2nd addr> BINSAVE <file name>

Saves the specified section of memory as a file. Prompts you for the file name if you do not include it.

USAGE

This command saves the program memory to disk. Saves everything in memory between the first address and the second address.

This command fully supports DOS pathnames.

EXAMPLE

100 4FF BINSAVE
saves target locations 100 - 4FF.

COMMENTS

Saves exact binary contents of a range of target memory as a named file. This file can later be re-loaded with the **BINLOAD** command.

Can save from target RAM as well. See **NMI** and **RB**.

BPEX BPEX <macro name> Macro Sys

Executes the specified macro at each breakpoint, after the register display.

USAGE

Allows you to automatically execute any command or group of commands, at every breakpoint. You must first define a macro, or use one of the pre-defined Orion command words.

BPEX will not accept a string of commands, only the first word that follows. This means that only certain commands are suitable-- those that require no parameters. In the example below, we first write a macro that requires no parameters, called **SEE-RAM**. Notice that **SEE-RAM** makes a call to **MDUMP**, which does require parameters.

See : for more info on macros.

TURN IT OFF

To turn off the automatic execution use **BPEX NOOP**.

EXAMPLES

: SEE-RAM 8000 8080 MDUMP ;
defines a macro called SEE-RAM which dumps out 80 memory locations.

BPEX SEE-RAM
executes your macro at every subsequent breakpoint.

COMMENTS

Available only with DEBUG packages. Useful if, for example, you want to watch a memory window as you single step through the program.

BPEX2 BPEX2 <macro name> Macro Sys

Execute a second macro at each breakpoint. See **BPEX**.

BYE no parameters

Exits from UniLab program.

USAGE

To return to DOS. Use **SAVE-SYS** first, if you want to save the current state of the system.

Use **DOS** instead if you want to execute just a few DOS commands and then return to the UniLab program.

EXAMPLE

BYE

This command never used in combination with anything else.

CATALOG no parameters

Displays a directory of all the available pinouts-- the proper cable hook-ups for each microprocessor.

USAGE

Once this word is entered, any of the listed pinouts can be displayed on the screen.

This word "opens" the pinout library. It closes again as soon as you enter another command.

Until you use this command, the only pinout diagram available is that of the mp you are using. You get that with the command **PINOUT.**

CKSUM <from addr> <to addr> CKSUM

Calculates the checksum for a given range of memory. Useful for error-checking.

USAGE

A good way to make a PROM easy to check for burn-in errors, or corrupted locations. Allows you to record the checksum of your program-- or better yet, make the checksum equal to zero.

EXAMPLE

800 FFF CKSUM
prints a 16-bit checksum for the data in addresses 800-FFF

COMMENTS

You may want to patch the complement of this value into your PROM. You can produce a PROM with a checksum of zero, using the following method, which sacrifices only two bytes.

First store zero where the checksum will be (0 FFE MM! in the above example). Second, find the checksum, using **CKSUM**. Lastly, patch in the complement of the sum.

For example, if the sum is 1234, then use the command **-1234 FFE MM!**. The resulting PROM will have a checksum of 0.

CLEAR no parameters RARELY USED

Clears the screen before performing a **PgUp.** Use with some of the older color monitor cards, that will otherwise flicker when you use **PgUp.**

CLEAR' no parameters RARELY USED

The normal default condition-- the screen is not cleared before a **PgUp** is executed. Use only to restore the default condition after executing a **CLEAR.**

CLRMBP no parameters

Clears all multiple breakpoints.

USAGE

Use to wipe the slate clean, and start out setting multiple breakpoints again. **SMBP** sets the breakpoints.

EXAMPLE

CLRMBP

This command never used in combination with anything else.

COMMENTS

Use to clear all the numbered breakpoints which you set with **SMBP** and can clear one at a time with **RMBP**.

CLRSYM no parameters

Clears out the current symbol table.

USAGE

When you want to get rid of the symbols that you have defined for your program. It's a good idea to first save the symbols, just in case you decide you want those symbols after all. See **SYMSAVE.**

The symbol table also gets cleared by **SYMFILE** and **SYMLOAD** before they load in the new symbols. **SYMFILE+** adds to the existing symbol table.

Unless you save the symbols, you cannot recover them later. You could instead use **SYMB'**, which turns off the symbol table without erasing it.

EXAMPLE

CLRSYM
This command never used in combination with anything else.

COMMENTS

You might want to clear out the table before loading in a new one from a file. See **SYMFILE** and **SYMLOAD.**

COLOR no parameters RARELY USED

Displays in color. Only has an effect with a color monitor.

USAGE

Turns on color display.

You have to save the system afterward, if you want the UniLab program to start up with color display.

CHANGING COLORS

Use the UniLab command **SET-COLOR**, which shows you what the new settings are as you change them.

You will have to save the system with **SAVE-SYS** if you want to preserve the new colors.

EXAMPLES

COLOR

This command never used in combination with anything else.

COM1 no parameters

Enables dumb terminal emulation mode, using serial communications port 1 of your personal computer. This is the port normally used by the UniLab.

USAGE

Allows you to use your PC as a dumb terminal while within the UniLab software. Press the ESCape key to exit.

COMMUNICATION SETTINGS

The default settings are:

300 baud
8 bits, 2 stop bits, no parity.

CHANGING SETTINGS

You can change these settings by changing the values stored in two constants, **BR2** (Baud Rate) and **LCR2** (Line Control Register: bits per character, etc.).

Put the value 60 into **BR2** to change to 1200 baud:

```
60 ' BR2 !
```

You may miss characters at 1200 baud, due to the screen scroll times. Put a 180 into **BR2** to change back to 300 baud.

You can change to 7 bits, 2 stop bits with:

```
6 ' LCR2 !
```

TABLE OF SETTINGS

# bits	parity	#stop bits	value to store at LCR2
7	None	1	2
7	None	2	6
7	Odd	1	A
7	Odd	2	E
7	Even	1	1A
7	Even	2	1E
8	None	1	3
8	None	2	7
8	Odd	1	B
8	Odd	2	F
8	Even	1	1B
8	Even	2	1F

(continued on next page)

(continued from previous page)

To change to 5 or 6 bits per character look at the information on the Line Control Register of the INS8250 in a reference manual on that chip, or in the Hardware Technical Reference Manual for your computer.

COM2 no parameters

Enables dumb terminal emulation mode, using serial communications port 2 of your personal computer. See the entry for **COM1** for details.

USAGE

Allows you to use your PC as a dumb terminal while within the UniLab software. Press the ESCape key to exit.

Change the communications settings the exact same way that you do for **COM1**.

CONT <byte> CONT RARELY USED
<byte> TO <byte> CONT
<byte> MASK <byte> CONT

Sets up the analyzer trigger spec for the CONT inputs (control lines C4 - C7, and A16 - A19).

USAGE

The CONT input lines actually represent two different types of information. The upper four bits represent the processor cycle type. The lower four bits come from the four highest address lines, A16 through A19.

When you precede it with one number, CONT causes the UniLab to trigger when the inputs equal that number. When you use **TO** the UniLab triggers on any value from **m** to **n**. **NOT** causes the UniLab to trigger when the value falls outside of the specified range or value.

You can use **k MASK l** to examine any subset of the 8 input lines. See Comments below for more details.

Unless you use **ALSO** the previous trigger spec gets cleared out.

EXAMPLES

B# 00011111 CONT
requires C7-C5 = 0, C4 & A19-A16 = 1.

70 TO 7F CONT
requires C7=0 and C6-C4 = 1, A19-A16 any value.

F MASK 3 CONT
requires A19 & A18 = 0, A17 & A16 = 1, C7-C4 any value.

(continued on next page)

(continued from previous page)

COMMENTS

The low four bits of the CONT lines refer to the highest four bits of the address-- the same segment address bits set by =EMSEG.

When you use the command k MASK l CONT, the value of k determines which bits the UniLab will examine-- the bits with a value of one. The l then indicates the value those lines must have before trigger occurs.

For example, **F0 MASK AF** tells the UniLab to only look at the upper 4 bits of the CONT lines. The **AF** tells the UniLab to trigger when bits 7 and 5 are high while bits 6 and 4 are low. The UniLab will "not care" about the value of the lower four bits.

CONTROL no parameters RARELY USED

Used before **FILTER** to set up a filter spec based only on the CONT inputs.

USAGE -- RARELY USED

You will probably never use this command. Triggers on the full specification, but filters based only on the 8 bits of the CONT inputs.

The filter mechanism of the UniLab gets turned on for you by the **xAFTER** macros. Those commands set the filter to **MISC' FILTER**, which allows you to set up a trigger spec based on all inputs except for the MISCellaneous wires.

See also **HDAT**, **MISC'**, and **NO**.

THE CONT INPUTS

The upper four bits identify processor cycle type, while the lower four bits identify the address bits A19-A16.

This command makes it possible to filter on cycle type and on memory segments.

EXAMPLE

NORMT CONTROL FILTER WRITE 1200 ADR A7 DEVENTS S
triggers on 1200 address, and then records only writes. You have to use **DEVENTS** to get a trace buffer full of the event you are filtering on.

CTRL-FKEY <# of key> CTRL-FKEY <command>

Assigns a command to a function key pressed while the CTRL key is held down.

USAGE

Reassign the function keys on PCs and PC look-alikes. Use **CTRL-FKEY?** (or CTRL-F1) to find the current assignments.

The function keys allow you to execute any command or string of commands with a single keystroke. The initial assignments represent our best guess at what you will need. But you might want to change them.

To make your reassignments permanent, use **SAVE-SYS.**

EXAMPLE

5 CTRL-FKEY DOS
assigns DOS to CTRL-F5.

COMMENTS

To execute a string of commands, define a macro first (using :) and then assign the macro to the function key.

See also **FKEY, ALT-FKEY,** and **SHIFT-FKEY.**

CTRL-FKEY? no parameters CTRL-F1

Displays the current assignments of the ConTroLled function keys.

USAGE

Whenever you want to be reminded what command will be executed when you press a function key while holding down the CTRL key.

See **CTRL-FKEY** to reassign the keys.

CYCLES? <from addr> <to addr> CYCLES?

Counts the number of bus cycles between two addresses.

USAGE

Can use to count the number of bus cycles in a loop, as in the first example below, or the "distance" between two addresses.

BUS CYCLE COUNT

Not the number of machine cycles, nor the number of instructions fetched, but instead the number of reads and writes that occur between one command and another. The read could be instruction fetches, or could be data fetches.

EXAMPLES

123 CYCLES?
counts cycles between two occurrences of the address 123.

123 456 CYCLES?
counts cycles between address 123 and address 456.

12300. 12450. CYCLES?
counts cycles between address 12300 and address 12450.

COMMENTS

Useful for checking quickly whether a loop works as you intended. **CYCLES?** makes its own trigger spec, so you will have to start fresh on your trigger after using this command. Use one of the **NORM**x commands to clear out the trigger spec set by **CYCLES?**.

When specifying a five-digit address, the . which designates a five-digit address must be used with both addresses.

D# D# <decimal number> RARELY USED

Treats the number that follows as a decimal value, rather than as a hexadecimal, which is the default.

USAGE

Saves you the trouble of converting the number by hand or with a calculator.

EXAMPLES

D# 16 ADR
equivalent to entering "10 ADR".

D# 32 .
will display 20 (the hex equivalent of 32 decimal).

D# 135 B.
converts a number from decimal to binary.

D# 1000 MS
will pause 1 second.

COMMENTS

See also **B#** for entering binary numbers.

DASM no parameters F8

Enables the trace disassembler.

USAGE

Turns on the translation of machine code into assembly language mnemonics. You will usually want to keep this on, only turning it off for special applications such as x**AFTER**. To turn off the disassembler, use **DASM'**.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

EXAMPLE

DASM
selects disassembled mode for trace display.

COMMENTS

Works only if you have an optional disassembler installed.

DASM' no parameters F8

Disables the trace disassembler.

USAGE

Turns off the translation of hexadecimal machine codes into assembly language mnemonics. See **DASM** above for more details.

Typically you will use the **MODE** panel (function key 8) when you want to change this feature.

EXAMPLE

DASM' selects hex mode for trace display.

DATA

<byte> DATA
<byte> TO <byte> DATA
<byte> MASK <byte> DATA

Changes the analyzer trigger for the DATA inputs (D0 to D7).

THE DATA INPUTS:
The UniLab gets both the address and the data from the bus during each memory read and write. The "data" that appears on the bus could be either a value or a machine code instruction. See COMMENTS below for information on triggering on a 16-bit data bus.

USAGE
The simplest use sets up a trigger for a single data value. The UniLab will search for the byte value, and trigger when it sees that hex number on the bus as data. See the first example below.

RANGES OF DATA:
TO lets you set up a trigger on any data between two byte values, inclusive. See the second example below.

NOT
NOT causes the UniLab to trigger when the value falls outside the specified range or value.

MASKING
You can use **k MASK 1 DATA** to examine any subset of the 8 data lines. The high bits of k mark which bits will be examined, while the bit configuration of byte 1 indicates the values the lines must have for a trigger to occur.

For example, **80 MASK FF DATA** selects only the highest data bit for examination (with binary value 1000 0000). The UniLab would trigger when this bit has a high value. The instruction **80 MASK 80 DATA** would have the same effect.

(**DATA** continued on next page)

(continued from previous page)

EXAMPLES

NORMT 12 DATA S
after clearing all previous settings with NORMT, sets up a trigger for data input 12, and then uses S to start the analyzer.

12 TO 34 DATA
requires data value between 12 and 34 hex.

F0 MASK 30 DATA
sets a trigger based only on the four highest bits of data. UniLab will look for a 3 on those lines.

23 DATA ALSO 45 DATA
sets a trigger on cycles where data is either 23 or 45 hex.

COMMENTS

The data inputs (D0-D7) are normally connected via the emulator cable at the ROM socket. On 16-bit processors DATA is only half of the data bus, while **HDATA** is the other half.

If you need to use a large number of ALSO terms, then see **NDATA**.

DCYCLES <number of cycles> DCYCLES

Sets number of cycles the UniLab will continue to record after the trigger.

USAGE

When the UniLab sees the trigger event on the target board, it consults the delay cycles variable to determine how many more cycles to record. Each time a new cycle enters the trace buffer you lose the oldest recorded cycle. After the UniLab records the specified number of cycles, it shows the trace buffer on the screen.

WHY YOU DON'T NEED TO BOTHER

This command gets executed by a number of other commands. **NORMT**, for example, sets the delay value to A0 (160 decimal). That delay count puts the trigger event near the top of the trace buffer, after the ten cycles that came just before it.

WHY YOU MIGHT WANT TO

You might want to see the trace starting 260 cycles after a known event-- perhaps you don't know where the program ends up at that time. The **DCYCLES** command will do the job perfectly.

EXAMPLE

104 DCYCLES
selects 104 (hex) delay cycles (260 decimal)

COMMENTS

NORMT, **NORMM**, and **NORMB** select A0, 55, and 4 DCYCLES respectively. **S+** increases the number of delay cycles by A6, so you can see what happens after the end of the current trace.

The maximum possible delay count is FFFF.

DEFW no parameters F5

Returns the window to the size last set with **WSIZE**, or to the default if you have not changed the window size.

USAGE

The help screens readjust the window size, to make the lower window as large as possible without overwriting the information in the upper window. After you have used a help screen, you might want to return the DEFault Window size. Just press Function key 5.

SAVING A DEFAULT

You can use **SAVE-SYS** to save all the current settings at any time.

EXAMPLES

DEFW

This command never used in combination with anything else.

DM <start address> <count> DM

Disassembles <count> number of lines, starting at the given address.

USAGE

A very useful tool for examination of memory. Allows you to see what instructions are in emulation memory and in RAM as well. See also **DN**.

Can give misleading results if you give an address that is not the first address of an opcode, but even then will generally come "into sync" after a few instructions.

EXAMPLE

100 10 DM
disassembles 10 lines starting at address 100

COMMENTS

Normally disassembles from ROM. Works only if you have an optional disassembler. Can disassemble from target RAM as well. See **NMI** and **RB**.

DMBP no parameters

Displays the settings of all eight multiple breakpoints.

USAGE

When you forget the settings of your multiple breakpoints. Automatically executed whenever you set one of the 8 multiple breakpoints with **SMBP**.

EXAMPLES

DMBP
This command never used in combination with anything else.

DN <start address> DN

Disassembles from memory into the right-hand side of current window. Displays one instruction per line and fills the right hand window.

USAGE

When you want to keep the disassembly from memory on the screen while preforming other operations. This command is similar to **DM**, except that it writes into a portion of the screen that is only used for this feature.

The disassembly produced by **DN** does not scroll off the screen. You can get rid of it by using **F2** to get out rid of the split screen and then scrolling the screen (or, if currently looking at an un-split screen, press **F2** twice).

EXAMPLES

20F0 DN
Fills the right side of the current window with assembly language code, starting from address 20F0.

DOS DOS <DOS command>

Execute a DOS command from the UniLab program.
Or, use with no parameters to exit to DOS temporarily. Return to UniLab program by typing EXIT.

USAGE

When you forget the name of the file where you stored that program, or have any other reason to use the DOS utilities. You can either execute a single command, or you can go to DOS and execute a series of commands.

If you go to DOS, you can re-enter the UniLab program. Return to the UniLab program by typing **EXIT** at the DOS prompt (A> or B> or C>).

If you use **BYE** to exit the UniLab program, you have to start it up again by typing **ULxx** at the DOS prompt.

EXAMPLES

DOS DIR /w
Executes the DOS command "**DIR /w.**"

DOS
Allows you to execute any series of DOS commands, then return to the UniLab program.

DOS ASMB SOURCE.ASM OBJECT.BIN
Assembles a new version of the program you are working on.

EMCLR no parameters

Tells the UniLab not to emulate ROM-- clears out the emulation memory settings.

USAGE

Commands the UniLab to not respond to any microprocessor requests for data or instructions. Use only when you want to run a program from on-board ROM.

This word also disables the DEBUG features. To turn them on again for use with emulation ROM, use the SWI VECTOR choice on the mode panel (**F8**) or **RSP**.

Instead of running the program from a chip, you can use the PROM READER MENU (**F9** from the MAIN MENU) to read a program into emulation memory from most ROM chips.

EXAMPLE

EMCLR

This command never used in combination with anything else.

EMENABLE <address> EMENABLE
<from address> TO <to address> EMENABLE

Enables emulation memory, needed before you can load in a program. But first, set **=EMSEG** properly.

USAGE

With a single address, enables the 2K memory region that includes the given address. **=EMSEG** just sets a variable in the host's memory, while **EMENABLE** sends all the information to the UniLab.

You can use **SAVE-SYS** to make the current settings permanent.

ON RANGES OF ADDRESSES

TO enables the emulation memory from the beginning of the 2K segment that includes the <from> address to the end of the 2K segment that the <to> address is in.

CLEARING PREVIOUS SETTINGS

Unless you precede emulation statement with **ALSO**, clears out previous **EMENABLE** statements.

WATCH OUT

When you try to emulate two separate ranges of memory, you can accidentally overlay the two. For example, with a 32K UniLab, 0 and 8000 reference the same memory location in the UniLab.

Of course, you can enable areas that do not overlap. For example, 0 TO 3FFF and also C000 TO FFFF would not conflict.

EXAMPLES

F =EMSEG O EMENABLE
enables target addresses 0-7FF with A16-19 all set high.

0 TO 1FFF EMENABLE ALSO FFFF EMENABLE
enables 0-1FFF and F800-FFFF

F =EMSEG O EMENABLE ALSO E =EMSEG O EMENABLE
enables locations F0000 - F07FF and E0000 - E07FF

(continued on next page)

(continued from previous page)

COMMENTS

The UniLab's enable logic first compares the A16-A19 value from the most recent **=EMSEG** statement with the present bus address. Address inputs A11-A15 then get compared to an enable map, where each entry corresponds to a 2K segment of memory. When both the segment and the 2K block are enabled, the UniLab accepts the address, and puts its data on the bus.

ESTAT no parameters

Tells you the current status of emulation memory.

USAGE

When you want to find out what range of addresses is currently enabled.

EXAMPLES

ESTAT

This command never used in combination with anything else.

EVENTS? no parameters

Starts the analyzer and counts occurrences of the currently defined trigger event.

USAGE

Useful for monitoring occurrences that you don't need a trace of. An excellent way to see whether the program does what it should. If the program messes up spectacularly, or performs flawlessly, then this command will show you that.

Otherwise, you're left in the dark.

EXAMPLES

NORMT 123 ADR EVENTS?
counts occurrences of address 123.

NORMT 123 ADR FF DATA EVENTS?
counts occurrences of data FF when the address is 123.

NORMT WRITE 78 TO FF DATA 1210 ADR EVENTS?
Counts the number of times a data value greater than 78 gets written to location 1210.

COMMENTS

You can also count such things as error conditions or system usage.

You can use this command if you want to sync a scope on the UniLab's test point output.

FETCH no parameters

Tells the UniLab to look for trigger event only during fetch cycles.

USAGE

To search for a particular opcode. After you give it this command, the UniLab will not look for the trigger event during reads or writes.

This command is not available on all processors.

This command is used as part of a trigger spec, as shown in the examples below.

EXAMPLES

NORMT FETCH 120 ADR S
triggers when the program fetches from address 120.

NORMT FETCH NOT 0 TO 7FF ADR S
triggers if the program tries to fetch an instruction from outside the 0 to 7FF range.

COMMENTS

This command, loaded with the disassembler, specifies a range of CONT values corresponding to fetch cycles.

FILTER no parameters RARELY USED

Selects trace filtering mode, according to previous word: **CONTROL, HDAT, MISC'** or **NO.**

WHY YOU DON'T NEED TO BOTHER

For most filtering of the trace, you will use commands such as **ONLY** or **xAFTER.** These words automatically select the **MISC'** filtering mode for you. The **NORMx** words turn off filtering.

You can use this command to set up a filter spec that is different from your trigger spec. This is sometimes a very useful thing to be able to do.

EXAMPLE

NORMT CONTROL FILTER READ 1200 ADR A7 DEVENTS S
triggers when the processor reads from address 1200-- then produces a filtered trace of the A7 (hex) read cycles that occur after that.

COMMENTS

You would want to bother when inventing your own filtering command.

FKEY <# of key> FKEY <command>

Assigns a command to a function key.

USAGE

Reassign the function keys on PCs and PC look-alikes. Use **FKEY?** (or F1) to find the current assignments.

The function keys allow you to execute any command or string of commands with a single keystroke. The initial assignments represent our best guess at what you will need. But you might want to change them.

You have to use "A" (hexadecimal) as the number to assign a command to **F10.**

To make your reassignments permanent, use **SAVE-SYS.**

EXAMPLE

```
2 FKEY STARTUP
```

assigns STARTUP to the F9 key.

COMMENTS

If you find yourself performing some one action repeatedly, you can save time by making it into a macro and then assigning it to a function key. For example

```
: DUMP100   0 100 MDUMP  ;
6 FKEY DUMP100
```

will allow you to dump locations 0 to 100 by pressing function key 6.

See also **ALT-FKEY, CTRL-FKEY,** and **SHIFT-FKEY.**

FKEY? no parameters F1

Displays the current function key assignments.

USAGE

Whenever you want to be reminded what pressing a function key will do for you.

See **FKEY** to reassign the keys.

EXAMPLES

FKEY?

This command never used in combination with anything else.

G <address> G

Goes to the indicated address. Exits DEBUG control, lets the target run.

USAGE

After you have set a breakpoint, and want to release debug control and let the target run. **G** is one of several ways to do this.

G just gets the target board going. After that, you can enter a trigger spec and restart the analyzer, or you can use one of the "big picture" words: **ADR?**, **SAMP**, or **NOW?**.

You could instead use **STARTUP** to restart the analyzer and the board at the same time. Or use **NORM**x followed by a trigger specification and **S**, to restart the analyzer and give you a trace of the event that you describe.

EXAMPLE

1030 G
exits from debug control, and resumes the target program at location 1030.

COMMENTS

Appropriate if you have a DEBUG package and have established control by entering **RESET adr RB**, or **NMI**. You can return to any point in the program you like, but debug control will be lost.

Use **GB** if you wish to resume the program at an address different from the one you are stopped at but with another breakpoint set.

GB <addr to go to> <bpoint addr> GB

Goes to the first address, and starts executing code, with a breakpoint set at the second address.

USAGE

When you want to move around the program without losing debug control.

EXAMPLES

1200 330 GB
resumes the program at address 1200, with a breakpoint set at 330.

COMMENTS

Available if you have an DEBUG package and have established DEBUG control. See **RB** to establish DEBUG control.

GW <address> GW

Goes to the indicated address and waits until the analyzer is started. Releases the target board from DEBUG control.

USAGE

To continue the execution of the program, starting at the given address, after a new trigger spec has been defined.

A rather specialized but very useful command.

EXAMPLE

1100 GW NORMT 1200 ADR S
Goes to address 1100 and waits for the analyzer to be started. The trigger spec command sets the analyzer to capture a trace showing the code at address 1200.

H>D <hex number> H>D RARELY USED

Displays the decimal equivalent of a hex number.

USAGE

Shows you the decimal equivalent-- compare this with **D#,** which allows you to enter a decimal number that will then be used by the next command.

This word is similar to **B.** which shows you the binary equivalent of a hex number.

EXAMPLE

10 H>D
will cause "16" to be displayed.

333 133 - H>D
will display "512," which is the decimal equivalent of 333 minus 133 (hex).

HADR < byte > HADR RARELY USED
< byte > TO < byte > HADR
< byte > MASK < byte > HADR

Changes the analyzer trigger for the high-order byte of the 16-bit address (A8-A15).

THE ADDRESS INPUTS

You should normally use **ADR** to set 16 or 20 bits at once, but there are limits to the use of **ALSO** in combination with **ADR.**

The UniLab gets both the address and the data from the bus during each bus cycle. The UniLab works with up to 20-bit addresses. You can change the trigger specification of the least significant byte with **LADR,** the second byte with this command, and the high four bytes with **CONT** or **ASEG.**

USAGE

You can use this trigger spec command in the same way you use **DATA, CONT,** etc.. However, the most frequent use of this command is to set up a trigger spec on the address lines that makes use of many calls to **ALSO.**

EXAMPLES

NORMT 12 HADR ALSO 34 LADR ALSO 10 LADR ALSO 5 LADR
sets up the analyzer to trigger on any of the addresses 1234, 1210 or 1205.

COMMENTS

Makes it possible to treat the first two bytes of the address separately. **LADR** is the lower half.

HDAT HDAT FILTER RARELY USED

Used before **FILTER** to set up a filter spec based only on the high byte of the DATA inputs (D8 - D15).

USAGE -- RARELY USED

You will probably never use this command. Triggers on the full specification, but filters based only on the 8 bits D8 through D15.

The filter mechanism of the UniLab gets turned on for you by the x**AFTER** macros. Those commands set the filter to **MISC' FILTER**, which allows you to set up a trigger spec based on all inputs except for the MISCellaneous wires.

See also **CONTROL, MISC** and **NO.**

THE HIGH DATA INPUTS

These lines read from the high byte of the 16-bit data path of 16-bit processors. On 8-bit processors, the lines can be left to float, or be used to sense other logic signals on your target board.

USAGE

Used to show only the cycles that meet your description. While deciding whether to include the current cycle in a filtered trace, the UniLab will check only these 8 bits of the 48 inputs.

A good way to look at all the bus cycles that have some specific data value as the upper byte of data.

EXAMPLE

NORMT HDAT FILTER 80 TO FF HDATA 3410 ADR A7 DEVENTS S
will give a trace showing only those cycles with D15 high, starting with the bus cycle that has D15 high and address 3410. You have to use **DEVENTS** to get a trace full of the event you are filtering on.

HDATA < byte > HDATA
< byte > TO < byte > HDATA
< byte > MASK < byte > HDATA

Changes the analyzer trigger for the high byte of 16-bit data path (D8 through D15). Spare inputs on 8-bit processors.

THE DATA INPUTS

The UniLab gets both the address and the data from the bus during each bus cycle. The "data" that appears on the bus could be either a value or a machine code instruction. On 8-bit processors the inputs D8 through D15 can be hooked up to anything you like.

USAGE

The simplest use sets up a trigger for a single value on the high order byte of the data inputs. The UniLab will search for the byte value, and trigger when it sees that hex number on the bus as data.

Note that just looking at the high order byte means the UniLab doesn't care about the low order byte, and so it actually searches for a range of values. See the first example below.

To specify just one full 16 bit wide data value, you must use both **HDATA** and **DATA**.

RANGES OF DATA

TO lets you set up a trigger on any data between two byte values, inclusive. See the third example below.

NOT

NOT causes the UniLab to trigger when the value falls outside the specified range or value.

MASKING

You can use **<i> MASK <j> HDATA** to examine any subset of the 8 most significant data lines. The high bits of i mark which bits will be examined, while the bit configuration of byte j indicates the values the lines must have for a trigger to occur.

For example, **01 MASK FF HDATA** selects only data bit D8 for examination (with binary value 0000 0001). The UniLab would trigger when this bit has a high value. The instruction **01 MASK 01 HDATA** would have the same effect.

(**HDATA** continued on next page)

(continued from previous page)

EXAMPLES

NORMT 12 HDATA S
after clearing all previous settings with NORMT, sets up a trigger for data input 12XX -- actually 1200 through 12FF-- then uses S to start the analyzer.

12 HDATA 80 DATA
sets a trigger for only data 1280.

12 TO 34 HDATA
requires data value between 12XX and 34XX hex. That is, 1200 through 34FF.

F0 MASK 00 HDATA
sets a trigger based only on the four highest bits of data. UniLab will look for a 0 on those lines.

12 TO 23 HDATA ALSO 45 HDATA
sets a trigger on cycles where the highest byte of data is either 12 to 23, or 45 hex.

COMMENTS

You must use a special 16-bit cable with processors that use a 16-bit data bus. That cable has two ROM plugs-- one for the even byte, one for the odd byte.

If you need to use a large number of ALSO terms, then see **NDATA.**

The HDATA inputs are named for their use in the 16BIT mode. In the 8BIT mode they are displayed as a separate column and can be used as for anything you like just like the MISC inputs. On eight- bit systems they are typically used to look at system inputs and outputs.

HDG no parameters F8

Has a fixed header for trace displays-- one that does not scroll up with the rest of the trace.

USAGE

One of the display attributes. Usually you will toggle this with the mode panel, function key 8.

HDG" no parameters F8

Makes a non-fixed header for trace displays-- one that scrolls with the rest of the trace.

USAGE

One of the display attributes. Usually you will toggle this with the mode panel, function key 8.

HELP HELP <command> F1

Finds the reference information for a command or feature. With no word, brings up the help screen, including soft-key prompt line.

USAGE

Look up information on a command, in the abridged on-line command reference. See also **WORDS.**

EXAMPLES

HELP
displays help screen.

HELP BYE
gives information on command "bye."

HEXLOAD HEXLOAD <file name>

Loads an Intel HEX format object file into the UniLab's emulation memory. Prompts you for the file name if you don't include it.

USAGE

Load into emulation memory a program stored in Intel HEX format. You can then run, debug and alter that program as you would any other.

Binary format files are more compact and load two to three times faster. You might want to direct your assembler to produce binary format files, if it has that capability. Or you can save your program memory with **BINSAVE** to produce a binary format file.

Binary format files are loaded with **BINLOAD.**

Intel HEX format files contain the information about where each opcode should be stored. Be certain to have the proper sections of emulation memory enabled before loading in the file. See **EMENABLE.**

LOADING INTO RAM

The UniLab will not load a file into RAM unless you have first established debug control. To do that you must first have a program already loaded into emulation memory (**LTARG** for example) and then run to a breakpoint with **RESET** <address> **RB.**

If DEBUG is not in control, attempts to load memory that is not enabled will generate an "auto-breakpoint." You will see "-nmi-" and then the "target address-not enb" message. Enabled areas in the same file will be loaded.

EXAMPLE

HEXLOAD MYPROG.HEX
load an Intel HEX format file called MYPROG.HEX.

(continued on next page)

(continued from previous page)

COMMENTS

Only record types 0 to 3 are supported. Bytes 7 and 8 of each line of the file tell what record type that line uses.

16-bit processor note: If the UniLab detects a type 2 (extended address) record then address bits A16-A19 will be compared to the current **=EMSEG** and data will not be loaded if it is intended for some segment other than the current one. This will be indicated by a "not enb" message for each invalid address. Enabled addresses in the file will be properly loaded.

HEXRCV no parameters

Loads an Intel HEX file from another computer, via a second serial port.

USAGE

The serial transmission must be done on a separate serial channel with the UniLab connected to its normal serial port. XON and XOFF characters are used to start and stop the data transmission. Transmission is normally done on COM2 on IBM PC's while the UniLab is connected to COM1.

EXAMPLE

HEXRCV
loads a hex file serially

HLOAD HLOAD <filename> PPA

Loads from a file the data describing a histogram. This is used only with the optional Program Performance Analyzer. You save the information to a file with **HSAVE.**

USAGE

Loads into memory a PPA template or a run that you previously had saved, then automatically calls up **AHIST, MHIST** or **THIST.**

EXAMPLES

HLOAD AUG28.HST
load into memory the information in the file AUG28.HST, that had been saved with **HSAVE.**

HSAVE HSAVE <filename> PPA

Saves to a file the data describing a histogram. This is used only with the optional Program Performance Analyzer, after exiting from **THIST, MHIST** or **AHIST.** You load the information back into memory with **HLOAD.**

USAGE

Save in a file a Program Performance Analyzer template or a run that you want to keep for future purposes.

This feature is handy when you are periodically running a histogram of a program, and want to save the bin settings. It can also be used to save a particular run of the Program Performance Analyzer.

EXAMPLES

HSAVE AUG28.HST
save as a file the data that describes the last histogram screen you saw before exiting from either **AHIST** or **THIST.**

INFINITE INFINITE PEVENTS RARELY USED

Used only before **PEVENTS**, instead of a count, to indicate that the trigger event must immediately follow the qualifying events.

USAGE

Along with a trigger specification (see **ADR, DATA, READ, WRITE,** etc.) and a qualifying event specification (see **AFTER** or **QUALIFIERS**), when you are only interested in the trigger event if it occurs immediately after the qualifying events.

BACKGROUND

The default is for the UniLab to search for the qualifying sequence only once. After the sequence has been found once, it is discarded and the UniLab looks for the trigger.

With **PEVENTS** and a normal count, the UniLab searches for the qualifying events until it finds them the count number of times. Then it discards the qualifiers, and looks only for the trigger.

WHAT IT REALLY DOES

INFINITE causes the UniLab to search for the qualifying sequence and then immediately look for the trigger event. If the trigger event is not the very next cycle, then the UniLab starts looking for the qualifiers again.

EXAMPLE

123 ADR AFTER 345 ADR INFINITE PEVENTS
triggers if address 123 follows immediately after address 345.

COMMENTS

Pretty obscure. But might be highly useful in certain restricted situations.

Pressing any key stops the search.

INIT no parameters

Sends an initialization message to the UniLab.

USAGE

To reset the UniLab after you are in the UniLab program.

When you start up the program, it tries to initialize the instrument after the screen has been cleared and the UniLab version number displayed. If you tap any key after the screen is cleared, then the automatic init will not occur. You will then have to use **INIT** before you can send any commands to the instrument.

Also, if the UniLab was not properly connected when you called up the program, or if you turned off the UniLab at any time during the program, then the UniLab needs to be initialized.

IF IT FREEZES

If the program stops after printing the "Initializing UniLab. . . ." message, press the BREAK key while holding down the CONTROL key. This breaks you out of the initializing sequence. Make certain that you have turned on the UniLab and connected it to the host computer.

Try **INIT** again. If it still freezes up, check the **TroubleShooting** chapter.

EXAMPLES

INIT

This command is never used in combination with anything else.

COMMENTS

Initializes all of the mode bits, baud rate and emulation enable map. Sent automatically after PROM programmer operations to re-initialize the analyzer modes.

INT no parameters RARELY USED

Generates the NMI- interrupt output when trigger state reached. The NMI- wire from the UniLab must be connected to either NMI or IRQ circuit of processor.

USAGE-- RARE

Useful for causing the target system to execute an interrupt routine when it goes into trigger search state (i.e., after the "qualifier has been found). Used to prevent damage to equipment by branching control to a "soft shutdown" routine when some error condition occurs.

You must write and install your own shutdown routine.

Orion DEBUG packages use this command internally. If you want to make use of it, you must disable the **NMI** feature of the Orion software with the Mode Panel (**F8**) or with **NMIVEC'**.

NORMx disables the **INT** mode.

EXAMPLES

NORMT INT AFTER 123 ADR S
will interrupt the target processor during the bus cycle after address 123 is reached, then trigger immediately.

NORMT INT 12 DATA AFTER 345 ADR S
will interrupt during the bus cycle after address 345 occurs, then the analyzer will trigger when 12 data occurs.

COMMENTS

The interrupt occurs when the qualifying sequence is complete, not on trigger event. This makes it possible to trigger on something specific after the interrupt occurs.

INT' no parameters RARELY USED

Disables the **INT** mode.

USAGE

Rare.

COMMENTS

Not often used since **NORM**x also disables the INT mode.

IS <value> IS <name>

Assigns a symbol name to an address or data value.

USAGE

To show mnemonic names of memory locations on traces. If you already have an assembler generated symbol table, you will prefer to use the symbol table features of the UniLab. See **SYMFIX** and **SYMFILE.**

You can use the **IS** command to add symbols after you have loaded in a symbol table. **IS** turns on symbol display mode.

EXAMPLE

1234 IS MREGISTER
gives 1234 the symbol name "MREGISTER"

COMMENTS

Used to manually create a symbol table or to add symbols to an existing table.

Use **SYMB** to enable the symbol display on trace. See also **SYMB', SYMSAVE, CLRSYM, SYMLOAD,** and **SYMFILE.** Symbol translation will work with or without a disassembler.

LADR <byte> LADR RARELY USED
<byte> TO <byte> LADR
<byte> MASK <byte> LADR

Sets the truth table for the low order byte of the address (A0-A7) separately.

THE ADDRESS INPUTS

You should normally use **ADR** to set 16 or 20 bits at once, but there are limits to the use of **ALSO** in combination with **ADR.**

The UniLab gets both the address and the data from the bus during each bus cycle. The UniLab works with up to 20-bit addresses. You can change the trigger specification of the least significant byte with this command, the second byte with **HADR** and the high four bytes with **CONT** or **ASEG.**

USAGE

You can use this trigger spec command in the same way you use **DATA, CONT,** etc.. However, the most frequent use of this command is to set up a trigger spec on the address lines that makes use of many calls to **ALSO.**

LADR is also useful for setting a trigger on a port address of the Z80. The ports of the Z80 processor have only one byte addresses-- and the Z80 puts the contents of the A register on the upper byte of the address lines when it outputs to a port.

EXAMPLE

NORMT 12 HADR ALSO 34 LADR ALSO 10 LADR ALSO 5 LADR
sets up the analyzer to trigger on any of the addresses 1234, 1210, or 1205.

COMMENTS

Makes it possible to treat the first two bytes of the address separately. **HADR** is the upper half.

LOG no parameters F8

Enables automatic logging of target program patches on printer.

USAGE

Keeps a record of any program patches you make, but other operations are not logged to the printer.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

LOG' no parameters F8

Disables logging of program patches to printer. See **LOG** above.

USAGE

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

LP no parameters

Goes around a loop once and stops.

USAGE

You must already have established debug control (see **RB**), and be stopped at a breakpoint within a loop. This command allows the program to run once around the loop and stop at the current address, displaying the registers as the UniLab does for any breakpoint.

WATCH OUT

Will not work if the program counter register is pointing above the first instruction or below the last instruction in the loop. Only works when you are within the loop. See the Target Application Note for your processor for any additional restrictions.

EXAMPLES

LP

This command never used in combination with anything else.

COMMENTS

Works by first saving the current breakpoint address, executing **N** (a single step without following branches) and then executing ‹saved_address› **RB**. Processors with multiple byte breakpoint opcodes will execute **N** several times.

LTARG no parameters

Loads a simple target program into the UniLab's emulation memory.

USAGE

A good way to gain familiarity with the UniLab. Comes packaged with the disassembler. This command enables the proper section of emulation memory and loads a simple program. You can then use the **STARTUP** command to capture a trace of your target system executing the simple program.

WATCH OUT: PROCESSORS WITH EXTERNAL STACKS

The **LTARG** program uses the memory map of the Orion MicroTarget. If your target system does not have RAM and ROM where the **LTARG** program needs them, then it will not run on your board without some patching.

The Target Application Note for each DDB includes an **LTARG** sample session.

EXAMPLE

LTARG

This command never used in combination with anything else.

M <byte> M

Stores one byte in ROM or RAM and increments reference address.

USAGE

Used after an **ORG** statement (which sets up address), to patch program memory. Can only be used to change RAM after debug control has been established. See **RB**.

REMEMBERING CHANGES

LOG sends all memory access commands such as **M** to the printer, saving a record of any changes you make.

EXAMPLES

3000 ORG 12 M
stores a 12 at 3000

150 ORG 5 M 10 M
stores 5 at location 150, 10 at 151

COMMENTS

Used for entering data tables, program patches, etc. See also **MM**, **MM!**, and **M!**.

Will store to emulated memory if the address is enabled, otherwise will store to target RAM. See **NMI** and **RB**.

As with all memory writing commands, don't write into your stack area when loading into RAM.

M! <byte> <address> M!

Stores a byte of data at the specified address.

USAGE

Used to patch program memory. Does not require a previous **ORG** command-- instead requires an address as the second parameter. See **M.** Can also be used to change RAM, but only after debug control has been established. See **RB.**

REMEMBERING CHANGES

LOG sends all memory access commands, such as **M!**, to the printer, saving a record of any changes you make.

EXAMPLES

12 3000 M!
stores a 12 at 3000.

5 150 M! 10 150 M!
stores 5 at location 150, 10 at 151.

COMMENTS

Used for entering small patches-- anything larger than one byte can be done by one of the other memory patch commands with fewer keystrokes. See also **MM, MM!**, and **M.**

Will store to emulated memory if the address is enabled, otherwise will attempt to store to target RAM. See **NMI** and **RB.**

M? <address> M?

Displays the byte that is stored at the specified address.

USAGE

To find out what is stored at a single memory location, either ROM or RAM. Use **MM?** for looking at words, and **MDUMP** or **DM** for larger areas of memory.

EXAMPLES

1210 M?
displays the byte stored at 1210.

COMMENTS

If the address is **EMENABLE**d then emulation memory will be displayed, otherwise the UniLab will use DEBUG features to display target RAM contents. See **NMI** and **RB.**

MACRO no parameters

Switches the UniLab software to a macro system.

USAGE

Only necessary when you want to write macros or you need access to the "internal" words of the UniLab control program. For information on macros refer to the glossary entry on : (colon), and Appendix F of this manual. For information on internal commands and other subjects, order the UniLab Programmer's Guide from Orion.

Several otherwise unused files must be in the UniLab directory when you request that the software switch to macro system. Included on your distribution diskette are the files necessary both for the operator system, with a **.OPR** extension, and for the macro system, with a **.MCR** extension. You should have one **.OPR** and one **.MCR** file for every **.EXE** or **.OVL** file.

When you switch to macro system, the UniLab software will search for a **.MCR** file whose name matches the current **.EXE** file.

If you had previously saved the system to a different name (using **SAVE-SYS**), you will have to rename the **.MCR** so that it matches your executable file before you can make a macro system.

You can save the UniLab software as a macro system anytime after you use the command **MACRO**. **SAVE-SYS** will save, to the new name you specify, a **.EXE** file and a matching **.MCR** file.

In the operator system (see **OPERATOR** and **MAKE-OPERATOR**) you have access only to the commands in the UniLab glossary.

EXAMPLE

MACRO
Converts to macro system.

MAKE-OPERATOR MAKE-OPERATOR <file1> <file2> Macro Sys

Use this command to create an operator system that has restricted access to the UniLab program, but also has access to the words that you have defined.

USAGE

This command performs four actions:

1) save the macro system to the first name,
2) create a non-standard operator system,
3) save the new system to the second name, and
4) exit to DOS.

The new, non-standard operator system recognizes the new commands that you have defined in your macro system.

The standard operator system gives you access only to the commands in the UniLab On-Line Glossary.

FILE NAMES

When you create an operator system with this command your UniLab directory must contain the MAKE file from your distribution diskette.

EXAMPLE

OPERATOR MACROZ80 TESTER

Saves a macro system with the name MACROZ80, then creates an operator system and saves that software to the name **TESTER.EXE**, with associated file **TESTER.OPR**.

MAPSYM MAPSYM <filename>

Reads from a .MAP file the information the UniLab needs to provide high level language support. Clears out the symbol table before loading the information. See also **MAPSYM+**.

USAGE

Reads in from a .MAP file the information needed for display of high-level language source files in the trace. After you issue this command, each line of your source code file will be displayed just before the instructions that the line generated.

You can use **SYMLIST** to see the contents of the symbol table after you load a .MAP file. You can save the entire symbol table with **SYMSAVE**, and reload it later with **SYMLOAD**.

You must have your source files in the current directory, or they will not be found.

.MAP FILE FORMATS

You can use either a MicroSoft format .MAP file or an ORION format file, described below. The MicroSoft .MAP file contains a mixture of symbol and line number data.

The Orion format is much simpler, which makes it easier to generate a .MAP file. It contains only line number information.

ORION .MAP FORMAT

The Orion format .MAP file is an ASCII file which contains a series of file records, one for each source file. You can have any number of file records per .MAP file.

The first line of each record starts with the keyword SOURCE, followed by a space and then the name of the source file. The remainder of each file record contains two numbers per line: the line number, then the absolute 16 bit address of the code generated by that source line.

A file record is terminated by a blank line. The .MAP file is terminated by two blank lines. Every line of the .MAP file must end in a carriage return and a line feed (ASCII codes 0DH and 0AH).

(continued on next page)

(continued from previous page)

ORION .MAP FILE--EXAMPLE
The following is a simple example of a valid Orion format .MAP file. This file describes the relationship between source files and machine code for a simple C program. The program was generated from two source files. Notice that only some lines of the source file generated code:

```
SOURCE SIMPLE1.C
2 0034
5 0040
6 0050
                                                <blank line>
SOURCE SIMPLE2.C
3 0055
5 0070
                                                <blank line>
                                                <blank line>
```

DISABLE
You turn off the display of high level source files with **SOURCE'**.

EXAMPLE

MAPSYM TEST.MAP
loads into the symbol table the information in .MAP file TEST.MAP. The source files themselves are not opened until they are needed, while looking at a trace display or at a disassembly from memory.

MAPSYM+ MAPSYM+ <filename>

Same as **MAPSYM**, except that it does not clear the symbol table before loading the .MAP file.

MASK <byte> MASK <byte>

Specifies a mask for the trigger spec that immediately follows.

USAGE

A modifier to **ADR, CONT, DATA, HADR, HDATA, LADR,** or **MISC.**

The first byte describes which of eight wires to pay attention to-- a one means pay attention, a zero means don't care.

The second byte tells the UniLab what inputs to look for on the wires that you care about. The UniLab ignores the bits for the inputs that the first byte told it to ignore. Thus **01 MASK 01** has the same affect as **01 MASK FF.**

EXAMPLES

NORMT 2 MASK 2 MISC S
will trigger if input M1 goes high.

NORMT B# 0010 MASK B# 0010 MISC S
has the same effect as the first example-- will trigger if input M1 goes high.

NORMT 3 MASK 2 MISC S
requires inputs M1=1, MO=O for trigger.

COMMENTS

MASK cannot be used with TO, NOT, ALSO

MCOMP <start addr> <end addr> <comp addr> MCOMP

Compares two areas of memory and indicates discrepancies.

USAGE

Compares the two areas of memory, and gives you a message about each discrepancy. Press any key to abort. For example:

110 117 810 MCOMP

```
Data is 16 at addr 0110 ..but is  5 at addr 0810
Data is 90 at addr 0112 ..but is 80 at addr 0812
Data is 27 at addr 0116 ..but is 23 at addr 0816
```

You only need to enter three addresses-- the starting and ending address of the first block of memory, and the starting address of the second.

VERIFYING ROMS

If you want to compare a ROM to a program on disk, first load the program using **BINLOAD** or **HEXLOAD.** After that use the PROM READER MENU to read from the PROM into a different memory area.

You can then use **MCOMP** to compare the two target areas.

EXAMPLE

100 300 800 MCOMP

compares data at target addresses 100-300 to the data at 800-A00.

COMMENTS

Works on either emulated ROM or target RAM. See **NMI** and **RB.**

MDUMP <from addr> <to addr> MDUMP

Display the contents of an area of memory.

USAGE

Allows you to look at any block of memory. For example:

121 131 MDUMP

```
121   F3 31 00 1C 21 78 02 11  78 02 01 2C 00 7C BA C2   .1..!x..x..,. ..
131   38 01 7D BB CA 42 01 7E  12 23 13 0B 79 B0 C2 38   8...B. .#..y..8
```

Press any key to freeze scrolling of display. Press any key again to continue scrolling. While scrolling is stopped, press any key twice quickly to stop.

EXAMPLE

1234 1334 MDUMP
displays the contents of locations 1234 to 1334 in hex and ASCII.

COMMENTS

As with all M commands, display will be from emulation memory if the address has been EMENABLEd, otherwise DEBUG features will display target RAM memory. See **NMI** and **RB**.

MEMO no parameters SHIFT-F2

Displays and allows editing of the on-line memo pad.

USAGE

A handy way to write notes to yourself. Pressing CONTROL and Z at the same time toggles the on-line editor help screen on and off. This screen shows you the ESCape key sequences and ConTRoL key combinations that you use with the editor. See COMMENTS below.

You exit the full screen editor with ESCAPE followed by F if you want to save the changed memo pad. ESCAPE followed by ESCAPE allows you to leave the memo pad without saving your changes.

EXAMPLE

MEMO

This command never used in combination with anything else.

COMMENTS

This command works only when the EDITxx.VIR file is present in the same directory as the UniLab program.

The powerful editor allows you to write complicated macros and enable them at will. If you want to use this feature to the fullest, order the PADS manual from

Mountain View Press
PO Box 4656
Mountain View, CA 94040

(continued on next page)

(continued from previous page)

EDITOR HELP (repeated on-line):

Press **SHIFT-F2** to get the editor.
Once in the editor, press **CTRL-Z** to get the on-line help.

Press WHILE HOLDING DOWN THE CONTROL KEY:

CURSOR CONTROL:

S=Left	D=Right	
E=Up	X=Down	Q=Home
F=Rtab	I=Ltab	
F=Forwd	A=Bkwrd	

CHARACTER CONTROL:	LINE CONTROL:
Del=Delete char	K=Kill line
J=Jerk-->buffer	G=Gobble-->buffer
C=Chars<--buffer	Y=Copy-->buffer
V=Insert chars	L=Line<--buffer
P=Pullup words	N=New lines

Press THE ESCAPE KEY AND FOLLOW WITH:

ESC=Esc no update	F=Updat & Fin edit
W=Word for search	B=Updat & Back scr
S=Search screens	N=Update & Nxt scr
U=Update now	L=Update & Load
R=Restore screen	

MENU no parameters F10

Selects the menu-driven mode.

USAGE

The menu- driven mode helps first time users by allowing you to use the UniLab simply by choosing from list of options. This command, whether typed in or picked with function key 10, reassigns the function keys and shows the menu on the screen. The command line that you would use gets displayed as it is executed, so you can learn how to enter the command directly.

While using the menu, you can also type commands directly.

Menu mode also comes in handy when you have forgotten a command.

All PROM programming commands are available under the PROM menu.

Pressing F10 again from the main menu gets you out of menu mode.

EXAMPLE

MENU

This command never used in combination with anything else.

MESSAGE no parameters

Gives a screenful of information on the most recent updates and additions to the UniLab software.

USAGE

Make certain that you know all the capabilities of the UniLab software.

MFILL <from addr> <to addr> <byte> MFILL

Fills every location in an area of memory with the same byte.

USAGE

A good way to check that memory address and data lines connect properly on the target board. You can fill an area of memory, and then examine it with **MDUMP.**

One way to find out what is happening on your board when **LTARG** test program will not run: fill a block of memory with NOOP instructions, starting at the reset address, and then use **STARTUP.** You should see a trace of consecutive addresses.

Also a heavy-handed way to push a byte into memory. See also **MM, M, MM!,** and **M!,** for more elegant ways to manipulate memory.

EXAMPLE

1200 1300 20 MFILL
fills locations 1200-1300 with the value 20 hex.

COMMENTS

As with all memory writing commands, don't write into your stack area when loading into RAM.

MHIST no parameters PPA

Multiple-Pass **HIST**ogram invokes the optional Program Performance Analyzer (PPA), which allows you to display the execution time of your target program in each of up to 15 user-specified address ranges. See also **THIST** and **AHIST.**

USAGE

Allows you to examine the performance of your software. You can find out where your program is spending most of its time.

Press **F10** to exit from this menu-driven feature.

You must (only once) issue the command **SOFT** to enable this optional feature. **SOFT** performs a **SAVE-SYS**, and then causes an exit to DOS. The next time you call up the software, the PPA will be enabled.

MENU DRIVEN

You produce a histogram by first specifying the upper and lower limits of each address "bin" that you want displayed, then starting the gathering of data.

When you give the command **MHIST** you get the chart screen with the cursor positioned at the first bin. You can then start typing in the lower and upper limits of each bin. Use return, tab or an arrow key after you enter each number, to move to the next entry field.

Press function key 1 (**F1**) or **ALT-F1** to start displaying the histogram.

SAVE TO A FILE

You can save the setup of a histogram as a file with the **HSAVE** <file>. Issue this command after you exit from the histogram.

You load the histogram back in with **HLOAD** <file>. This command also invokes the histogram.

EXAMPLE

MHIST

This command is never used in combination with anything else.

MISC <byte> MISC
<byte> TO <byte> MISC
<byte> MASK <byte> MISC

Changes the analyzer trigger for the miscellaneous inputs.

THE MISCELLANEOUS INPUTS

The UniLab's 48-bit-wide trace buffer has room for 8 more bits than are used for data, address, and control lines. These eight input lines are available to you, for sensing anything on the target board that you want to know about, or that you want the UniLab to trigger on.

For example, you might hook them up to an output port, to trigger when a particular bit configuration gets asserted on that port.

The qualifier and filter specifications always ignore the **MISC** inputs.

USAGE

The simplest use sets up a trigger for a single value on miscellaneous inputs. The UniLab will search for the byte value, and trigger when it sees that hex number on the lines. See the first example below.

RANGES

TO lets you set up a trigger on any input between two byte values, inclusive. See the second example below.

NOT

NOT causes the UniLab to trigger when the value falls outside the specified range or value.

MASKING

You can use **k MASK l MISC** to examine any subset of the 8 miscellaneous lines. This is particularly handy when you only have one or two of the MISC inputs connected to your board. You don't care about the logic level of the other 6 lines, since they don't mean anything.

The high bits of k mark which bits will be examined, while the bit configuration of byte l indicates the values the lines must have for a trigger to occur.

(continued on next page)

(continued from previous page)

For example, **03 MASK FF MISC** selects only bits M0 and M1 for examination (with binary value 0000 0011). The UniLab would trigger when both these bits have a high value. The instruction **03 MASK 03 MISC** would have the same effect.

WITH TRACING

All trace filtering modes and qualifiers ignore the MISC inputs. Since they still effect triggering, this makes the MISC inputs particularly useful as trigger inputs for filtered traces.

EXAMPLES

NORMT 12 MISC S
after clearing all previous settings with NORMT, sets up a trigger for miscellaneous input 12, then uses S to start the analyzer.

12 TO 34 MISC
requires miscellaneous input value between 12 and 34 hex.

F0 MASK 00 MISC
sets a trigger based only on the four highest bits. The UniLab will look for a 0 on those lines.

23 MISC ALSO 45 MISC
sets a trigger on cycles where the misc input is either 23 or 45 hex.

COMMENTS

The MISC inputs can be connected to anything you like. They are often used to look at system input and output ports.

MISC' MISC' FILTER RARELY USED

Used only before **FILTER** to enable trace filtering on all inputs except the MISCellaneous wires(M0 to M7). **NORMx** turns this mode off.

WHY YOU DON'T NEED TO BOTHER

Because this is taken care of for you by **ONLY** and by x**AFTER**, so it is unlikely that you will need to use this command.

See also **CONTROL**, **HDAT**, and **NO**.

EXAMPLE

MISC' FILTER
enables filtering on all except M0-M7 inputs.

MLOADN <start> <end> <targ addr> MLOADN RARELY USED

Moves a block of memory from the memory of the host to the target memory.

USAGE

Allows you to assemble or load a program into host memory, and then move it to UniLab emulation ROM or target RAM.

Most people will prefer to assemble into a file, and then load from the file into UniLab emulation memory.

FREE MEMORY

The host memory area that is available generally starts right above C000. **PAD 100 + U.** displays the first free address. **SO U.** shows you the upper limit of the unused memory.

EXAMPLE

C000 C800 0 MLOADN
moves data at C000-C800 in the host computer to target locations 0-800.

COMMENTS

You must have emulation memory enabled to load the program into ROM (see **EMENABLE**).

MM <word> MM

Stores one 16-bit word in ROM or RAM and increments reference address.

USAGE

Used after an **ORG** statement (which sets up address), to patch program memory. Can only be used to change RAM after debug control has been established. See **RB**.

REMEMBERING CHANGES

LOG sends all memory access commands such as **MM** to the printer, saving a record of any changes you make.

EXAMPLES

3000 ORG 1210 MM
stores 1210 at 3000.

150 ORG 5000 MM 7001 MM
stores 5000 at location 150, 7001 at 152.

COMMENTS

Used for entering data tables, program patches, etc. See also **M**, **MM!**, and **M!**.

Will store to emulated memory if the address is enabled, otherwise will store to target RAM. See **NMI** and **RB**.

As with all memory writing commands, don't write into your stack area when loading into RAM.

If you have a disassembler the byte order is set correctly, otherwise you can set it with **HL** or **LH**.

MM! <word> <address> MM!

Stores a 16-bit word of data at the specified address.

USAGE

Used to patch program memory. Does not require a previous **ORG** command-- instead requires an address as the second parameter. See **MM.** Can also be used to change RAM, but only after debug control has been established. See **RB.**

REMEMBERING CHANGES

LOG sends all memory access commands, such as **MM!**, to the printer, saving a record of any changes you make.

EXAMPLES

1200 3000 MM!
stores a 1200 at 3000

5000 150 MM! 1000 152 MM!
stores 5000 at location 151, 1000 at 153.

COMMENTS

Used for entering small patches-- anything larger than one word can be done by one of the other memory patch commands with fewer keystrokes. See **MM** and **M.**

Will store to emulated memory if the address is enabled, otherwise will attempt to store to target RAM.

As with all memory writing commands, don't write into your stack area when loading into RAM.

If you have a disassembler the byte order is set correctly, otherwise you can set it with **HL** or **LH.**

MM? <address> MM?

Displays the 16-bit word that is stored at the specified address.

USAGE

To find out what is stored at a single memory location, either ROM or RAM. Use **M?** to look at bytes and **MDUMP** or **DM** for larger areas of memory.

EXAMPLE

1210 MM?
displays the word stored at 1210.

COMMENTS

If the address is **EMENABLE**d, then emulation memory will be displayed. Otherwise the UniLab will use DEBUG features to display target RAM contents. See **NMI** and **RB**.

If you have a disassembler, the byte order is set correctly, otherwise you can set it with HL or LH.

MMOVE ‹start addr› ‹end addr› ‹dest› MMOVE

Moves a block of memory from one area to another in the target memory space.

USAGE

Good way to make a little more room when you need to patch some extra code into a program.

You can also use it to relocate a relocatable code module.

SMART MOVER

Automatically chooses the order of moving, to prevent overwriting caused by moving from one area to an area that overlaps. Starts moving from either the beginning or the end of the area to be moved, as necessary. See the two examples below.

EXAMPLES

1000 2000 1005 MMOVE
moves the data in locations 1000-2000 up 5 places. Starts moving from the end.

200 300 125 MMOVE
moves the data in 200-300 down 75 spaces. Starts moving from the beginning.

COMMENTS

Make certain that the code you moved is relocatable. If it is not, you might have to patch some of the absolute address references. In general, exercise caution, and use **DM** on the moved memory, to see if the instructions still do what you want them to do.

As with all memory writing commands, don't write into your stack area when loading into RAM.

MODE no parameters F8

Gives you the mode panels, which allow you to change mode of display, disable and enable the DEBUG, etc.

USAGE

Press function key 8 (**F8**) once to get the first mode panel, which contains the analyzer mode switches. Press **F8** again to get the second panel that contains the trace display mode switches. The third panel contains the log mode switches and debug disable switches.

MOVING AROUND

To get from one panel to another, press F8 repeatedly, or use PgDn key. Use the END key to exit from mode setting.

Once you are in a pop-up panel, you can move around, selecting different features, with the up arrow and down arrow keys. The right arrow key toggles the feature on and off.

WHAT THEY ALL DO

See the Special Functions section of the manual for the complete story.

You can also check the listings in the glossary for each feature:

Panel One	**DASM**	**SYMB**	**RESET**		
Panel Two	**SHOWM**	**SHOWC**	**=MBASE**	**PAGINATE**	**HDG**
Panel Three	**LOG**	**TOFILE**	**PRINT**	**NMIVEC**	**RSP**

EXAMPLE

MODE

This command never used in combination with anything else.

MODIFY <addr> MODIFY

Dumps a screenful of memory, in a format similar to **MDUMP**, but also puts the cursor on the first location and lets you alter any location by overwriting the old value with a new one.

USAGE

The best way to display and alter memory. The interactive screen display shows you the value in each location and lets you alter any value.

You can alter any location by typing in a new hexadecimal value or by moving to the ASCII area and typing a character.

Press the **End** key to exit from **MODIFY.**

MOVING AROUND

The cursor keys move the cursor around on the screen. If you try to move up or down off the screen with an arrow key, one new line of memory will be bought onto the screen.

PgUp moves up one screenful, **PgDn** moves down one.

Use **Ctrl-Right Arrow** to move from the hexadecimal dump area to the ASCII. **Ctrl-Left Arrow** moves the cursor back.

EXAMPLE

20 MODIFY

Displays a screenful of memory, starting at address 20. The cursor keys will be reassigned as described above (and on the prompt line) until you press either **End** to save changes and exit, or **Esc** to exit without saving changes.

MS <count> MS RARELY USED

Pauses for count number of milliseconds.

USAGE

In test programs where you need a pause.

400 (hexadecimal) milliseconds is one second.

EXAMPLE

800 MS
pauses for 2 seconds (800 hex ms)

N no parameters

Resumes program, with a breakpoint set to the address after the next instruction.

USAGE

While stopped at a breakpoint, when you want to execute only the next instruction pointed to by the Program Counter. However, you will "fall through" loops and branches.

This "falling through" is often very useful. For example, if the PC is pointing at a subroutine call, **N** will show you the state of the processor when it returns from the call.

Use **NMI** to follow the execution of loops and branches.

FALL THROUGH LOOPS

When you single-step through a program, you will usually not want to bother going through loops the same number of times that the microprocessor does. This command allows you to go through a loop just once.

HOW IT WORKS

This command uses **RB** to set a breakpoint at the address just after the disassembled instruction that the PC points to. So the program runs until it reaches that address.

WATCH OUT

If the program never reaches the address of the breakpoint, then the program will run without stopping. For example, if the program contains an infinite loop, and you will not want to use **N** on the last command in the loop (the jump back up to the top). The program never reaches the code that follows that last jump.

COMMENTS

Available only after DEBUG control has been established.

NDATA <byte #1> <byte #2> . . . <byte #N> <N> NDATA

Sets N different bytes as trigger events for the analyzer.

USAGE

A quick way to set triggers on many different data codes that do not fall into ranges. Easier than using **ALSO** again and again, as in:
18 DATA ALSO 32 DATA ALSO 36 DATA ALSO 47 DATA.

RANGES OF DATA

If the data does fall into ranges, then you can use **TO** instead. For example, **12 TO 25 DATA** sets the analyzer looking for any data between twelve and 25, inclusive.

EXAMPLE

18 32 36 47 4 NDATA
Does the same thing as the ALSO example in the text above.

COMMENTS

Really the same as "ORing" together the terms with **ALSO.** Any number of terms can be listed, but be sure to get the count correct.

You can use **ALSO** in combination with this command to add a range of values.

NMI no parameters F4

Either establishes DEBUG control immediately or, if you already have DEBUG control, executes a single instruction.

USAGE

Now supported on all DEBUG software packages. Uses either the non-maskable interrupt pin (NMI) or the interrupt request pin (IRQ) of the target processor.

Allows you to establish debug control on a program that is currently running. See also **RB** and **RI**.

AUTO-BREAKPOINT

You can now read or write RAM and I/O without first gaining DEBUG control-- the UniLab will automatically do it for you. When you try to read RAM while a program is running, the UniLab will issue an NMI signal, gain DEBUG control, perform the requested function and then allow the program to resume.

SINGLE-STEPPING

After you have established DEBUG control, the **NMI** command allows you to execute one instruction, no matter what that instruction is. See also **N**.

NMI is especially useful for following jumps and branches-- instructions that **N** cannot follow.

DISABLE

If your target board makes use of the non-maskable interrupt (or IRQ) feature of your processor, or you wish to disable **NMI** for any other reason, use the Mode Panel (**F8**) or **NMIVEC'**.

Disabling the DEBUG features (Mode panel choice "SWI VECTOR" or command **RSP'**) also disables **NMI**.

COMMENTS

The hardware interrupt feature is also utilized by **SI**. Disabling **NMI** also disables that feature.

NMIVEC no parameters F8

Enables the Non-Maskable Interrupt vector installation.

USAGE

This command re-enables the UniLab's ability to perform **NMI**. You only want to disable this feature when you want more transparent operation and don't need to use all DEBUG features. See **RSP'** for complete transparency.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

NMIVEC' no parameters F8

Disables the Non-Maskable Interrupt vector installation.

USAGE

This command disables the UniLab's ability to perform **NMI**.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

The DEBUG features include the ability to send either a non-maskable interrupt (NMI) or an interrupt request (IRQ) to the microprocessor. Orion software packages use this feature to gain DEBUG control over your processor at any time.

WHEN YOU WILL WANT TO DISABLE NMI

When your system makes use of the NMI or IRQ, and you want to preserve that ability while testing the system.

COMMENTS

Either the panel toggle or **NMIVEC** re-enables the vector installation.

NO NO FILTER RARELY USED

Used before **FILTER** to disable the filter.

USAGE -- RARELY USED

You will probably never use this command. Used only when you want to turn the filter off while preserving the current trigger spec.

The filter mechanism of the UniLab gets turned on for you by the **xAFTER** macros. Those commands set the filter to **MISC' FILTER**, which allows you to set up a trigger spec based on all inputs except for the MISCellaneous wires.

See also **CONTROL**, **HDAT** and **MISC'**.

EXAMPLE

NO FILTER
turns off the filtering of bus cycles, but leaves the rest of the trigger spec untouched.

NORMB no parameters

Clears out (NORMalizes) all trigger descriptions and sets the trigger event near Bottom of trace buffer.

USAGE

To start a new trigger definition when you want to see the events that led up to the trigger.

Use **TSTAT** to look at how this command changes the **DCYCLES** setting.

When you want to start from scratch with a new trigger description, always begin with one of the variations of **NORM.** The three commands, **NORMB, NORMM,** and **NORMT,** vary only in where within the trace buffer they place the trigger event-- at the bottom, in the middle or at the top.

TO SEE WHAT HAPPENS NEXT

S+ restarts the target board with the same trigger specification, but with 170 (decimal) added to the delay cycle count, so that you can see what happened after the current trace window.

HOW THEY WORK

The commands clear out the truth tables the analyzer used to search for the trigger event, and set the number of delay cycles that the analyzer will wait between seeing the trigger and freezing the buffer. See **DCYCLES** for more information about delay cycles.

EXAMPLES

NORMB

Sets 4 delay cycles

NORMB NOT 0 TO 1000 ADR S

will show what happened before the address went outside of the 0-1000 range.

COMMENTS

NORMB should be used where you want to know what happened before the trigger.

NORMM no parameters

Clears out (NORMalizes) all trigger descriptions and sets the trigger event at Middle of trace buffer.

USAGE

To start a new trigger definition when you want to see the events that led up to the trigger, and also see what followed.

You will find it very useful when you want to see the complete context within which the trigger occurred.

Use **TSTAT** to look at how this command changes the **DCYCLES** setting.

See **NORMB** for more details.

EXAMPLE

NORMM
sets delay cycles to 85 (decimal).

NORMT no parameters

Clears out (NORMalizes) all trigger descriptions and sets the trigger event near Top of trace buffer.

USAGE

To start a new trigger definition when you want to see the events that followed the trigger.

Use **TSTAT** to look at how this command changes the **DCYCLES** setting.

See **NORMB** for more details.

EXAMPLE

NORMT
sets delay cycles to 165 (decimal).

NOT NOT <trigger description>

The trigger description gets interpreted as a description of when not to trigger.

USAGE

To tell the analyzer to trigger when some byte of the 48-channel input bus goes outside of a certain range or value. Most commonly used to trap bad data or a bad address.

EXAMPLES

NORMT NOT 00 TO 4FF ADR S
triggers if the address goes outside the 00 to 4FF range.

ONLY 127 ADR NOT 12 DATA S
shows only cycles where the data at 127 address is not 12.

NORMM NOT 12 DATA ALSO NOT 34 TO 56 DATA S
triggers when the data is not either 12 nor between 34 and 56.

COMMENTS

Sets a flag for the next trigger word (**ADR, CONT, DATA, HADR, HDATA, LADR,** and **MISC**).

Except when used with **ALSO**, the **NOT** command causes the truth table to be cleared to all 1's. Then 0's get written into the specified areas. This is the opposite of what happens without **NOT**.

With **ALSO**, the **NOT** command does not clear out the truth table first.

NOW? no parameters

Shows you what is happening on the target board right now.

USAGE

To see the code the microprocessor executes during the next 170 bus cycles.

EXAMPLES

NOW?

This command never used in combination with anything else.

COMMENTS

This command is a simple macro that turns off the RESET, so that it does not restart the target board, then sets its own trigger and captures a trace.

ONLY ONLY < trigger description >

Gives you a trace buffer filled only with cycles that match your description.

USAGE

Clears out the previous trigger spec and enables trace filtering. Only the bus cycles that contain the trigger cycle will be recorded.

Use this command when you want to see on the trace only the cycle described in the trigger specification. For example, only the read cycles, or only the command at address 0100.

ELIMINATE BORING LOOPS

This command is especially useful for filtering out status and timing loops that hog the trace space. See the second example below.

Notice that when filtering you have to use **AFTER** if you want to start the trace at some particular point in the program.

ONLY AND THE DISASSEMBLER

You will sometimes want to turn off the disassembler while using this feature. Disassembling partial instructions will give confusing results. Either the mode panel (**F8**) or **DASM'** turns off the disassembler.

EXAMPLES

ONLY READ
searches for and records only the read cycles.

ONLY NOT 120 TO 135 ADR AFTER 750 ADR S
produces a trace starting at address 750, excludes from the trace the routine at addresses 120 through 135.

ONLY 0100 ADR
records only the cycle executed at address 0100.

(continued on next page)

(continued from previous page)

COMMENTS

The analyzer will run until the trace buffer is full while keeping you informed of the number of spaces remaining. You can stop the analyzer at anytime by pressing a key. Then enter **TD** to see what you have captured in the trace buffer.

OPERATOR no parameters Macro Sys

Switches the UniLab software back to operator level.

USAGE

Your UniLab software was an operator system when you received it. You use this command after you have created a macro system with the command **MACRO** and now wish to return to the operator system.

The operator system you create with this command will not recognize any words you defined while in the macro system. See **MAKE-OPERATOR** to find out how to make an operator system that recognizes your macros.

The operator system gives you access only to the commands in the UniLab On-Line Glossary. The operator system has less power than the macro system, but contains enough power for all of your usual work with the UniLab.

FILE NAMES

If you request that the software switch back to operator system, your UniLab directory must contain a **.OPR** file which matches the name of your current **.EXE** file.

When you save a macro system you specify a new name that gets used for both the **.EXE** and the **.MCR** files. If that name was different from the original file name, you will now have to rename your original **.OPR** file before you can return to an operator system.

Use the DOS command **COPY** to make a copy of your **.OPR** file which has the same name as your current **.EXE** file.

Of course, if you had saved the macro system to a new name, then you could call the old standard operator software from DOS, rather than switching to a new standard operator system. See **MAKE-OPERATOR** to create a non-standard operator system.

EXAMPLE

OPERATOR
Converts software back to operator system.

ORG <address> ORG

Sets the origin (address at which you will start to poke new values into memory) for subsequent **M** and **MM** commands.

USAGE

To change the information stored in several sequential bytes of program or data memory.

You can alter emulation ROM at any time. If you keep **NMI** enabled, you can alter RAM at any time, since the Orion software will automatically gain DEBUG control, read the RAM and then resume execution of the target program.

If you disable **NMI** (see **NMIVEC'**), then you will need to run to a breakpoint before reading from RAM. See **RB**.

EXAMPLES

101 ORG 12 M 3410 MM
stores 12 to location 101 and 3410 to locations 102 & 103.

COMMENTS

Useful for entering program patches.

See also **M!** and **MM!**.

PAGE0 no parameters

Only for UniLabs with 128K of memory. Selects the bottom 64K page of emulation memory.

USAGE

Addresses that are four hex digits long (16 bit binary numbers) cover a 64K memory space, but your UniLab has 128K memory space. You must establish a context for the addresses to follow.

This command sets the offset to 0000, while **PAGE1** sets the offset to 10000. Thus, address 1300 after **PAGE0** refers to location 1300. Address 1300 after **PAGE1** means location 11300.

EXAMPLE

PAGE0
This command never used in combination with anything else.

PAGE1 no parameters

Only for UniLabs with 128K of memory. Selects the top 64K page of emulation memory.

USAGE

See **PAGE0** above.

Address 1300 after **PAGE1** means location 11300.

EXAMPLE

PAGE1
This command never used in combination with anything else.

PAGINATE no parameters F8

Enables pagination of trace display.

USAGE

The default condition. The trace stops after each screenful.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

You can turn this off with the pop-up panel, or with **PAGINATE'**.

COMMENTS

If you press any key while display is scrolling, trace display will stop.

PAGINATE' no parameters F8

Disables pagination of trace display.

USAGE

The trace display will scroll by continuously. Not very useful, unless you want to save an entire trace to a disk file. See **PAGINATE** above.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

PATCH no parameters

Redisplays the menu of processors supported by the current DDB software.

USAGE

Use when you make an error choosing your processor from the menu, or if you want to try a different configuration.

EXAMPLES

PATCH

This command never used in combination with anything else.

PCYCLES <count> PCYCLES

Sets the number of bus cycles that the analyzer waits between seeing the last qualifier and starting to search for the trigger event.

USAGE

The default is zero. Usually you will want the analyzer to start its search for the trigger event immediately after the qualifiers.

However, you will sometimes want the UniLab to wait some number of cycles after the qualifiers, before it looks for the trigger.

For example, you know that the program jumps to address 1000 from address 235. What you can't understand is why the code at address 1000 is being executed again, later on. So you do not want the UniLab to search for address 1000 until some time has passed since it saw address 235.

EXAMPLES

NORMB 1000 ADR 10 PCYCLES AFTER 235 ADR S
triggers if 1000 occurs 10 or more cycles after address 235.

COMMENTS

A pass cycle count can be used to hold off the search for a trigger, for whatever reason.

If there are several qualifiers the pass count starts after the complete sequential qualifier sequence has occurred.

PEVENTS <n> PEVENTS

Sets the number of times the UniLab will want to see the qualifying events before starting to search for the trigger event.

USAGE

The default value is one-- the UniLab will start to search for the trigger as soon as it has seen the qualifying event once.

You would use **PEVENTS** when you don't want to search for the trigger until the qualifiers have been seen a number of times. Useful for catching a trace after the nth iteration of a sequence.

This command is different from **PCYCLES**, which delays searching for the trigger an absolute number of bus cycles after the qualifiers have been seen.

EXAMPLES

NORMT 12 DATA 4 PEVENTS AFTER 30 DATA S
searches for 12 data anytime after 30 data has been seen four times

NORMT 100 PEVENTS AFTER 123 ADR S
triggers as soon as address 123 has occurred 100 times.

PINOUT no parameters

Displays pinout of target processor.

USAGE

A handy reference showing signal names and analyzer cable connections versus pin numbers.

EXAMPLE

PINOUT

This command never used in combination with anything else.

PRINT no parameters F8

Logs all screen output to the printer.

USAGE

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

PRINT' no parameters F8

Turns off logging all screen output to printer.

USAGE

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

PROMMSG no parameters

Use after a **STANDALONE** EPROM programming command, to display completion message.

USAGE

You use **STANDALONE** when you want to make use of your host computer while the UniLab is programming an EPROM. After the programming light goes out, you can use **PROMMSG** to check the outcome of the programming operation.

Q1 Q1 <trigger spec> RARELY USED

Selects the event description (trigger spec) that follows as qualifier one.
See **QUALIFIERS.**

USAGE

When you don't want to use **AFTER**, which you will find to be a more natural way to set qualifiers.

You will rarely use this, since **AFTER** automatically increments the context from **TRIG** to **Q1** to **Q2** to **Q3** each time it is used. You will find these words handy when you want to change your mind about one of the qualifying steps without entering the entire definition again.

EXAMPLES

Q1 15 LADR
Changes qualifier number one, so that the UniLab looks for 15 on the low byte of the address lines.

Q1 ALSO 28 LADR
Alters qualifier one, so that the UniLab accepts either 15 or 28 on the low byte of the address.

Q2 Q2 <trigger spec> RARELY USED

Selects the event description that follows as qualifier number two. See **Q1** for details.

Q3 Q3 <trigger spec> RARELY USED

Selects the event description that follows as qualifier number two. See **Q1** for details.

QUALIFIERS <1, 2, or 3> QUALIFIERS RARELY USED

Selects the number of qualifying events.

USAGE

Allows you to reduce the number of qualifying events. Usually you'll use **AFTER** to set qualifiers, and would use this command only to reduce the number of qualifiers if you change your mind.

When there are qualifiers, the UniLab searches for the qualifying events before it looks for the trigger.

You will probably prefer to use **AFTER**, rather than this command.

THE ORDER OF QUALIFIERS

If you have defined three qualifiers, the UniLab looks first for Q3, then for Q2 and lastly for Q1. It must see the qualifying events one immediately after the other. If it does not see one of them, it starts searching for Q3 again.

Of course, if there are only two qualifiers, then the UniLab looks for Q2 and Q1.

AFTER THE QUALIFIERS

Unless **PEVENTS** or **PCYCLES** has been set, the UniLab will immediately start searching for the trigger after it finds the last qualifier. OF course, the trigger event does not have to follow immediately after the last qualifier.

EXAMPLE

2 QUALIFIERS S

changes the number of qualifiers, so that the third one is ignored.

RB <address> RB

Resumes executing program, with a breakpoint set at indicated address. Must be used with **RESET** to establish debug control.

USAGE

The first breakpoint must be in emulated ROM, and come after the stack pointer has been initialized. If your program does not initialize the stack pointer, then you cannot set a breakpoint. However, setting up the stack pointer usually only takes three or four bytes.

You can also use **NMI** or **RI** and **SI** to establish debug control.

MISSED BREAKPOINTS

If the breakpoint is not reached, then the program will continue to run until you press any key. You must then use **RESET** <address> **RB** to gain debug control. You can only set a breakpoint on the address of the first byte of an instruction.

After a missed breakpoint the UniLab will try to achieve debug control by asserting NMI.

Make certain that the address you try to set a breakpoint on gets executed by the program-- set an analyzer trigger on the same address with **NORMT** <address> **AS.**

And make sure that your program does initialize the stack pointer to point at RAM. DEBUG uses the stack to save the state of your system.

EXAMPLES

RESET 123 RB
enables reset, and then restarts the target system with a breakpoint set at address 123

(continued on next page)

(continued from previous page)

1007 RB
without restarting the target system, run the program with a breakpoint set at address 1007.

COMMENTS

The second example above will work only if you have already established DEBUG control.

The first example will establish DEBUG control, as will an **NMI** command. **RESET** does not restart your target board-- it enables the "reset" flag, so that the **S** or **RB** which follows restarts the target.

READ no parameters

Narrows the trigger specification to read cycles only.

USAGE

Instructs the UniLab to trigger only on read cycles. Handy when you want to trigger on data memory values, not program memory opcodes. Or, when you want to trigger on reads rather than writes to some address range.

On some disassembler packages, **FETCH** instructs the UniLab to trigger only on fetches from program memory.

EXAMPLES

READ 13 DATA
sets up to trigger when microprocessor reads a 13.

NORMT READ 1000 TO 2000 ADR S
triggers when processor reads any data from address range 1000H to 2000H.

COMMENTS

A simple macro which specifies a range of CONT input values. This command, like **WRITE** and **FETCH**, gets defined for a particular processor by the optional disassembler.

RES <n> RES

Clears bit n of the stimulus generator output. The number, of course, must be between 0 and 7.

USAGE

Simulates a peripheral input going from voltage high to voltage low. The stimulus generator allows you to test how your system responds to digital signals on certain lines.

EXAMPLES

2 RES
resets output S2.

1 SET 1 RES
pulses output S1.

COMMENTS

Used to reset individual bits of the 8 stimulus outputs. See also **SET** and **STIMULUS.**

RES- RES- <memory command> SHIFT-F9

Pulls the RES- output of the UniLab low and holds it low until the analyzer is started. This is one way to prevent your target processor from having problems when you read or write emulation memory.

USAGE

You can use this command with some target systems to hold the target processor in a reset state, before you access emulation memory. Otherwise, while you access emulation memory the target microprocessor will see only FFs when it tries to fetch from emulation memory (that is, all the data lines high). Some processors will quietly vector to an error-handling address when this happens, but other processors might "go south," taking peripheral devices and battery backup RAM with them when they go.

RES- will cause your target processor to reset, regardless of whether you have reset enabled or disabled.

RES- will not work if your target system has a "one-shot" in the reset circuit.

ALTERNATE SOLUTION

There is a more general solution to the same problem: establish DEBUG control before you access emulation memory. That way, your processor will be held in the idle loop while you access emulation ROM.

EXAMPLES

RES- 10 DN
pulls the reset line low and then disassembles from memory, starting at address 10. Reset will stay low until the next time you start the analyzer.

RES- 500 ASM
pulls the reset line low, then invokes the assembler, starting at address 500.

RESET no parameters F8

Selects automatic reset mode, which resets the target system when you next start the analyzer.

USAGE

Along with **RESET'**, allows you to choose whether to restart the target board when you start searching for a trigger, or just watch a program already in operation.

To gain DEBUG control with **RB** you must enable reset. Always type **RESET** ‹address› **RB** to be sure.

Automatic reset gets turned on by **STARTUP**, and gets turned off by **NOW**, **ADR?**, **SAMP**, and **RB**. The status of reset is not affected by **NORMx**.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

EXAMPLE

RESET
selects auto-reset

RESET' no parameters F8

Turns off the automatic reset mode. See **RESET** above.

USAGE

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

RI RI ‹trigger spec› SI

Allows you to gain DEBUG control on any bus condition.

USAGE

To gain debug control on a data value, or a control column value, or a range of address values. Always used in combination with **SI**. **RI** marks the beginning of the trigger specification, **SI** marks the end.

When the bus state you specify occurs this sophisticated feature asserts an NMI signal. Generally, it takes one or two instructions cycles to gain debug control.

EXAMPLES

RI 450 TO 470 ADR SI
Will achieve DEBUG control after any address in the range 450 to 470 appears on the bus.

RI WRITE 34 DATA SI
Will achieve DEBUG control after the value 34 is written into RAM.

RMBP <break point #> RMBP

Resets (clears) one of the multiple breakpoints and displays new status of the multiple breakpoints.

USAGE

When you want to get rid of one of the breakpoints that you set with **SMBP.**

See also **CLRMBP,** which clears out all the multiple breakpoints.

EXAMPLE

3 RMBP
clears multiple breakpoint number 3.

COMMENTS

Multiple breakpoints are used with to break on more than one address. There are 8 multiple breakpoints available in addition to the standard (unnumbered) breakpoint set by **RB** or **GB.**

RSP no parameters F8

Re-enables DEBUG, after it has been disabled by **RSP'** or **EMCLR.**

USAGE

Only when you have turned off the DEBUG features, and now want to be able to use it again. Not the same as establishing debug control, which you do with **NMI** or **RB.** However, if you have disabled the DEBUG, then you cannot use either of those commands.

Normally you will use the **MODE** panel (function key 8) when you want to enable or disable this feature.

RSP' no parameters F8

Turns off DEBUG features.

USAGE

Enables complete transparency-- no emulation memory is affected by the UniLab operation.

DEBUG is disabled for you by **EMCLR.**

RESERVED AREA

Allows you to use for your program the areas that Orion otherwise reserves for DEBUG vectors and overlays. Press **CTRL-F3** to get a help screen that includes information telling you where the reserved bytes are for your processor.

MODE PANEL

You will not be able to use the DEBUG features until you turn them on again from the **MODE** panel, or with **RSP.**

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

RZ no parameters

Resume program from breakpoint, without any breakpoints set. Debug control will be lost.

USAGE

When you want to run the program starting from the current address.

A handy command for exiting from DEBUG control. However, a better command is **GW** which waits until you start the analyzer, so that you can start the program from the breakpoint with a trigger set.

EXAMPLE

RZ

Continues the program after a breakpoint.

COMMENTS

Don't try to specify a trigger event before **RZ**-- it will not work.

S no parameters

Starts the bus state analyzer. Resets the target system if automatic **RESET** is enabled.

USAGE

You do not need to start the analyzer on the same line as the command that sets up the trigger event specification, though that is the usual practice. **S** is a separate command that gets the analyzer going with whatever spec you created already in place.

You can use **TSTAT** to see what the trigger has been set up to (Trigger STATus).

EXAMPLES

S

Starts the analyzer, with whatever trigger was last defined.

NORMT RESET 123 ADR S

clears out the trigger spec, turns on auto-reset, and then sets it to address = 123 before starting the analyzer (and restarting the target board).

S+ no parameters

Identical to **S**, except that it increases the delay cycle count by A6 counts.

USAGE

Handiest when you find that your current trace just starts getting interesting at the end. **S+** by itself will trigger on the same event, but with a new trace window that starts 3 cycles before the end of the previous one.

You should use this when your trigger spec is an event that gets regularly repeated during the program, or with **RESET** enabled. All **S+** does is change the value of **DCYCLES** and then start the analyzer again.

So if your trigger spec only happens once in the program, and **RESET** is disabled, then the UniLab will be searching a program in progress for an event that has already occurred.

EXAMPLE

S+

restarts the analyzer with an increased delay setting.

SAMP no parameters

Samples the 48 input lines several times a second, and displays them until any key is pressed.

USAGE

A good way to get a vague idea of what is going on. It will be clear to you that the program has been stuck in an infinite loop, or that it has gone far astray. But you will not be able to tell much, as you only see one cycle out of every several thousand.

DISASSEMBLY

You will probably want to turn off the disassembler, with the **Mode Panel (F8)** or **DASM'**. When the disassembler is enabled the isolated cycles will probably be disassembled incorrectly.

EXAMPLE

SAMP

This command never used in combination with anything else.

COMMENTS

Useful when you are trying to connect analyzer inputs to something and you want to continuously monitor their state. Similar to **1 SR** but it runs faster. Gives more detail on program execution than **ADR?**. Don't forget to start from scratch on trigger specs after using **SAMP**, because it defines its own trigger.

It also turns off the **RESET**.

SAVE-SYS SAVE-SYS <file name>

Saves the entire UniLab system program in its present state as a named DOS file. Prompts you for file name if you do not include it on command line.

USAGE

To save a version of the system with new macros, or with default drives changed. Or, just to save the current emulator enable values, the current trace, and the trigger definition.

Warning-- does not save the symbol table. Do that with **SYMSAVE** command.

EXAMPLE

SAVE-SYS B:NEWUL
Saves the system to a new file on the B: drive.

COMMENTS

The target program, which is in the UniLab itself, is not saved by this command. Use **BINSAVE.**

This command automatically makes the "file extension" **.COM.**

Since the entire program image is saved including any unintentional damage to the program, always keep backup copies.

SC <count> SC <file name>

Starts the analyzer and waits the specified maximum number of milliseconds for trigger. When trigger occurs, the trace gets compared to a previously saved trace.

USAGE

Very useful when writing test programs that compare the trace to a known good trace that you have stored away. Save traces with the **TSAVE** command. If a trace does not match, the host computer beeps and displays both a section of the previous trace and the first bad step of the new trace.

HARDWARE CHECKOUT

Probably most useful for hardware checkout. To get a vague idea of the capabilities, save a trace right now **(TSAVE test)**. Then pull the RAM off your target board and execute the command below. Don't change your trigger spec between saving the good trace and getting the new one. See Appendix F for examples.

EXAMPLE

test 400 SC

Starts the analyzer board with a 400H ms trigger time limit (1 sec.) and compares the trace to the one saved in file "test."

COMMENTS

If the time limit passes with no trigger, the host displays a "NO TRIGGER" message and beeps.

SET <n> SET

Sets bit n of the stimulus generator output. The number, of course, must be between 0 and 7.

USAGE

Simulates a peripheral input going from voltage low to voltage high. The stimulus generator allows you to test how your system responds to digital signals on certain lines.

EXAMPLES

7 SET
sets stimulus output 7.

1 SET 1 RES
pulses output S1.

COMMENTS

Used to set individual bits of the 8 stimulus outputs. See also **RES** and **STIMULUS.**

SET-COLOR no parameters

Change the display colors for a color monitor.

USAGE

After you have issued the command **COLOR** to inform the UniLab software that you have a color monitor, you can change the display colors with this command.

You use the cursor keys to choose different colors, and see them displayed as you choose. Press the **END** key on the numeric key pad when you have completed your choices. You will need to save the system with **SAVE-SYS** if you want the colors to be permanent.

SET-GRAPH-COLOR no parameters PPA

Change the display colors of the graph generated by the optional Program Performance Analyzer option (**AHIST** and **THIST**). This is only appropriate for a color monitor.

USAGE

After you have issued the command **COLOR** to inform the UniLab software that you have a color monitor, you can change the display colors of the histogram portion of the **AHIST** and **THIST** display screens with this command.

You use the cursor keys to choose different colors, and see them displayed as you choose. Press the **END** key on the numeric key pad when you have completed your choices. You will need to save the system with **SAVE-SYS** if you want the colors to be permanent.

SHIFT-FKEY <# of key> SHIFT-FKEY <command>

Assigns a command to a function key pressed while the SHIFT key is held down.

USAGE

Reassign the function keys on PCs and PC look-alikes. Use **SHIFT-FKEY?** (or SHIFT-F1) to find the current assignments.

The function keys allow you to execute any command or string of commands with a single keystroke. The initial assignments represent our best guess at what you will need. But you might want to change them.

To make your reassignments permanent, use **SAVE-SYS.**

EXAMPLE

6 SHIFT-FKEY TSTAT
assigns TSTAT to SHIFT-F6 .

COMMENTS

To execute a string of commands, define a macro first (using :) and then assign the macro to the function key.

See also **FKEY, CTRL-FKEY,** and **ALT-FKEY.**

SHIFT-FKEY? no parameters SHIFT-F1

Displays the current assignments of the SHIFTed function keys.

USAGE

Whenever you want to be reminded what command will be executed when you press a function key while holding down the shift key.

See **SHIFT-FKEY** to reassign the keys.

SHOWC no parameters F8

Shows the control lines on the trace display (the default condition).

USAGE

Turn on display of the control lines, **C7** through **C4**, as well as the high four bits of the address bus, **A19** through **A16**.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

SHOWC' no parameters F8

Turns off display of the control lines on the trace display.

USAGE

Turn off display of the control lines, **C7** through **C4**, as well as the high four bits of the address bus, **A19** through **A16**.

Though the UniLab must always monitor these wires, and sometimes they give you vital information (such as that you have the wires hooked up wrong), usually you don't need to see them.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

SHOWM no parameters F8

Shows the miscellaneous lines and the HDATA lines on the trace display (the default condition).

USAGE

Turn back on display of the miscellaneous lines and the high data lines (on 8 bit processors).

You will want to see these lines when you have them hooked up to your board. Otherwise, you can ignore them.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

SHOWM' no parameters F8

Hides the miscellaneous lines and the HDATA lines on the trace display (the default condition).

USAGE

Turn off display of the miscellaneous lines and the high data lines (on 8-bit processors).

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

SI RI <trigger spec> SI

Allows you to gain DEBUG control on any bus condition.

USAGE

Always used in combination with **RI**. Please consult the reference section on **RI**.

SMBP <addr> <breakpoint #> SMBP

Sets one of the 8 multiple breakpoints at the given address.

USAGE

Allows setting of up to 8 breakpoints, in addition to the unnumbered breakpoint that is set by **RB** or **GB**. The status of all 8 breakpoints gets displayed each time you set or clear one.

You must already have debug control before you issue this command.

To use multiple breakpoints, set all but one of your breakpoints with this command, and then use **RB** or **GB** to get the target program going again.

EXAMPLES

123 4 SMBP
sets a breakpoint #4 at address 123.

250 RB
sets a breakpoint at 250 and starts the target program going again.

COMMENTS

See also **N, CLRMBP, RMBP.**

Before using multiple breakpoints, you should examine the possibility of using the more powerful capabilities of the analyzer to do the same thing.

SOFT SOFT <filename> PPA

Enables the optional Program Performance Analyzer for the new command file that it creates. Need by used only once. Prompts you for the filename if you do not include it on the command line.

USAGE

Reconfigures your software, so that you can use the Program Performance Analyzer commands **AHIST, MHIST, THIST, HSAVE, HLOAD,** and **SET-GRAPH-COLOR.**

Do not use **SOFT** until after you have copied the **HISTxxx.OVL** file into your ORION directory.

EXAMPLE

SOFT ppaZ80
Creates a new **.EXE** file, PPAZ80, which will recognize the Program Performance Analyzer commands.

SOURCE No parameters

Re-enables the display of source code interleaved with disassembly of machine code. SOURCE is automatically enabled when you load a .MAP file in with **MAPSYM**.

USAGE

It is necessary to use this command only after you have disabled the high-level support feature with **SOURCE'**.

ABOUT HIGH-LEVEL SUPPORT

Orion high level support shows you the line of your source code which generated your assembly code. To use this feature, you must load your .MAP file with **MAPSYM** and have the relevant source files in the current directory.

See **MAPSYM** for more information.

SOURCE' no parameters

Turns off the display of source code. See **SOURCE**.

SPLIT no parameters F2

Toggles split screen mode on and off.

USAGE

Gives you the ability to compare traces, or parts of the same trace. You can also compare a trace to the assembly code (**DN**), or to your source text file (**TEXTFILE**).

WHAT WINDOWS ARE FOR

The right quadrants are reserved for the output of **DN**, and for the pop-up panels (**MODE**). **TEXTFILE** only works in the top window. Help screens are always shown in the top window.

MOVING AROUND

The END key moves you from one window to the other.

HISTORY

The history mechanism, which saves a record of what has happened during your session with the UniLab, only records information off of the bottom screen.

EXAMPLES

SPLIT

This command never used in combination with anything else.

SR <n> SR

ReStarts the analyzer Repeatedly. Displays n lines each time trigger occurs.

USAGE

Very useful for logging things repeatedly. You should first set up the trigger and starting point of the display with **S** and **TN**.

STOPPING

You start the infinite loop by entering **SR**. You break out by pressing any key.

HARD COPY

Use the **Mode Panel (F8)** or **PRINT** to log your output to the printer. The **Mode panel** also contains a feature that allows you to log to a file. See **TOFILE**.

RESETTING OR INTERRUPTING THE TARGET

If you use **RESET**, then the target system will be reset each time the analyzer starts.

WHEN TO USE SOMETHING ELSE

If the events you want to see occur more often than once per second and you want to see them in sequence, you can use **XAFTER** along with **A9 SR** to log bursts of the events in filtered format.

EXAMPLES

20 SR

Repeatedly displays twenty lines of trace buffer, starting the analyzer again after each display.

SSAVE SSAVE <filename> ALT-F9

Saves the screen image as a DOS text file.

USAGE

Save the image of a graph generated by the Program Performance Analyzer option, or save any other screen image that you want.

EXAMPLE

SSAVE nice.scr
Saves the current screen as a file, nice.scr.

SST <trigger spec> SST

Starts the analyzer in the standalone mode.

USAGE

Set the analyzer looking for a bug that you think will take a while to find. After you issue this command, you can disconnect the UniLab from your host, or you can keep it plugged in but exit from the UniLab program **(BYE)**.

Either way, the LED on the UniLab goes out when it finds the trigger. You then plug in the UniLab again, call up the UniLab program, and enter **TS** to display the trace.

EXAMPLE

NORMB 1200 TO 1300 ADR WRITE 3F TO FF DATA SST
Searches for this trigger in standalone mode.

COMMENTS

Handy when you want to search for an obscure bug without tying up the host computer.

SSTEP no longer available

The functionality of this command is now assigned to the new "smart" **NMI**. **NMI** now achieves DEBUG control for you, or executes one instruction if you already have DEBUG control. See also **N**.

STANDALONE STANDALONE <prom programming command>

Selects the standalone mode for the EPROM programming command that follows.

USAGE

Allows you to use the host computer for something else while the UniLab programs an EPROM. Especially handy when programming large EPROMs.

You can type in **STANDALONE** and press return, then use the PROM programming menu to program the EPROM.

When the LED next to the PROM programming socket goes out, the command has been completed. You can then enter **PROMMSG** to get the completion status message. The UniLab must remain connected to the host computer, or you will not be able to get the message.

EXAMPLES

STANDALONE
use this command and then make use of the convenient PROM programming menu to burn an EPROM in standalone mode.

STANDALONE 0 TO 1FFF P2764
you can also use **STANDALONE** along with a PROM burning command, if you know the commands.

STARTUP no parameters F9

Restarts the target system and gives a trace of the first 170 cycles of target system operation.

USAGE

Very useful mode at the first stages of system checkout. Allows you to check out the first few instructions, make certain that they execute properly.

The **RES-** wire from the analyzer cable must be properly connected to the target system, or the UniLab will not be able to reset the target processor. See the **Installation** chapter of the User Manual.

The very first cycle (cycle 0) is particularly important because if correct data is not fetched (often due to the address not being properly **EMENABLE**d), then the program will immediately "blow up."

MULTIPLE RESET

Some systems with simple R-C reset circuits (no hysteresis) will appear to reset intermittently many times before they finally settle down to stable operation. This is a nuisance if you want to look at a trace early in the program, but you will be able to see the program when it does finally settle down.

If your system does this, you might want to consider putting a logic element-- such as two Schmitt triggers in a row (part number LS14)-- into your reset circuit. That way your system will always get a good strong reset signal.

EXAMPLES

STARTUP

This command never used in combination with anything else.

COMMENTS

This is a target specific macro that usually looks for the reset vector address on the bus. If that address does not show up, system will wait forever. Or if a HALT instruction is fetched, will give a "NO ANALYZER CLOCK" message. See **TroubleShooting** chapter.

STIMULUS <byte> STIMULUS

Changes the 8 stimulus outputs (S0-S7) to correspond to the specified byte. Also pulses the ST- output.

EXAMPLE

10 STIMULUS
makes all stimulus outputs zero, except S4

COMMENTS

Useful for changing all stimulus outputs at once. Use **SET** or **RES** to set and reset individual signals. The stimulus outputs originate in the PROM socket on the front of the UniLab and are normally connected by the stimulus cable provided with your system. The stimulus signals are usually used to provide test inputs for the target system.

SYMB no parameters F8

Enables the symbol translation feature.

USAGE

Turns symbol translation back on, after it has been disabled with **SYMB'**. Symbols make the trace more readable, by allowing you to replace data and addresses with symbolic names.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

Symbols are entered by using **IS** or **SYMFILE**, either of which will turn on symbol translation.

SYMB' no parameters F8

Disables the symbol translation feature.

USAGE

To turn symbol translation off without clearing out the symbol table. See **CLRSYM** if you want to clear out the table.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

SYMDEL <number> SYMDEL

Allows you to remove one symbol from the current symbol table.

USAGE

Use **SYMLIST** first to get a list of all the symbols in order of increasing value. This list of symbols is numbered. You delete a symbol by using its number, not its value.

EXAMPLE

5 SYMDEL

Deletes the fifth symbol in the list that **SYMLIST** shows you.

SYMFILE SYMFILE <file name>

Loads a symbol table file produced by a cross assembler . Prompts for a file name if you don't include it on command line.

USAGE

Capable of loading symbol tables in almost any format. The first time you use it, **SYMFILE** presents you with a menu of predefined formats. You can choose one of those, and then save the system with **SAVE-SYS** to make that the default format.

You can change the default format with **SYMTYPE.**

Formats not on the menu can be defined using **SYMFIX** for fixed length files. Variable length files only come in two formats: name and then value, or value and then name.

The AVOCET format on the menu is for symbol files that are name, then value.

The MANX format on the menu is for symbol files that are value, then name.

The MICROTEK format refers to "MicroTek/New Micro," **not** to "Microtec Research."

EXAMPLES

SYMFILE C:\ASM\OUT.SYM
Loads into the UniLab a symbol file created by an assembler.

SYMFILE+ SYMFILE+ <file name>

Appends the contents of a symbol file to the symbol table.

USAGE

Provides a way of adding to a symbol table that already exists. **SYMFILE**, on the other hand, automatically clears the existing symbol table.

SYMFILE+ allows you to combine several symbol tables.

See also **CLRSYM**.

EXAMPLES

SYMFILE+ A:EXTRA.SYM
Adds to the symbol table the symbols stored in a file on the A drive.

SYMFIX <a> <b> <c> <d> <e> <f> SYMFIX

Defines symbol file parameters for formats that use fixed length records.

USAGE

Use this word to define your own **SYMFILE** format for fixed length records, if none of the predefined formats available on the **SYMFILE** menu suit your purposes. There are only two types of variable length record formats (value then name or name then value) and both appear in the menu.

The definitions of the 6 parameters:

a = offset from start of record to start of name field.
b = 1 if address is 4 ASCII digits or 0 if 16-bit binary.
c = address field offset from start of record.
d = 1 if binary address has most significant byte first.
e = pad characters used to fill between symbols.
f = record length.

EXAMPLES

0 0 B 1 0 E SYMFIX
defines the format for 2500AD abbreviated symbol table files. These tables follow the format:
ten bytes for the symbol name,
two bytes for the symbol value,
two pad bytes.

SYMLIST no parameters

Shows you a numerically ordered list of all currently defined symbols.

USAGE

To verify that your symbol file has successfully loaded in, or to remind yourself which symbols you have defined with **IS**.

This command also gives you the information that you need to selectively delete symbols. See **SYMDEL**.

EXAMPLE

SYMLIST
Lists all the current symbols.

SYMLOAD SYMLOAD <file name>

Loads a UniLab format symbol table file from the disk. Prompts for you for file name if you don't include it on command line.

USAGE

Loads up a symbol table that was saved with **SYMSAVE.**

These files are variable length, allowing symbols up to 255 characters long.

Warning: not compatible with symbol tables saved with pre-version 3.0 **SYMSAVE.**

EXAMPLE

SYMLOAD B:oldsyms
Loads into the UniLab a symbol table file from the B drive.

SYMSAVE SYMSAVE <file name>

Saves the symbol table as a named DOS file. Prompts for file name.

USAGE

This command saves only the symbol table, which you will be able to load in later with **SYMLOAD.**

Use **SAVE-SYS** to save the entire system.

EXAMPLE

SYMSAVE july3.sym
Saves the current symbol table to a file called july3.sym.

SYMTYPE no parameters

Re-defines the file format assumed by the **SYMFILE** command.

USAGE

Presents you with the symbol table format menu. This allows you to choose a different format after you have chosen one with **SYMFILE.**

The first time you use the SYMFILE command you are presented with a menu of formats. Once you have chosen a format, **SYMFILE** executes immediately, using the selected format. **SYMTYPE** allows you to alter your choice of format.

See also **SYMFIX.**

The MICROTEK format in the menu refers to "MicroTek/New Micro," **not** to "Microtec Research."

EXAMPLE

SYMTYPE

This command never used in combination with anything else.

T no parameters

Displays the trace from its current starting point until any key is pressed.

EXAMPLE

T

displays the trace.

COMMENTS

The current starting point for the trace display is defined by the most recent TN command. (STARTUP usually sets it to -4.)

If the starting cycle # is not actually in the trace buffer, the trace is started 4 lines from the closest cycle number which is in the trace buffer.

TCOMP <n> TCOMP <file name>

Compares the present trace buffer to a previously stored trace in the named file. Compares the last <n> cycles. Aborts and indicates error if any bit fails to compare.

USAGE

Very useful for writing automatic system test programs. Use the value **AA** to compare the entire trace.

Use **TSAVE** to save the trace of a good system. You can then use that saved trace to test other systems.

If **TCOMP** finds a difference between the current trace and the one in the file, it will display 9 lines of the stored trace and the first bad line in the trace of the system under test.

You can use **TMASK** to tell **TCOMP** to ignore one or more of the columns in the trace display. See **TMASK** for details.

You can also use **SC** to compare traces.

EXAMPLE

AA TCOMP march.2
compares the entire trace to the one stored as file "march.2."

COMMENTS

If you want to compare only part of the trace, use a smaller number. **TCOMP** will then skip over the first part of the file. This is useful for skipping over the already known discrepancies between two traces.

If **TCOMP** behaves in a confusing manner, try using it with the disassembler disabled (**DASM'** or use the mode panel, **F8**).

TD no parameters

Stops the analyzer and displays the current contents of the trace buffer.

USAGE

To see what is going on, when trigger has not occurred, or when you are producing a filtered trace that you do not think will fill up the trace buffer. Normally the trace is automatically uploaded to the host when trigger occurs.

TD skips over the first cycle in the buffer, and any other empty space (all 1's) at the top of the buffer.

EXAMPLE

TD

This command never used in combination with anything else.

COMMENTS

Since the buffer is filled with 1's before the analyzer is started, a partially filled filtered trace buffer will have good data only near the end. TD automatically skips over the empty space.

TEXTFILE TEXTFILE <filename>

Allows you to look over a text file from within the UniLab program.

USAGE

TEXTFILE only works from the upper window. It will take a few seconds to analyze the file, and then will show you the first window full of text.

This feature is useful for looking at your source code while you debug it-- this could replace hard copy listings.

MOVING AROUND THE FILE

Use the PgDn key or the Down Arrow to see more of the text. The PgUp key scrolls the screen back, the Up Arrow moves you up one line. Use <line#> **TX** to move to a specific line number in the file.

The HOME key takes you back up to the top. The END key just toggles you to the lower window.

WATCH OUT

You can't alter the file in any way-- only look it over.

EXAMPLE

TEXTFILE \memo\project1
Opens the DOS file project1, in a directory called memo.

THIST no parameters PPA

Time **HIST**ogram invokes the optional Program Performance Analyzer (PPA) that allows you to display how often the elapsed time between two addresses falls into each of up to 15 user-specified time periods. See also **AHIST** and **MHIST.**

USAGE

Allows you to examine the performance of your software. You can find out how the elapsed time between any two addresses changes, as different conditional jumps or branches are taken.

To get interesting and useful results, you will probably want to measure the time between two addresses in your main loop.

Press **F10** to exit from this command.

You must (only once) issue the command **SOFT** to enable this optional feature. **SOFT** performs a **SAVE-SYS,** and then causes an exit to DOS. The next time you call up the software, the PPA will be enabled.

MENU DRIVEN

You produce a histogram by first specifying the upper and lower limits of each time "bin" that you want displayed **(F9),** then starting the display **(F1).**

When you give the command **THIST,** you get the histogram screen with the cursor positioned at the first bin. You can then start typing in the lower and upper limits of each bin. Use return, tab, or an arrow key after you enter each number, to move to the next entry field.

Press function key 1 **(F1)** to start displaying the histogram.

SAVE TO A FILE

You can save the setup of a histogram as a file with the **HSAVE** <file>. Issue this command after you exit from the histogram.

You load the histogram back in with **HLOAD** <file>. This command also invokes the histogram.

TMASK <byte value> TMASK

Set up a mask which tells **TCOMP** which columns to compare.

USAGE

The lower six bits of the byte value tell **TCOMP** which groupings of the trace display to use when comparing traces. The default is 3F (00111111 binary) which tells **TCOMP** to check all columns.

Used when comparing traces to filter out erroneous error messages-- due, for example, to different wiring of the **MISC** lines.

MASK VALUES

Each of the six bits corresponds to one of the groupings. If the bit is one, then **TCOMP** will include that grouping:

BINARY	GROUPING	HEXADECIMAL
0000 0001	LADR	1
0000 0010	HADR	2
0000 0100	CONT	4
0000 1000	DATA	8
0001 0000	HDATA	10
0010 0000	MISC	20

TN <n> TN

Displays the trace buffer, starting at cycle n. Sets the starting point for future trace displays.

USAGE

For random access to the trace buffer, when you also want to reset the starting point used by **T.** To access the buffer without changing the default value of the point where the display starts, use **TNT.**

EXAMPLE

12 TN

Displays the trace, starting 12 cycles after the trigger. The rest of the traces this session will also be initially displayed starting 12 cycles after the trigger.

COMMENTS

You will usually want to use **TNT.** Use **TN** when you think that you will want to display from the same point on future trace displays.

TNT <n> TNT

Displays the trace buffer, starting at cycle n.

USAGE

Allows you to immediately look at any point in the trace buffer. **TN** does the same thing, but also changes the default trace starting point used by **T.** The default trace starting point is set to -5, until you change it.

EXAMPLE

-7 TN

displays the trace starting 7 cycles before the trigger.

TO <number> TO <number> <command>

Sets a flag that indicates that a range of numbers is being entered.

USAGE

Used with all of the trigger event description commands to define a trigger on a range of numbers. See **ADR, CONT, DATA, HADR, HDATA, LADR,** and **MISC.**

EXAMPLE

12 TO 34 DATA

Tells the analyzer to look for any data on the range 12 to 34 on the data inputs.

COMMENTS

In the example above omitting the TO would result in a trigger spec that would accept only data = 34.

TOFILE TOFILE <filename> F8

Use to start sending screen output to a DOS textfile as well as to screen.

USAGE

Use for toggling on the logging of information to a file. You can include that command on the DOS command line as a "command tail." For example:

C> **ULZ80 TOFILE A:JUNE7.LOG**

The usual DOS rules for naming files apply.

You will be prompted for the file name if you do not include it.

Turn off logging to the file with **TOFILE'**.

You can use the **MODE** panel (function key 8) to toggle logging to a file on and off, but you have to use the command to open the file in the first place.

COMMENTS

Files produced in this way can then be edited with a word processor, or shown on the screen using the DOS command: **TYPE** file name.

TOFILE' no parameters F8

Use to stop sending screen output to DOS textfile as well as to screen.

USAGE

Use for toggling off the logging of information to a file.

Normally you will use the **MODE** panel (function key 8) when you want to change this feature.

TOP/BOT no parameters END key

Moves you from top window to bottom, or from bottom window to top.

USAGE

You will usually want to just use the END key, which is the number one key of the numeric key pad.

Only active when the screen has been split with **SPLIT** (function key 2). See that word for details about windows.

EXAMPLE

TOP/BOT

This command never used in combination with anything else.

TRAM no parameters

Turns off a flag, so that subsequent memory reference commands refer to RAM rather than ROM. Necessary only with processors that allow ROM and RAM to occupy the same address space, or that address more than one 64K segment of memory.

USAGE

Only needed when you want to refer to RAM that occupies the same 16 bit address space as ROM-- for example, with the 68000 microprocessor.

The flag stays reset until you use the command **TRAM'**.

EXAMPLE

TRAM 0 F MDUMP
Dumps the contents of RAM, from address 0 to F.

COMMENTS

This command can sometimes get you into trouble-- if you use **RB** after **TRAM'** you will be setting a breakpoint in RAM. Which is fine if you meant to set a breakpoint in RAM, but disastrous if you meant to set the breakpoint in ROM.

TRAM' no parameters

The default condition-- turns on a flag, so that subsequent memory reference commands refer to ROM rather than RAM. Necessary only after **TRAM**, or similar processor-specific commands.

USAGE

Only needed after you use **TRAM**.

EXAMPLE

TRAM' 30 3F MDUMP
Dumps the contents of ROM, from address 30 to 3F.

TRIG TRIG ‹event description›

The event description that follows will be a trigger event.

USAGE

As opposed to **Q1, Q2,** and **Q3,** which tell the analyzer that the following description is a qualifying event. Useful if you want to alter the trigger event without altering the qualifiers.

EXAMPLE

TRIG 123 ADR
searches for 123 on the address lines.

COMMENTS

Used to select the TRIG truth table context again after **AFTER, Q1, Q2,** or **Q3** has caused another truth table to be selected. Useful if you want to change your mind about the trigger step after you have just defined a qualifier. The 4 truth tables are Q3, Q2, Q1, and TRIG.

TS no parameters

Displays trace after standalone mode trigger.

USAGE

To retrieve the trace from the UniLab's trace buffer, after you start the analyzer in standalone mode with **SST.**

When you use **SST** to start the analyzer, you can disconnect your host computer from the UniLab and run other programs on the computer. When the analyzer sees the trigger, the light next to the analyzer goes out. You can retrieve the trace at anytime after that.

To retrieve the trace you must start up the UniLab program while the UniLab is disconnected from the host. Use **CONTROL - BREAK** to break out of the **Initializing UniLab...** message. Then reconnect the UniLab and issue the **TS** command.

EXAMPLE

TS

This command never used in combination with anything else.

COMMENTS

TS begins by sending a "wake-up" code to the UniLab. Since this does not fit into the normal UniLab communications protocol, don't enter **TS** unless you have previously entered **SST.** If you do, the system will hang.

TSAVE TSAVE <filename>

Saves the current trace buffer as a file.

USAGE

A good way to save information about a trace for later review with **TSHOW** or for automatic comparison to another trace with **TCOMP** or **SC.**

EXAMPLE

TSAVE good.trc
saves current trace as a file called good.trc.

TSHOW TSHOW <file name>

Displays a previously saved trace.

USAGE

A useful way to examine traces saved while in the field, or by an automatic testing program. **TSAVE** saves the trace in the first place.

TCOMP will compare the present trace to the numbered trace, and let you know if they differ. That will probably, most of the time, serve your purposes better than looking over a trace.

EXAMPLES

TSHOW good.trc
Displays the trace saved by TSAVE into a file called good.trc .

COMMENTS

If you are tracking down a problem you can save interesting traces as you go so that you can look at them again later or even print them out (by using control-P to turn on the printer).

After you use **TSHOW** the trace image in the host contains the recalled trace, so you can use **T**, **TN**, or **TNT** to view it from various points.

When you want to load the UniLab's trace buffer back into the host, enter **TD**. Since **TSHOW** changes the setting of **DCYCLES**, the cycle numbers will be incorrect unless the changed delay setting is the same as the previous one.

TSTAT no parameters F7

Displays the complete status of the current trigger specification including qualifiers, delay and pass counts filtering, and auto reset.

USAGE

A good way to determine what the current settings are. Also a good way to check on how the UniLab interprets your trigger specifications.

EXAMPLE

TSTAT

This command never used in combination with anything else.

TX ‹line #› TX

Use with an open textfile to move to the specified line number of the file.

USAGE

A good way to quickly move around a textfile. See **TEXTFILE.**

EXAMPLE

300 TX

Moves to line 300 of the current open textfile.

WORDS WORDS <command>

Displays an alphabetical listing of the UniLab's commands, starting with the command or characters you include on the command line.

USAGE

To remind you of the names of some UniLab commands. Press any key to stop the listing.

EXAMPLE

WORDS INIT
shows a list of commands, starting with INIT.

WSIZE no parameters SHIFT-F8

Allows you to redefine the size of the windows.

USAGE

Once you enter this command, only the cursor keys are active. Use the "END" key (numeric pad key 1) to exit.

Use whenever you want to set the window size to something other than the standard setup.

EXAMPLE

WSIZE
This command never used in combination with anything else.

Chapter Eight:
TroubleShooting

Contents

Overview

We designed this chapter to help you get the **LTARG** sample program running on your target board with the UniLab. The use of a standard program makes it much easier to pinpoint and solve any problem that you have.

A sample trace of the **LTARG** for your processor can be found in the Target Application Note for your Disassembler/DEBUG and, in many cases, on your distribution diskette as well.

Back to Basics

If you have a problem while running your own target program, we recommend that you first go back to the **LTARG** program. Check whether your target system and your UniLab system function properly while your hardware runs that simple program.

Most of the time you will be able to solve the problem while working with the **LTARG** program.

How to use this chapter

Symptoms

Find the symptom on the previous page that most nearly matches your problem. Then turn to the page that covers that symptom, to find the solution.

When the UniLab does not work properly with your target system, be certain to check whether the target board has a problem. See the first entry in **Solutions.**

Sub-systems of the UniLab

If the problem is with the UniLab, you can usually trace your problem to only one of the four functional subsystems of the UniLab. These four systems handle:

1) communications between UniLab and the host computer,
2) emulation of target ROM,
3) analysis of target board program, setting triggers and capturing traces,
4) DEBUG features, such as setting breakpoints in target board program, single stepping, using **MDUMP** on RAM, etc.

The above list forms a hierarchy of dependence. That is, the other three subsystems all depend on the communications subsystem-- if it does not work, then the other three will work

erratically, or not work at all.

And, further down the hierarchy, if your trace is not correct, then you will not be able to set a breakpoint.

First things first

Similarly, always give the greatest emphasis to the first thing that goes wrong. For example, if the trace shows that your target board starts to execute at the wrong address, then you should ignore the rest of the trace.

You've already found the vital information-- the first opcode address is wrong, which indicates a bad address line on the target board.

When to call

If you cannot get the **LTARG** program to run, or you cannot figure out where the problem lies, then the next step is Orion Technical Support. Call (415) 361-8883 for assistance.

Solutions in Depth

Addresses do not appear on bus in proper sequence, or occasionally are incorrect.

Quick Check:

-- UniLab emulator and analyzer cables properly connected to target? Double-check against the diagram in your Target Application Note.
-- Bus short or cross-connection on the target? See below.
-- UniLab input buffers working properly? See below.

WHY

If the trace buffer shows bad addresses or bad data, then the problem might be caused by
the target system bus, or
the cables that carry the bus signals to the UniLab, or
the input buffers in the UniLab that receive those signals.
It is easy to set up a simple test of all three of these possible causes. The first step is to find the problem bit. Then you can find out whether the target, the cable, or the UniLab is to blame.

WHAT TO DO

Find the problem: lower address bit(s). Fill emulation ROM with NOOP instructions, and first look at a **STARTUP** trace (you might also need to put an appropriate address in your processor's reset vector). If the problem is with the lower address bits, or with an upper address bit stuck high, it will be obvious in this trace.

For example, if address line A4 is shorted to ground, then you will get this sequence of addresses:

Binary	**Hexadecimal**	
00001110	0E	
00001111	0F	
00000000	00	(should be 10)
00000001	01	(should be 11)

Obviously, A4 is the problem bit, though there might be another one as well-- for example, A4 might be shorted to another bus signal rather than to ground. You can check for this possibility by seeing if A4 ever goes high. Set a trigger spec that looks only for a high signal on that trace:

NORMM 10 MASK 10 LADR S.

Find the problem: upper address bit(s). If the problem is with the upper address bits, you might have to go through one of two slightly different procedures.

One possibility is to get a separate trace of each upper address bit as it makes the transition from zero to one. For example, to see A12 go high, set a trigger on address 0FFF (**FFF AS**). The cycle after the trigger cycle should show address 1000.

A more elegant solution is to capture in one or two traces all the transitions of the upper address bits. You do this with a series of jump instructions (use the line-by-line assembler, **ASM**). Jump to just before the transition, and then right after the transition, jump to just before the next one. For example, jump to address FFF, and then from address 1001 jump to 1FFF.

Find the cause of the problem. Once you've found the problem bit, you should first check the target board for a short. Turn off power to the target, and remove the cables that connect it to the UniLab. Use an ohmmeter to check for continuity between the problem bit and the ground or the source voltage (or the other trace that it seems to be shorted to).

If you find no problem on the target system, connect the UniLab cables to the target, but not to the UniLab. Use the ohmmeter again to check for a short. Check the UniLab cables by themselves-- perhaps the problem signal is simply not getting through to the UniLab, or is shorted to an adjacent wire in the cable.

If you can find no evidence of a short, connect the target system to the UniLab, power up the target, and check the UniLab input buffer. First force the problem bit to a voltage opposite to the one it is "stuck" at (only do this after you are certain that the trace is <u>not</u> shorted to the ground or the source voltage). Then use **STARTUP** to capture a trace.

If the problem bit is still stuck at the wrong voltage, you have a problem with the UniLab input buffers. Turn to page on "Bad input buffers" in this chapter.

But if the bit shows up at the voltage you are forcing it to, then you probably have a short between the "problem" bit and some other address or data line trace on the target system. Examine the trace to see if some <u>other</u> bit is also showing up at the new signal level.

Incorrect data fetched from memory.

Quick Check:

-- Bus contention between emulation ROM and target system RAM or ROM chips? Check the memory map of your target system. If a target system chip and the emulation ROM try to put data on the bus at the same time, the trace buffer will see mangled data.
-- Are you actually fetching from emulation ROM? See the next page.
-- UniLab emulator and analyzer cables properly connected to target? Double-check against the diagram in your Target Application Note.
-- Bus short or cross-connection on the target? UniLab input buffers working properly? To find the problem data bit, you follow the procedure below. Then turn to the previous page to find the cause of the problem bit.

WHY

As explained in the two previous pages, if the trace buffer shows bad addresses or bad data, then the problem might be caused by

- the target system bus, or
- the cables that carry the bus signals to the UniLab, or
- the input buffers in the UniLab that receive those signals.

WHAT TO DO

Find the problem: data bits. Put 0000 and FFFF into the first two words fetched from emulation ROM. If any of the data lines are shorted to either ground or source voltage, this simple test will show the culprit.

If data lines are shorted to each other, or to some other signal on the bus, you might have to put a number of different values into emulation ROM and check the outcome. For example, check adjacent data lines by putting into emulation ROM one of the data words that have alternating bits:

Binary	**Hexadecimal**
01010101	55
10101010	AA

Emulation memory does not respond to fetches.

Quick Check:

-- Is the 16-bit address enabled? Use **ESTAT** to check the **EMENABLE** settings.
-- Do the upper four bits of the 20-bit address match the enabled segment? Use **ESTAT**, and compare the low four bits of the CONT column to the **=EMSEG** setting.
-- If the full 20-bit address matches, does the emulation ROM see an output enable signal? See below.

WHY

The active low OEE- (Output Enable for Emulator, pin 42 on the analyzer connector of the UniLab) signal is derived by the UniLab circuitry from signals on the target board. The signals used vary slightly from processor to processor. If the emulation ROM does not get an OEE- signal, then it will not respond to a fetch.

The OEE- signal is jumpered at the Analyzer cable connection to the UniLab. You should check this jumper if you are having problems.

Depending on the target hardware design, the signal the UniLab picks up at a ROM socket might not be the signal the UniLab expects.

WHAT TO DO

OEE- jumper? Check your Target Application Note for the diagram of the jumpering of your analyzer cable. First make a visual inspection of the jumper to OEE-. If it looks okay, use an ohmmeter to check the electrical connection.

Wrong signals at ROM socket? When you connect to the target system with a separate ROM plug (instead of an Emulation Module), you can sometimes run into problems with OEE-.

Some target systems have a CS- (chip select) signal going to the OE- (output enable) input of each ROM. If the UniLab emulation cable is plugged into one socket but also emulating the ROM in another socket, this can cause problems.

If this turns out to be the problem, you might have to route a system OE- signal (rather than a chip-specific one) to the ROM socket, or order a "direct emulation cable" from Orion to pick up that signal. In some cases you can ground the OE- input.

Program hangs up on "Initializing UniLab. . . " message

First, press **CTRL-BRK** to get out of the initialization routine.

Quick Check:

-- UniLab plugged in and turned on? Turn it on, enter **INIT**, and try again.
-- UniLab connected to COM1? See below.
-- Do you have two serial ports? If so,is the second one properly "jumpered" as COM2? See page 8-12.
-- Is your serial port set up to work with a printer? See below.

WHY

If the system freezes right after "Initializing UniLab..." is displayed, it means that the program is waiting to receive a character from the UniLab. You can unlock the program by pressing the CONTROL and BREAK keys at the same time.

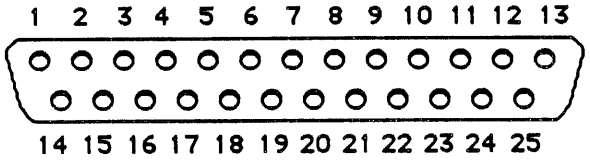
It might be that the UniLab simply has not been hooked up to the correct port.

Or, the UniLab might be on the correct port, but the computer is expecting to communicate over the wrong pin. The UniLab looks like Data Communications Equipment (DCE) to the host computer. That is, the UniLab expects to receive data on line 2 and send it on line 3 of the RS-232 connection.

WHAT TO DO

Check that the UniLab has been plugged into the correct DB-25 pin connector on your computer. If you have two 25 pin sockets on your computer, you should unlock the program with CONTROL-BREAK and move the UniLab cable to the second socket. Then use the **INIT** command to initialize the UniLab.

DB-25 Connector



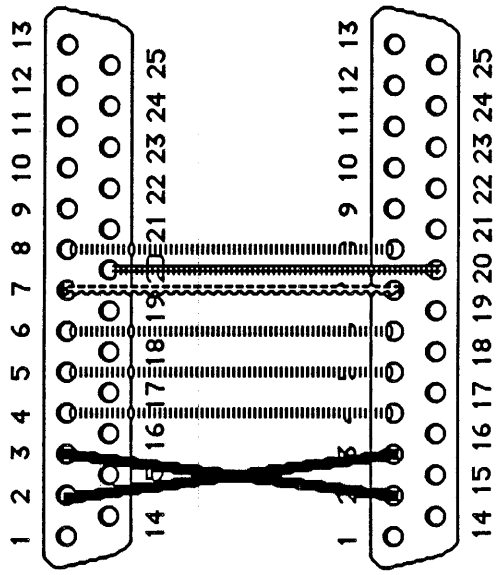
If that does not work, use **AUX2** to reconfigure your software, so that it expects the UniLab to be connected

to communications port 2. Again, try to initialize with the UniLab on each of the two serial ports.

If that doesn't work, check that the connector on the outside of your computer has actually been connected to the circuitry inside.

The jumpers on the serial board must be configured for operation with a modem or other DCE (Data Communications Equipment), rather than for operation with a printer or other DTE (Data Terminal Equipment). All serial port boards have jumpers that allow you to change the port to connect to a DTE or a DCE.

Only if you want to keep the port configured for a DTE, you should make or buy a standard "null modem" connector, or the non-standard null modem shown here.



Cable Configuration for Connection between UniLab and DTE serial port

(Or use standard null modem)

Program hangs on initialization some of the time, not all of the time

Quick Check:

-- Are your cables poorly connected? Check the UniLab-to-host RS-232 cable, and see page 8-8.
-- Are you running a background task, or a program that tries to write to the screen? See below.
-- Does your AUTOEXEC.BAT file set up a background task when you turn on the computer? See below.

WHY

Your real-time clock interrupt might cause RS-232 problems. The UniLab software can miss characters if too much time is going to process interrupts. One common cause of such problems is on-screen clock display utilities from programs such as Side-kick. If the clock interrupt routine does anything time-consuming, like writing to the screen, it can affect communications with the UniLab.

WHAT TO DO

Do not run desk accessories or background tasks such as a print spooler, on-screen clock display, alarm clocks, or multitasking, if you find that they affect the communications with the UniLab.

Take a look at your autoexec file, with the DOS command **TYPE AUTOEXEC.BAT,** to see if a background task has been set up to start automatically.

RS-232 error message: "RS-232 Error #XX"

Quick Check:

-- Poorly connected cable? Check the UniLab to host cable and see page 8-8.
-- Running a background task or desk accessory program? See previous page.
-- Two serial ports on your computer? Are they configured to two different addresses? See below.
-- Need to examine the RS-232 port of computer? See below.

WHY

The program running on your host sees incorrect data coming back when it tries to talk to the UniLab. The hex number following the # sign indicates what the error is.

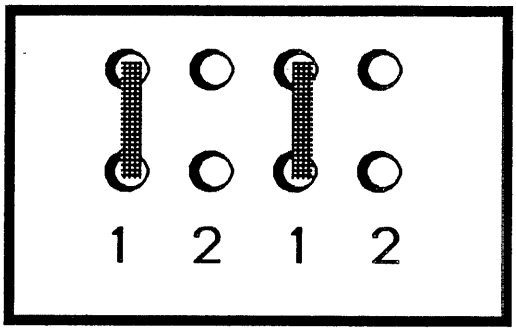
All RS-232 messages are checked with a 16-bit checksum and acknowledged with a single ACK (06) byte. Errors are signaled with other single-character responses as follows:

9B	Timeout error
A1	Overrun or framing error
69,70,75, 7C	Checksum error
2D	Length error
54	Load address error.

Other error codes indicate that the host is not receiving the UniLab transmissions correctly. For example, a 9600-baud host will usually get error CC or FC from a 19,200-baud UniLab, while a 19,200 baud host connected to a 9600-baud UniLab will usually get error 9E or FE.

WHAT TO DO

If you have two ports that might be "jumpered" to the same address, open up your computer and look at the boards that connect to the DB-25 connections on your computer. One or both boards should have a group of eight pins, as shown on the next page. This set of jumpers is different from the set that determines whether your port tries to talk to a DTE or a DCE, which are explained on page 8-9.



Typical pin arrangement on Serial port board

When the pins are jumpered as shown, then the port will be COM1. If the pins labeled 2 are joined, then the port will be COM2. If both ports are trying to be the same port, then you will have a bus contention problem.

To run a diagnostic test on your RS-232 port, first disconnect the UniLab from the host. Try sending single characters from within the UniLab program. You will have to jumper pins 2 and 3 on the DB-25 connector of your computer. Then type **INITRS232** to get the port ready. **30 SEND** will then send out a 30 on the serial port. Since pins 2 and 3 are jumpered together, the same port should receive a 30. Type **RCV .** to see what was received.

Try sending other numbers as well. If the port works fine, then you should suspect that pins two and three are reversed on that port. You can take care of this by putting in a null modem that switches pin 2 to pin 3 and pin 3 to pin 2. Pin 7 carries ground, and no other connections are necessary. See diagram below.

Another way to check the port. Put an oscilloscope on pin 2 of the DB-25 connector, and then follow the procedure above for sending characters. You will see on the oscilloscope whether or not signals are being sent on pin 2.

STARTUP does not work -- never get to see trace, or see trace filled with garbage

Quick check:

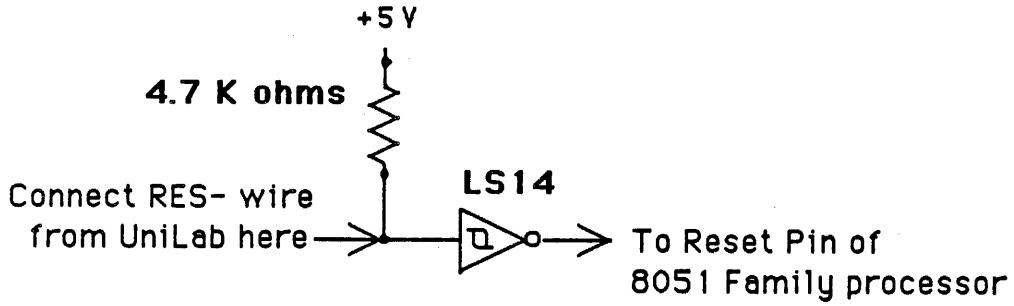
-- UniLab turned on and initialized? Turn it on now, then issue command **INIT.**
-- **LTARG** has been loaded? Issue the command **LTARG.**
-- **RES-** wire not connected properly? See below and explanation in section 3 of Chapter One.
-- Are address lines properly hooked up? See below.
-- Error message: "NO ANALYZER CLOCK" ? See page 8-15.
-- Do you have an **8051 family** processor? These require a positive going reset signal. See your Disassembler/DEBUG Target Application Note, or Section 3 of Chapter One.

WHY

STARTUP watches for the reset address on the bus, and then lets the trace buffer fill up before freezing the trace. The trace buffer will never be displayed on the screen if a proper reset never occurs, or if the lines the UniLab uses to sense the address have not been hooked up properly.

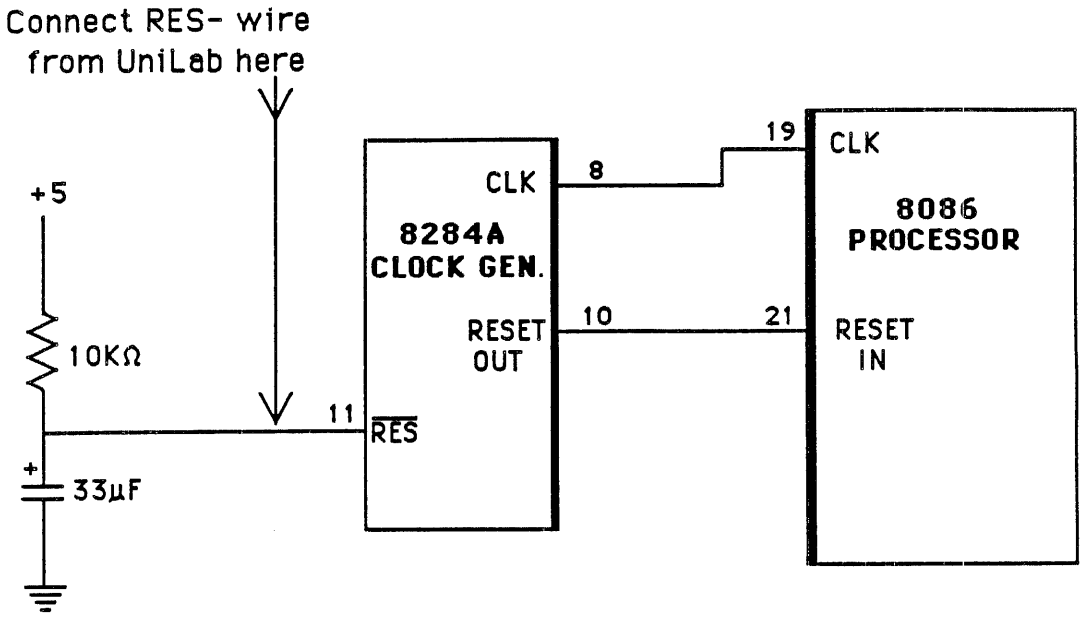
WHAT TO DO

Check the reset wire. First check it visually-- make certain that it has been connected to the Resistor-Capacitor circuit, that in turn drives the logic gate which drives the reset pin or your microprocessor. See below. Then check at the reset pin of the processor, with a logic probe or oscilloscope, to make certain the chip gets a good reset signal from **STARTUP.**

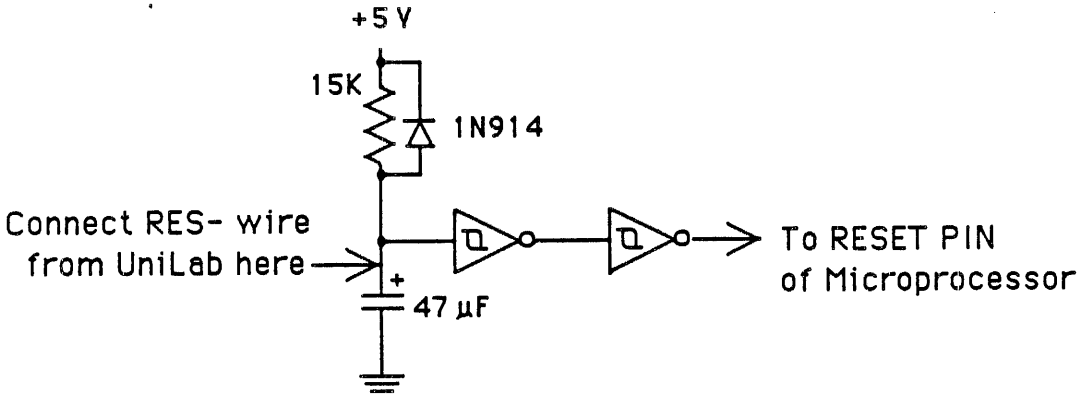


Typical Reset Circuit necessary for 8051 family processor

Check the address lines. Make certain that the ROM cable makes a good connection in the socket on your board, and that the extra address lines, if any, connect to the DIP clip at the proper pins.



Typical "power on reset circuit" for Intel microprocessor, showing connection of RES- line from Unilab



Typical "power on reset circuit" for Z80 microprocessor, showing connection of RES- line from Unilab

Error message: "NO ANALYZER CLOCK"

Quick check:

-- Power supply connected to target, and power on?
-- Do you have the correct analyzer cable? Use **PINOUT**, and compare the identifying letter (A, B, etc.) to the letter on your analyzer cable.
-- Bad timing wire connections? See below.
-- **RES-** wire not connected properly? See pages 8-13 and Section 3 of Chapter One.
-- Target system has gone to sleep? See below.

WHY

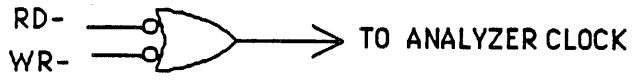
Usually the result of a bad target board or a bad connection-- the UniLab does not sense the control signals from the microprocessor, which tell it when the bus contains valid data and when it holds a valid address. If the power is not on, or the processor is doing nothing because it has not seen a reset, then there will be no clock.

The same symptom will result if the target system receives a command to wait or stop. This could happen if there is a bad data or address line, or if the target board does not restart because of an improperly connected reset signal.

WHAT TO DO

Bad timing wires. Double-check all the inputs to the UniLab that sense the timing signals. These are K1-, K2-, RD-, and WR-. Check these signals with an oscilloscope, to make certain that they are active.

You can also check the derived signal, TCY', at pin 35 on the analyzer connector of the UniLab.



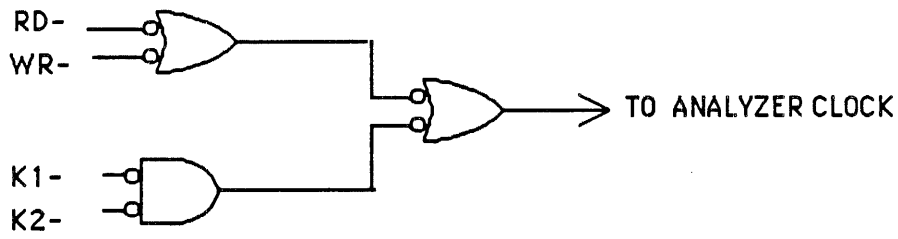
CLOCK LOGIC FOR MOTOROLA PROCESSORS

Target system asleep. Hit any key to stop the search for trigger, and then use the command **TD** to dump the trace. Look at the bottom of the trace, and see if the processor executed a halt command.

If it did, then check the address of the code that told the processor to stop. Is that an address that the processor should reach? If you use **DM** to disassemble from memory, is the opcode commanding a halt actually there?

If the program should not get to the code address where it reads the halt, check the **RES-** wire. See two pages back. You should also check the address lines.

If the address is correct but the data is wrong, check the data lines.



CLOCK LOGIC FOR INTEL PROCESSORS

Program runs, UniLab traces, but reads bad data from stack

Quick Check:

-- Stack pointer not pointing at correct location? See below.
-- RAM chip bad? See below.

WHY

The test boards that we use at Orion have RAM at some location in the memory map. The program loaded into emulation memory by the **LTARG** command was developed for our test boards.

But you might very well not have RAM at the address range where the Orion boards have RAM.

WHAT TO DO

To determine whether the stack pointer is okay, look at the first few lines of the trace, where the stack pointer gets initialized. Check the value of the stack against the addresses on your board that are occupied by RAM. Remember that the stack grows by decrementing the pointer.

If the stack pointer needs a different value, use the UniLab command ‹word› ‹addr› **MM!**. You use that command to change the program memory. It pokes a new 16 bit word into emulation ROM. Or use the line-by-line assembler, **ASM**, to change the instruction.

You will want to change the address field of the instruction that initializes the stack pointer. You should patch the program, so that the 16- bit address of the stack pointer points to RAM.

If the stack pointer points to RAM, but you still get bad values off the stack, you should suspect that you have a bad RAM chip. Try swapping in a new chip. If the bad values continue, then start checking your data and address line connections, both between the board and UniLab, and on the board.

Program runs and UniLab traces, but does not disassemble properly

Quick check:

-- First address of trace is not reset address? You might have an improperly connected reset wire. See page 8-13.
-- No disassembly at all? See below.
-- Clock speed of target board too high? See below.
-- **CONT** column correct? See below.
-- Correct disassembler for microprocessor? See below.
-- Correct cable for microprocessor? Double-check it.
-- Cable wired to DIP CLIP correctly? See below.

WHY

The UniLab listens to the bus, and interprets each 8-bit value it sees as opcode, data, or address, depending on the control signals it reads from the microprocessor. It then tries to disassemble the commands it sees, based on the disassembly table.

If you don't have all the wires connected properly, or you have the wrong disassembler, or you don't have the disassembler turned on, then you will not see what you expect.

WHAT TO DO

If the first address of the trace is wrong, then there is no point in looking at anything else. The only thing that matters is the first wrong step. Look at page 8-13.

If you think that you need to enable disassembler, just type in the command **DASM.**

If the target system clock is too fast for the UniLab, then some bus cycles will be skipped. Check for missing addresses in the fetch stream.

If the left digit of the CONT column is different from the sample trace, then the UniLab software does not know whether each bus cycle is a read, a write, or a fetch. That digit is generated by **K1-, K2-, RD-,** and **WR-.**

If you don't have the proper disassembler for your processor, then it's a surprise that you have gotten this far. Type the UniLab command **PINOUT** to find out what processor your software thinks it supports.

To check that you connected to the DIP clip correctly, look at the connection diagram which appears in each DDB Target Application Note.

Program runs, UniLab traces properly, but cannot set a breakpoint-- gives a "DEBUG Control not established" message

Quick Check:

-- Are you trying to set a breakpoint at address 0? This is not allowed-- try another address.
-- Does the analyzer work correctly? See below.
-- Does the emulator work correctly? See below.
-- Stack pointer points to RAM? See below and page 8-17.
-- Breakpoint set in code after stack pointer initialized? See below.

WHY

The UniLab sets breakpoints by preserving the byte at the address you specify, and inserting an absolute jump instruction, or a software interrupt. When the target system reaches that code, control gets passed to our idle register. The idle register asserts a jump instruction on the bus, which causes the processor to jump to the beginning of the jump instruction. The idle register thus holds the processor in an infinite loop.

Meanwhile, the UniLab software uses "an overlay area" in ROM, putting a routine there to save and read the state of the processor. (**HO** tells you where the overlay area is for your processor.) The idle register then changes, to assert a jump to the overlay area, which ends in a jump back to the idle register. All this swapping of control requires that the functions of the analyzer and emulator already work properly, and that the stack pointer points to functioning RAM.

WHAT TO DO

If either the analyzer or emulator does not work properly, then you must get those functioning before you try to set a breakpoint. Especially, read/write cycles must be properly identified-- which means that the disassembler must be working.

On processors with an external stack, if the stack pointer does not point to a good RAM chip, then you will not be able to set a breakpoint. Look at the trace that results from **STARTUP**, paying special attention to the POPs off the stack. Does the same data you push on the stack come back off it? If not, turn to page 8-17.

You must not set a breakpoint until after the stack pointer has been initialized. Try setting the breakpoint at the example address shown in the Target Application Note for your DDB software.

Program runs, UniLab traces properly, but cannot set a breakpoint-- hangs with red light next to Analyzer socket on until key pressed.

Quick Check:

-- Can you set a trigger on the breakpoint address? (That is, does program actually reach that address?) See below.

WHY

The UniLab sets a breakpoint at the address that you specify. It sets the break point by swapping in a special code at that address.

When the microprocessor hits that code, it behaves like a train sent onto a siding in the train yard. But if you don't specify the address of the first byte of an opcode, then the microprocessor gets derailed.

WHAT TO DO

Are you setting a breakpoint at the start of an opcode? Double-check this by setting a trigger on the address at which you are trying to set a breakpoint. The UniLab command is **NORMT** ‹address› **AS.** If you cannot set a trigger on the address, then that address is not the start of an opcode. Try a byte or two in either direction.

NMI does not work-- get "DEBUG control not established"

Quick Check:

-- Does **RESET** <addr> **RB** work? Check this before you try to use **NMI.** See previous two pages.
-- Intermittent problem, after working for a while. See below.
-- **RB** consistently works, but **NMI** doesn't. See below.

WHY

The **NMI** command sends a hardware interrupt signal to the target processor. For this feature to work, you need to have the NMI- wire from the UniLab properly connected to the target system. Some processors lack a non maskable interrupt and so the NMI- signal from the UniLab gets connected to an IRQ (interrupt request) pin.

WHAT TO DO

Intermittent problem. If the **NMI** stops working in the middle of a DEBUG session, it is possible that the UniLab has lost DEBUG control, but the host software doesn't realize it. Try using the command **IDLE'** and then **NMI.**

NMI doesn't work. First try it with the **LTARG** program. Some processors need to have some interrupt control registers set up for the **NMI.** **NMI** won't work unless you put the commands to set up those registers into your target software.

If **NMI** does not work with **LTARG,** then check the connection of the NMI- wire to the target. If that seems correct, use an oscilloscope to look at the NMI- signal. Does it go low when you issue the **NMI** command? Does the signal reach the processor pin?

You might also want to make certain that the processor responds to an NMI- signal properly. Use **Ctrl-F3** to get the address of the reserved ares, and set a trigger on it with **RESET NORMM** <addr> **ADR S.** Then use a wire to momentarily connect the NMI (or IRQ) pin of the processor to ground (or to the source voltage if active high NMI). This should cause the processor to go to the hardware interrupt vector and then from there to the reserved area.

Look at the trace to see what happened. If the UniLab does not trigger, look up the hardware interrupt vector address and set a trigger on that address.

Bad input buffers on the UniLab, as if an IC has been blown

Quick check:

-- Does the UniLab mysteriously show bad data coming in? Do some inputs always show up as high or as low, even if you apply a different voltage directly to the UniLab input? See below.

WHY

If you blow one of the input buffers-- by frying it with a high voltage, or through some other mishap-- then the IC will be damaged.

All ICs with external inputs are socketed, so they can easily be replaced.

WHAT TO DO

Make certain that you know what the problem is. Connect the suspicious input to ground, and then capture a trace. Connect the input to +5 voltage, and capture a trace. Inspect both traces to determine whether or not the input responds to the actual state of the circuit that it is meant to measure.

If you have a blown IC. Either send it to Orion for repair work, or replace the ICs yourself. All the Orion chips are standard pieces. All chips with external inputs are socketed for easy replacement.

Inputs:	Chip # on board:	Input Group:
A0-A7	U14	LADR
A8-A15	U15	HADR
D0-D7	U9	DATA
D8-D15	U8	HDATA
M0-M7	U7	MISC
C4-C7	U6	CONT

Screen flickers when you use PgUp key to look at line history

Quick check:

-- Issue the command **CLEAR.** Then use **SAVE-SYS** to save the altered system.

LIST OF APPENDICES

Appendix A:
UniLab Command and Feature List

This alphabetical list of the UniLab commands includes the page number of its entry in Chapter Seven.

Appendix B:
Sources of Cross Assemblers and C Compilers

The UniLab software is designed to work with any assembler or compiler. The only thing the UniLab needs is the object code in either binary format or INTEL hex format.

Even this hurdle can be overcome with one of the various conversion programs on the market. For example, Avocet has a product which converts Motorola S-records into binary format. See the Vendor listing for Avocet below.

As a service to our users we have compiled the following list of inexpensive cross assemblers and compilers. The two character appreviations indicate the sources listed on the following pages. We would appreciate any user feedback so that we can keep this list current.

PROCESSOR	ASSEMBLER SUPPLIERS	COMPILER SUPPLIERS
1802/5	25 AV MI WE AA EN RE SD UW	
6301	25 AV CY EN LO RE UW	IT AR
6305	AV MT RE	
6502	25 AV LO MI EN RE SD	MX
6800/2/8	25 AA AV MI DM EN LO QU RE SD UW	
6801/3	25 AA AV DM EN LO MI RE SD UW	AR IT
6805	25 AV DM EN LO MI RE SD UW	IT
6809	25 AV DM EN LO MI RE SD UW	IT
68000	25 AV EN LO QU RE SD UN UW	MX IT MT LA UW
68HC11	AV CY LO RE SD UW	AR IT
NSC800	25 EN RE	MT
8048-50/41	25 AA AV CY LO MI RE SD UN	
8051/31	25 AA AV CY LO MI RE SD UN US UW	AR MC
8080	EN LO UN	
8085	25 AA AV CY EN LO MI RE SD UW	MT
8086/8	25 AV CY EN SD SP SW UN UW	MX MS MT LA
8096	25 AA AV CY LO UN	
Z-8	25 AV AA CY EN LO RE SD UW	
Z-80 64180	25 AV AA EN LO MT RE SD US UW	MX KY LA
Z-8000	25 EN RE UN	

Vendor List begins on next page.

VENDORS

NOTE: All prices are approximate. Contact the vendor directly for latest information. This listing is a service to our customers, and does not constitute a recommendation.

25 2500AD Software Inc.
17200 East Ohio Dr.
Aurora, CO 80017, (303) 369-5001.
Eight-bit versions are $199.50, 16-bit are $299. They include recursive macros, nested conditional assembly, listing control, and a linker.

AA Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107, (818) 793-5748.
Resident editing capability, assemble to memory. $150. Macro/relocatable versions also available for $250.

AR Archimedes,(415) 771-3303. C cross compiler for 8051. $851.

AV Avocet Systems Inc.
120 Union Street
PO Box 490
Rockport, ME 04856 (800) 448-8500.
$200 for CP/M-80 or MS-DOS versions. $500 for the NEC 7500 and 68000.
HEXTRAN converts Motorola S-records to binary format. $250.

CY Cybernetic Micro Systems
P.O. Box 3000
San Gregorio, CA 94074, (415) 726-3000.
Conditionals, Macros. $295. Written in 8088 assembler.

DM Decision MicroSystems Co.
Box 120783
Nashville, TN 37212, (615) 320-7221. $210.

EN Enertec Inc.
19 Jenkins Ave.
Lansdale, PA 19446, (215) 362 0966. $250 and up.

FA Farbware
1329 Gregory
Wilmette, IL 60091.
Structured macro assembler $200.

IT Introl, (414) 276-2937.
C cross compiler for 6801, 6301, 6805, 6809, 68HC11, 68000, 68020. $1950.

KY KYSO, (503) 389-3452.
C cross compiler for Z80.

LA Lattice, (312) 858-7950.
C cross compiler for 68000, 8088, Z80. $500.

LO Logical Systems
6184 Teall Station
Syracuse, NY 13217 (315) 457-9416
Cross-assemblers for a variety of processors.

MC MicroComputer Control, (609) 466-1751.
C cross compiler for 8051. $1495.

MI Midwest Micro-DelTek, Inc.
5930 Brooklyn Blvd.
Brooklyn Center, MN 55429, (612) 560-6530.
Limited macros, cross reference, conditionals, 1K of object/minute. $300.

MS MicroSoft
10700 Northup Way
Bellevue, WA 98004. C compiler for 8086, 8088. $495.
As of release 4.0 of their software, MicroSoft does not make ROMable code directly. You can purchase utilities which are supposed to make the output of the MicroSoft compiler into ROMable code.

NOTE:
Microtec Research is **not** the same as "Microtek." The MicroTek symbol table format refered to in the SYMTYPE menu is compatible with the "MicroTek/New Micro" products.

MT Microtec Research
Box 60337
Santa Clara, CA 94088, (408) 733-2919.
C cross compiler for 68000, 68008, 68010, 68020. $1750.
C cross compiler for 8085, Z80, 64180, 8088, 8086, 80188. $1550.

NM MicroTek/New Micro
Supports a wide variety of processors. Call for latest product availability. (213) 538-5369.

MX Manx Software Systems
One Industrial Way
Eatontown, NJ 07724, (800) 221-0440.
C cross compiler for 8086, 68000, 8080, Z80, 6502.
$750.

PC Program Concepts Inc.
P.O. Box 8164
Charlottsville, VA 22901, (804) 978-1850. $595.

QU Quelo
843 NW 54 th St.
Seattle, WA 98107, (206) 784-8018.
Macros, conditionals, linker, cross ref, in C, $300.

RD RD Software
1290 Monument St.
Pacific Palisades, CA 90272, (213) 459-8119.
This is based on one that appeared in Dr. Dobb's Journal in June 1981 and April 1982. $200.

RE Relational Memory Systems
PO Box 6719
San Jose, CA 95150, (408) 265-5411.
Three different prices:
Macro assembler, non-relocatable. $139.
Relocatable code for 8-bit systems. $395.
Relocatable code for 16-bit systems. $495.

SD Software Development Systems
3110 Woodcreek Dr.
Downers Grove, IL 60515, (312) 971-8170. $295
Relocatable code, macros.

SE Seattle Computer Products, Inc.
1114 Industry Dr.
Seattle, WA 98188, (206) 575-1830. $95.

SW Speedware
9719 Lincoln Village Drive, Ste. 303
Sacramento, CA 95827, (916) 361-8664. $99.
With resident editor similar to Turbo-Pascal. Written in 8088 assembly language for speed.

UN Unidot Inc.
602 Park Point Dr. #225
Golden, CO 80401, (303) 526-9263.

US U.S. Software Corp.
5470 NW Innisbrook Pl.
Portland, OR 97229, (503) 645-5043.

UW UniWare
Software Development Systems
3110 Woodcreek Dr.
Downers Grove, IL 60515, (312) 971-8170.
8 and 16-bit cross-assemblers, $295.
C cross-compiler for 68000, $595.

WE Westico
25 Vanzant St.
Norwalk, CT 06885, (203) 853-6880. $225 Macro, $225 Linker.

WW Western Wares
Box C,
Norwood, CO 81423, (303) 327-4898. $395.

All of the Intel Series III MDS software can be run on the IBM PC with the UDI package from Real-Time Computer Science Corp., P.O. Box 3000-886, Camarillo, CA 93011, (805) 482-0333 ($500), or the ACCESS package from Genesis Microsystems, 196 Castro St., Mountain View, CA 94041, (415) 964-9001.

Appendix C:
Cabling Chart

PROCESSOR:	1802	16032	6301X0+	6303R	6303X	6502	6800
CABLE	g	c	b	b	b	b	b
Cable wires:							
A11		12	26	26	38	20	20
A12		11	25	25	37	22	22
A13		10	24	24	36	23	23
A14		9	23	23	35	24	24
A15		8	22	22	34	25	25
*RES-	3	34	6	6	6	40	40
*NMI-	36	45	4	4	8	6	6
GND	20	25	1	1	1	1	21
RD-	7	33	40	40	64	39	37
WR-	35						5
K1-							
K2-							
ALE	33	37					
C7	5	40	38	38	61	34	34
C6	6	41			60	7	
C5		42			63		
C4		43			62		
A19		4					
A18		5					
A17		6					
A16		7					
A0						2	

\+ at the end of the processor name indicates expanded mode or max mode

* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.
RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.
Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv.**

PROCESSOR:	68000	68008	6801+	6802	6805E2 6805E3	6809E
CABLE	p	p	b	b	b	b
Cable wires:						
A11	39	9	26	20	16	19
A12	40	10	25	22	15	20
A13	41	11	24	23	gnd	21
A14	42	12	23	24	gnd	22
A15	43	14	22	25	gnd	23
*RES-	18	37	6	40	1	37
*NMI-	23	42	4	6	2	2
GND	53	15	1	21	20	1
RD-	14	13	40	37	4	34
WR-				5		
K1-	10	31				
K2-	6	28				
ALE						
C7	9	30	38	34	5	5
C6	26	43		7	3	32
C5	27	44				38
C4	28	45				36
A19	47	19				
A18	46	18				
A17	45	17				
A16	44	16				
A0	7	46				

\+ at the end of the processor name indicates expanded mode or max mode

* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.
RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.
Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv.**

PROCESSOR:	68HC11	80186	80188	80286	8031+
CABLE	b	a	a	i	t
Cable wires:					
A11	13	10	10	20	24
A12	12	7	7	19	25
A13	11	5	5	18	26
A14	10	3	3	17	27
A15	9	1	1	16	28
*RES-	39	24	24	24	**inv**9
*NMI-	40	**inv**46	**inv**46	**inv**46	12
GND		26	26	9	20
RD-	27	62	62	(11)	29
WR-		63	63	(9)	16
K1-		40	40	(17)	17
K2-		39	39	**inv**(16)	31
ALE		61	61	(5)	30
C7	28	54	54	67	
C6	25	53	53	4	
C5		52	52	5	
C4				66	
A19		65	65	12	
A18		66	66	13	
A17		67	67	14	
A16		68	68	15	
A0		17		34	

\+ at the end of the processor name indicates expanded mode or max mode

( ) indicates a bus controller pin

inv connect to processor pin through an inverting circuit

* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.

RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.

Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv**.

PROCESSOR:	8048+	8080	8085	8086	8086+
CABLE	e	h	a	a	1
Cable wires:					
A11	24	40	24	5	5
A12	gnd	37	25	4	4
A13	gnd	38	26	3	3
A14	gnd	39	27	2	2
A15	gnd	36	28	39	39
*RES-	4	12	36	21	21
*NMI-	6		**inv**6	**inv**17	**inv**17
GND	20	2	20	20	20
RD-		18	32	32	(11)
WR-	10		31	29	(9)
K1-	8	{ 1}	11	27	(1)
K2-	gnd	{ 6}	3	24	(4)
ALE		17	30	25	(5)
C7		4	34	28	28
C6		3			27
C5		9	29		26
C4		10	33		
A19				35	35
A18				36	36
A17				37	37
A16				38	38
A0				16	16

\+ at the end of the processor name indicates expanded mode or max mode
( ) indicates a bus controller pin
{ } indicates a clock controller pin
inv connect to processor pin through an inverting circuit
* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.
RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.
Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv**.

PROCESSOR:	8088	8088+	8096	HD64180	NSC800
CABLE	a	1	r	e	q
Cable wires:					
A11	5	5		24	4
A12	4	4	**latch**30	25	5
A13	3	3	**latch**31	26	6
A14	2	2	**latch**32	27	7
A15	39	39	**latch**33	28	8
*RES-	21	21	62	7	33
*NMI-	**inv**17	**inv**17	**inv**7	8	21
GND	20	20	42	1	20
RD-	32	(11)	17	63	32
WR-	29	(9)	38	62	31
K1-	27	(16)	9	61	26
K2-	24	(4)		58	37
ALE	25	(5)	16		30
C7	28	28			
C6		27			
C5	34	26			29
C4			15		27
A19	35	35		59	34
A18	36	36		31	
A17	37	37		30	
A16	38	38		29	
A0	16		**latch**18		

\+ at the end of the processor name indicates expanded mode or max mode

( ) indicates a bus controller pin

inv connect to processor pin through an inverting circuit

latch attach the UniLab wires to the outputs of the latches, not directly to the processor pin.

* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.

RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.

Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv**.

PROCESSOR:	(8800) SUPER 8	(8681/82) Z-8+	(8400) Z-80	Z8001	Z8002
CABLE	d	d	e	c	c
Cable wires:					
A11	45	16	1	4	3
A12	44	17	2	5	4
A13	43	18	3	6	5
A14	42	19	4	10	9
A15	41	20	5	9	8
*RES-	30	6	26	16	14
*NMI-	23	25	17	15	13
GND	34	11	29	36	31
RD-	37	8	21	19	17
WR-			22		
K1-			27		
K2-			20		
ALE				34	29
C7	31	7		30	25
C6				20	18
C5				21	19
C4				23	20
A19			19		
A18					
A17					
A16					
A0				1	40

* RES- and NMI- are open collector outputs. Connect only to appropriate points. (NMI- needed only for certain DEBUG operations)

* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.
RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.
Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv.**

2. Pigyback Chips

PROCESSOR:	HD63P01	65/11EB	65F11Q	65/41EB	68P01
CABLE	n	k	k	k	n
Cable wires:					
A11	**rom**	**rom**	9	**rom**	**rom**
A12	join	join	8	join	join
A13	together	together	60	together	together
A14	A12-A15	A12-A15	61	A12-A15	A12-A15
A15			7		
*RES-	6	20	6	20	6
*NMI-	4	22	23	22	4
GND	40	21	44	40	40
RD-	1	3	45	3	1
WR-					
K1-					
K2-					
ALE					
C7			40		
C6					

* RES- and NMI- are open collector outputs. Connect only to appropriate points. (NMI- needed only for certain DEBUG operations)

inv connect to processor pin through an inverting circuit

rom A11 connects through the ROM plug

* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.

RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.

Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv**.

RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.

RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.

Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv**.

PROCESSOR:	68P05V07	80C51VS	87P50	(8613/03) Z8
CABLE	m	t	f	e
Cable wires:				
A11	**rom**	**rom**	**rom**	**rom**
A12	**gnd**	**gnd**	**gnd**	**gnd**
A13	**gnd**	**gnd**	**gnd**	**gnd**
A14	**gnd**	**gnd**	**gnd**	**gnd**
A15	**gnd**	**gnd**	**gnd**	**gnd**
*RES-	2	**inv**9	4	6
*NMI-	3	12	6	25
GND	1	20		11
RD-				9
WR-			10	
K1-		gnd	8	
K2-		30	gnd	
ALE		30		
C7				7
C6				

inv connect to processor pin through an inverting circuit
rom A11 connects through the ROM plug
gnd ground these address lines
* RES- and NMI- are open collector outputs. Often they cannot be connected directly to the processor.
RES- usually needs to be connected to the capacitor on the reset circuit that drives the processor pin.
Both NMI- and RES- sometimes connect to processor pin through an inverting circuit, as indicated by **inv**.

Appendix D:
Custom Cables

How Cables Work

The Sockets

The two 50-pin connectors on the front of the UniLab bring out extra signals so that operation of the instrument can be easily altered to meet the needs of different processors.

Since clocking logic requirements vary from one processor family to another, jumpers on the connector are used to make some interconnections.

Altering Standard Cables

Standard ribbon cables are provided that will work for most systems. In some cases, these cables must be reconfigured for proper operation with your system.

Since the connections are all made by the same insulation-displacement "U" contacts used in "Scotchflex" and "Quick-Connect" prototyping systems, they can easily be changed. A special wire-insertion tool is included with your UniLab for this purpose.

The Analyzer Cable

The analyzer is internally connected to all of the signals on the ROM cable. Any additional signals required for full monitoring of bus operations are picked up by connecting patch wires on the analyzer cable to your processor pins. This is usually done with a 40-pin Dip-Clip. The wires can also be plugged in to .025" wire wrap pins.

Your UniLab comes with an analyzer cable that is configured for the processor of your choice. You can alter your cable to support other processor families, or purchase additional cables.

Problems with Decoded OE- Signals

Most of the analyzer cable configurations (all except B, H, M, & N-- see the diagrams at the end of this appendix) assume that you have a memory enable signal connected to the OE- pin of the ROM socket into which the emulator cable is plugged. the OE- pin is pin 20 of the 24-pin ROM, pin 22 of the 28-pin ROM.

If you have address decoding in this signal, or if this pin is simply grounded, there may be problems.

The problem is that this signal is used as a low true master enable for the emulator's tri-state data bus outputs. This signal is necessary in systems that multiplex addresses over the data bus to prevent the UniLab from getting on the bus while addresses are being multiplexed.

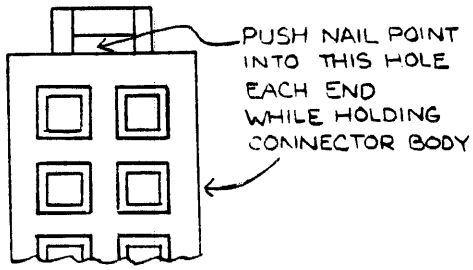
Most of the cable designs connect this input to the OE- signal on the ROM socket with a jumper between pins 42 and 39 on the analyzer connector. (The OE- signal is passed inside the UniLab from the ROM cable to pin 39 of the analyzer connector.)

If your target system has a ground or address decoding on the OE- pin of the ROM socket, you may have to separately connect pin 42 of the analyzer cable to an appropriate memory enable signal. On Intel bus controllers this signal is called MEMR-. Note that this signal also prevents the UniLab from getting on the bus during I/O cycles.

The problem with address decoding in the OE- signal is that the UniLab will then be unable to emulate memory for other ROM sockets.

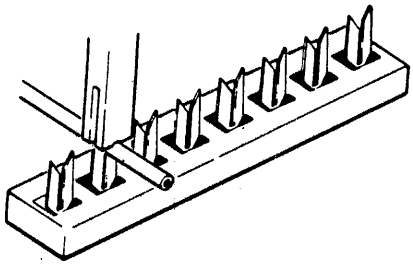
Customizing Cables

Two special tools are included with the UniLab so that you can easily modify the cables supplied. The first tool is actually a # 16 wire brad with a piece of shrink tubing on it. You can use it to open the connector by poking it into the space at the end of the connector as shown below:



Carefully release each end of the connector before you try to remove the cover. Don't try to remove the cover without the nail or you may break the plastic tabs!

Wires can be disconnected by simply pulling them out of the wedge in the connecting pin with small needle-nose pliers. A special tool is included with your UniLab for pushing wires into the connecting wedges (it looks like a tiny baseball bat). Note that these connections are identical to the ones used on "Quick Connect" , "Speedwire", and "Scotchflex" prototyping boards. The drawing below shows the proper way to use the tool.



You can use #26-30 solid or stranded wire for making connections. Wire-wrap wire works nicely. If you are jumpering a probe wire across to a second pin (as on pins 21-22 in fig e), hold the wire in place with your thumb while you use a small needle-nosed plier to put the necessary "jog" in the wire before you use the insertion tool.

If you are working with several different families of processors which require different jumper options, you should probably buy additional analyzer cables so you don't have to change jumpers whenever you change processor family.

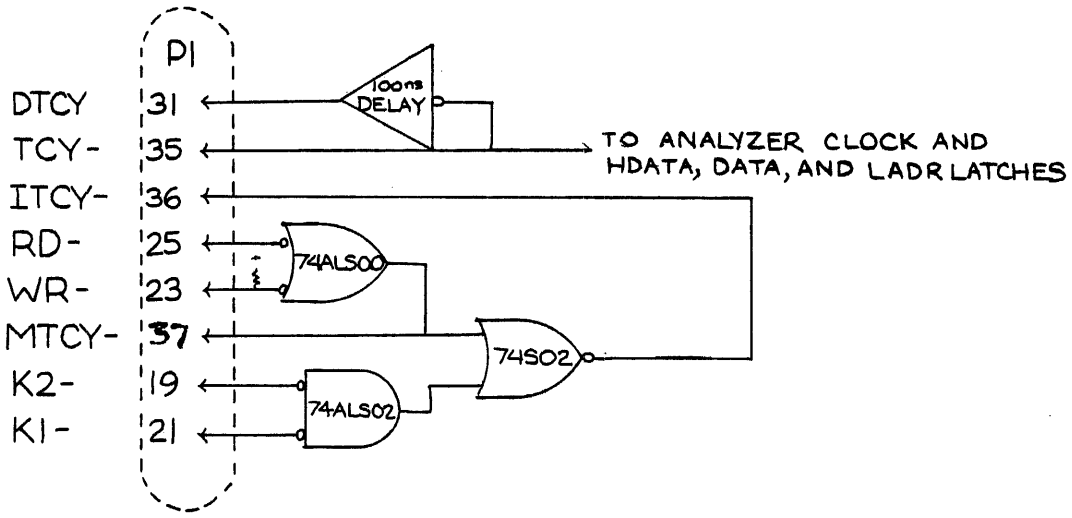
Analyzer Connector Signals

PIN#	SIGNAL	REMARKS
1	M7	MISC analyzer byte input.
2	M6	"
3	M5	"
4	M4	"
5	M3	"
6	M2	"
7	M1	"
8	M0	"
9	GND	Signal ground.
10	A19E	Msb emulator address inputs. Page select only
11	A18E	"
12	A17E	"
13	A16E	Msb emulator address input. To enable RAM.
14	+5V	Not used. Could power external circuits.
15	RDD	Emul enab. Can use to disable processor RD'.
16	RES-	Target reset. Open 7406 collector + 47 ohms.
17	NMI-	Interrupt. Open 7406 collector.
18	GND	Return for RES-.
19	K2-	Clock input. ITCY' = MTCY + (K1' and K2')
20	C7	Control input. Normally used for R/W,I/O,etc
21	K1-	Clock input. ITCY' = MTCY + (K1' and K2')
22	C6	Control input. Normally used for R/W,I/O,etc
23	WR-	Clock input. MTCY = RD' + WR'
24	C5	Control input. Normally used for R/W,I/O,etc
25	RD-	Clock input. MTCY = RD' + WR'
26	C4	Control input. Normally used for R/W,I/O,etc
27	A15E	Msb emulator address input.
28	ALE	Address Latch Enable for A0-A19.
29	INVI	Uncommitted inverter input.
30	INVO	Uncommitted inverter output.
31	DTCY	TCY clock delayed 100 ns.
32	CCK'	CONTROL byte input register clock (+ edge).
33	HACK'	HADRess byte input register clock.
34	MCK'	MISC byte input register clock.
35	TCY'	LADR,DATA,& HData input register clock.
36	ITCY'	Intel clock output. Jumper to TCY,CKs above.
37	MTCY'	Motorola clock output. Jumper to TCY,CKs.
38	ALE'	Inverter output from pin 28 input.
39	OE'	Output Enable' signal from ROM socket.
40	IDLE'	Not Used. Low when IDLE loop active. DMA?
41	CE'	Chip Enable' signal from ROM socket.
42	OEE'	Emulator Output Enable' (unlatched).
43	C3	CONTROL inputs. Normally A16-A19 from below.
44	C2	"
45	C1	"
46	C0	"
47	A19S	Latched emulator address signal (A19E).
48	A18S	"
49	A17S	"
50	A16S	"

Analyzer Cable Design

You can design your own cables for new processor types by copying and combining the techniques used in the cables shown in figure 7.1. The signal list for the analyzer connector immediately preceding this section should assist you further. While most of those signals are self explanatory, the analyzer input register clocking deserves some explanation. The analyzer logic and 3 of the 6 analyzer input bytes are clocked by the + edge of TCY' (pin 35). The other 3 input bytes (pins 32-34) are usually jumpered to this clock, but can be connected separately to clock their inputs at other parts of the clock cycle.

The diagram below shows the UniLab clocking logic:



The usual source of TCY' is ITCY' for Intel-type processors, or MTCY' for Motorola-types. MTCY' goes low whenever both RD- and WR- are high. By connecting these inputs to Motorola's E and VMA signals, the analyzer will be clocked on the falling edge of E if VMA is true. ITCY' goes low whenever RD- or WR- go low, or both K1- and K2- go low. Clocking in this case occurs at the end of the WR- or RD- pulse. DTCY is an inverted and 100 ns delayed version of TCY'. By using this signal to clock the CONTROL input register (CCK') the source of the clock (eg WR-) can be captured reliably as an analyzer input.

The ALE input (pin 28) controls transparent latches on the A0-A19 inputs to the emulator. The outputs of these latches are internally connected to the emulator and analyzer inputs, so clocking of the analyzer's address byte inputs can occur any time after they are stable. An inverted version of ALE is brought out on pin 38 so that inputs can be clocked by the end of ALE. An uncommitted inverter is also provided at pins 29-30 for processors, like the Z8000 & 16032, which use a low-true ALE signal.

signal.

The emulator output is enabled whenever the OE' signal on the ROM socket goes low and the 20 address bits satisfy the =EMSEG and EMENABLE statements which have been made. It is assumed that address decoding is done in the standard way using the CE' input to the ROM with OE' as a general memory enable. It is possible that some systems will make use of the OE' signal for address decoding. If this is the case, and you want to emulate several ROMs, you will have to change the jumpering to pin 42 to connect it to a more general enable' signal (such as MRDC' on Intel systems).

If there are I/O ports in the system which may have addresses which fall within the emulated address range, you must be sure that this enable excludes I/O operations. If no such signal exists, you can use an unused address input. For example, if the A19 input is connected to an IO/M' signal, and you specify 7 =EMSEG, the emulation memory will only be enabled when A19 goes low (memory cycles). Remember that =EMSEG acts as a modifier for all EMENABLE statements which follow, so if =EMSEG is changed the EMENABLEs must also be restated.

The descriptions which follow refer to the analyzer cables whose schematics appear at the end of this chapter.

(a.) Intel 8085, 8088-min mode, 80186, 80188, NSC-800: The CONTROL byte input register clock (pin 32) is isolated from the other input clocks and connected to ALE' (pin 38). This causes S0-2 to be clocked at the end of the ALE signal. All other input clocks (pins 33-36) are jumpered to ITCY' (pin 36), so that clocking will occur at the end of a low pulse on RD' or WR' or INTA'(K1-). Since K2- must be held low, it is connected to RES OUT on the 8085. C6 is internally connected to K1- to identify interrupt cycles.

To provide fewer cable variations, the 16-bit Intel processors all use the K1- & K2- inputs to derive a read and INTA clock by gating DT/R' and DEN' together. The write clock for the expanded mode can then come from either IOWC' or MWTC' at the bus controller. Note also that K1- is jumpered to C6 at the cable connector so that (DT/R') can be used both as a clock and an analyzer input without two separate connecting wires. A0 need be connected only if you have a 16-bit processor, as it is connected directly by the 8-bit ROM cable.

(b.) Motorola 6800, 6801/3, 6802/8, 6805E2, 6809: The UniLab address latch enable and CCK' pin are jumpered to the inverted DTCY signal so that control signals and addresses will be latched 100 ns after the rise of the E clock signal. This prevents trouble from the extremely short hold time of the address signals. Also note that the analyzer is clocked by the MTCY signal on the fall of E.

(c.) National 32016, Zilog Z8001, Z8002: The ALE signal is inverted, using the uncommitted inverter on pins 29 & 30. The control signals are latched at the end of the address strobe.

(d.) Motorola 68000, 68008, TI 9900, 99000, Z-8 romless, Intel 8085: A very simple configuration with all analyzer inputs clocked by ITCY.

(e.) Zilog Z-80, Z-8 piggyback, Intel 8051, 8031: WR', M1', and IORQ' are all used for clocking and captured by the analyzer to identify the cycle type. C5, C6, & C7 are therefore jumpered to WR-, K1-, & K2- so that a single probe can be used to make both connections. All analyzer inputs are clocked by DTCY so that the source of the clock will be captured. The address latches are enabled by DTCY' to prevent trouble from the short address hold time on Z-80A' B' & C' instruction fetches.

For the Z-80 only, A19 is connected to MREQ' so that OE' on the ROM socket needn't include an I/O term. Because of this you must use "7 =EMSEG" to get enable only when this signal is low.

(f.) Universal ROM-clocked: The OE' signal at the ROM is jumpered directly to the (RD-) clock input. C5 is jumpered to WR- and C6 is jumpered to K1- so that if a clock signal is connected to either of these leads, the signal will be captured by the analyzer without a separate connection. To reliably capture that input, the CONTROL byte input clock (CCK) is connected to DTCY. The address latch enable is connected to DTCY through the uncommitted inverter to prevent trouble from short address hold times. Since CE' at the ROM socket is connected to A16, you must use "E =EMSEG" to get enable when this signal goes low. You can make the UniLab ignore this signal by entering "F =EMSEG" then the EMENABLE statement, then "ALSO E =EMSEG" then repeat the EMENABLE statement.

(g.) RCA 1802: The TPB signal is used to clock the control inputs while the analyzer is clocked by MRD or MWR. Since the UniLab address latches cannot be separated, the MSB addresses must be connected to the target address latch outputs.

(h.) Intel 8080: The Ø2TTL clock signal is taken from pin 6 of the 8224 clock generator. The DBIN' signal is inverted and connected to K1-. The analyzer clock function is thus Ø2.DBIN + WR. The MEMWR- signal at the 8228 bus controller is used as an emulator enable. I/OW-, MEMR-, MEMW-, and INTA- are connected to C7 thru C4 so that the left digit of the analyzer control column will identify the cycle types as follows: F=I/OR, B=MEMR, D=MEMW, E=INTA, 7=I/OW.

(i.) Intel 80286: The ALE signal from the 82288 is jumpered to the CONTROL clock input so that S0 and S1 will be captured.

(k.) Rockwell 65/11 piggyback: Connects C7 to OE' which is R'/W, inverts CE and connects it to A15. A15 can be jumpered at the end of the cable to A12-14 for true address display.

(l.) Intel 8088/8086 max mode: Identical to cable A except that the uncommitted inverter is used to invert DEN signal. This inverter output is jumpered to the K1- input. Connect the DEN wire to pin 16 of the 8288 bus controller. A0 need be connected only if you have a 16-bit processor, as it is connected directly by the 8-bit ROM cable.

(m.) 6805 piggyback: Since no bus clock signal is provided by the processor, a circuit board is provided that derives clock from the signal at the crystal. This circuit includes a 74HCT74 CMOS divide-by-4 counter, which is reset whenever a transition occurs on the A0 signal. This reset ensures that the analyzer clock will be in sync with the internal processor clock.

(n.) 6801 piggyback, 6301: Identical to cable K except that the clock polarity is reversed.

(p.) 68000, 68008: Identical to cable D except that the OEE' input is grounded so that emulation will be enabled when either half of the data bus is read.

Note that some of these diagrams are untested and are provided only to help you get started. If you find any errors, please report them to us so that others can benefit from your discovery.

Also note that, since +5 volts is available at the connector, it is possible to make cables with logic gates on them if necessary. If you want to make a more conventional processor-specific emulator plug that plugs strictly into the processor socket in the target system, the RDD signal on pin 15 can be used with an OR gate to disable the RD- strobe at the processor when the emulation memory is active. This makes it unnecessary to unplug any ROMs that are being emulated, so all UniLab connections from both connectors could be made directly to a piggy-back processor with all signals except the RD' strobe directly connected. Of course this sacrifices universality and some transparency, but it might be more convenient in some situations.

The ROM Cable

There are 4 types of standard ROM cables:

C8-24. For 8-bit processors and 24-pin PROMs (2716,2732).

C8-28. For 8-bit processors and 28-pin PROMs (2764,128,256).

C16-24. For 16-bit processors and 24-pin PROMS.

C16-28. For 16-bit processors and 28-bit PROMS.

The C8-24 has an A11 pin, which can be left plugged into the A11 receptacle on the ROM cable if you are using 2732s or 32K ROMs. If you are using 16K ROMs, the receptacle must be plugged into the proper pin on the DIP clip to pick up the A11 signal at the processor. Other MSB address signals are likewise connected to the processor. Pin numbers for making these connections to the major processors are shown in the table in the previous section. Note that 24-pin cables will work fine in 28-pin ROM sockets if they are plugged in leaving the pin 1 & 2 end of the socket open. Extra address signals are simply picked up at the processor.

To minimize interconnections and signal loading, the analyzer data and address connections are taken from the ROM cable also. If your system has a unidirectional buffer between the ROM socket and the processor, these connections will not show data during write cycles. You can correct this condition by cutting the jumper ribbon cable on your ROM cable and installing a separate ribbon cable to the analyzer inputs on pins 35-42 (also 27-34 for 16-bit). You can order a cable that makes all connections at the processor by just ordering a C8-D or C16-D.

Note that all ROMs that are simulated must be removed from their socket to prevent bus contention. The ROM cable plugs into only one of the sockets-- except in the case of 16-bit systems, where there must be a second ROM plug in one of the most-significant-byte ROMs. In 8-bit systems the most-significant data bits are brought out in a separate cable, so they can be used as extra general-purpose analyzer inputs.

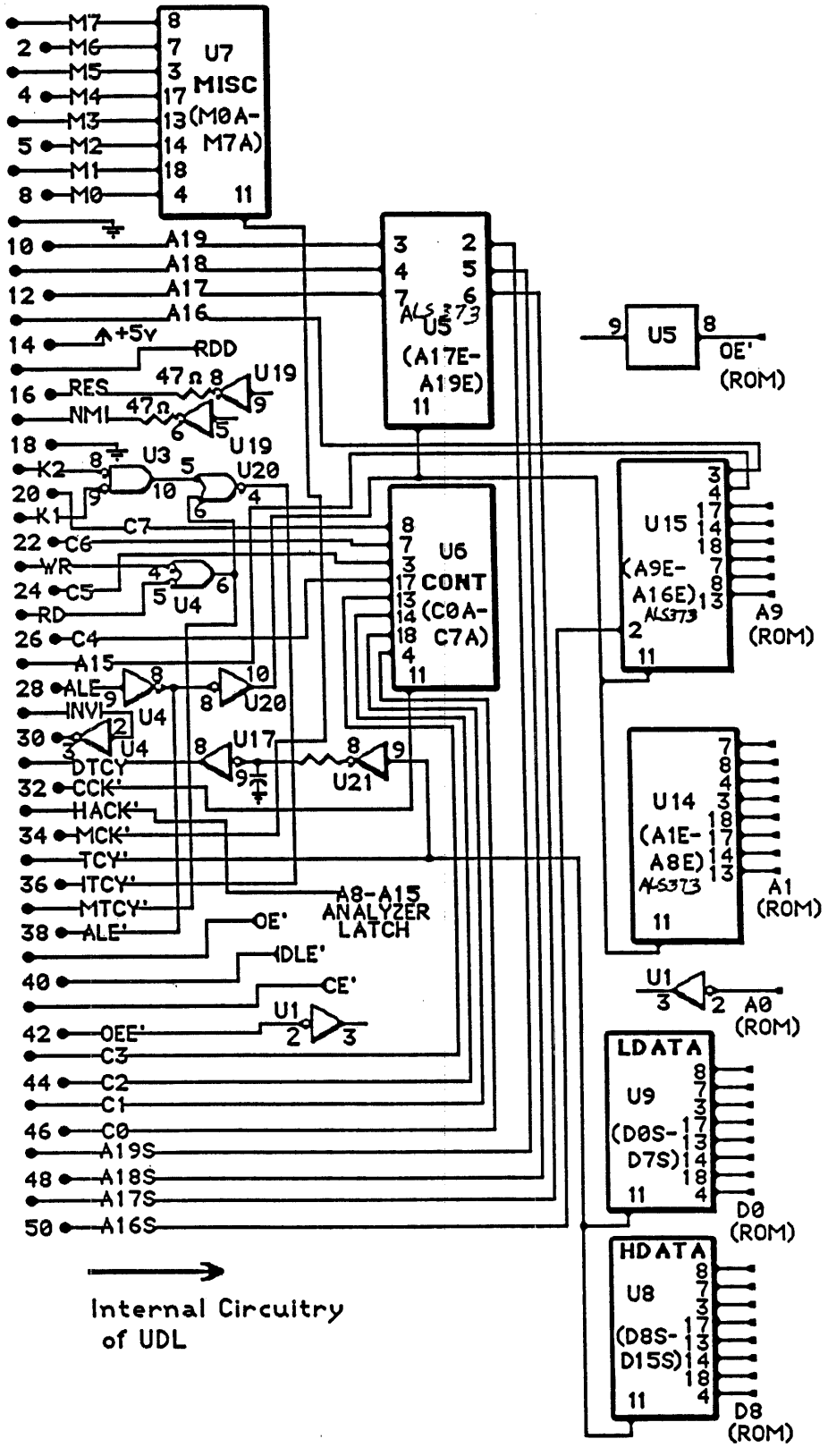
ROM Connector Signals

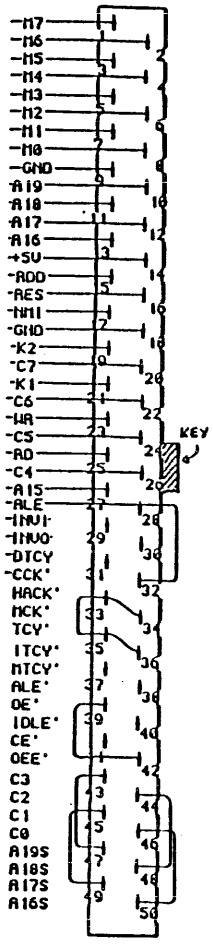
PIN#	SIGNAL	REMARKS
1	A14E	Direct connect on 256K ROMs. (lower left pin)
2	A12E	Direct on 64K or larger ROMs.
3	A13E	Direct on 128K or larger ROMs.
4	A7E	Emulator address inputs.
5	A8E	"
6	A6E	"
7	A9E	"
8	A5E	"
9	A11E	Direct on 32K or larger ROMs
10	A4E	Emulator address input.
11	OE'	To pin 39 on analyzer connector (for jumpering)
12	A3E	Emulator address input.
13	A10E	"
14	A2E	"
15	CE'	To pin 41 on analyzer connector (for jumpering)
16	A1E	Emulator address inputs.
17	A0E	"
18	GND	Signal ground. Shields adr inputs from data out.
19	D7E	Emulator data output. (odd addresses)
20	D6E	"
21	D0E	"
22	D5E	"
23	D1E	"
24	D4E	"
25	D2E	"
26	D3E	"
27	D11E	(even adr. LSB & MSB byte paralleled for 8-bit)
28	D10E	"
29	D12E	"
30	D9E	"
31	D13E	"
32	D8E	"
33	D14E	"
34	D15E	"
35	D7A	Analyzer data inputs. Usually jumpered to DnE.
36	D6A	"
37	D0A	"
38	D5A	"
39	D1A	"
40	D4A	"
41	D2A	"
42	D3A	"
43	D15A	(LSB & MSB byte paralleled for 8-bit)
44	D14A	"
45	D8A	"
46	D13A	"
47	D9A	"
48	D12A	"
49	D10A	"
50	D11A	"

Note: MSB and LSB are swapped for Intel convention. e.g. D8 above is really D0, etc.

UniLab Circuitry

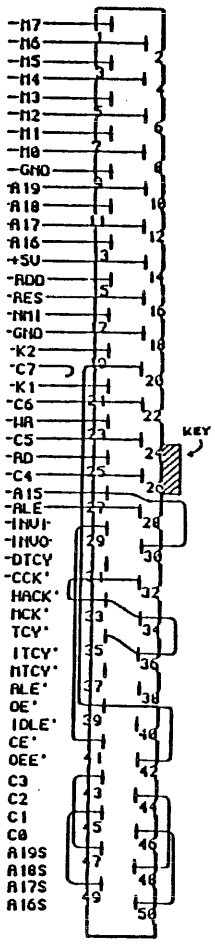
This page shows part of the internal schematics of the UniLab-- the input circuitry. The combination of this schematic and the diagrams that follow, showing the internal jumpers of all our standard cables, should give you enough information to customize a cable.





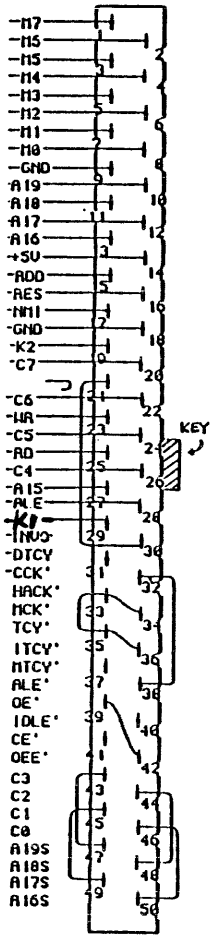
CABLE I

FOR 80286 (w/82288)



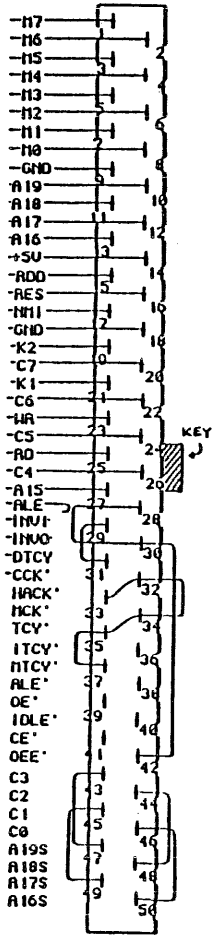
CABLE K

FOR 65/11 P-Back



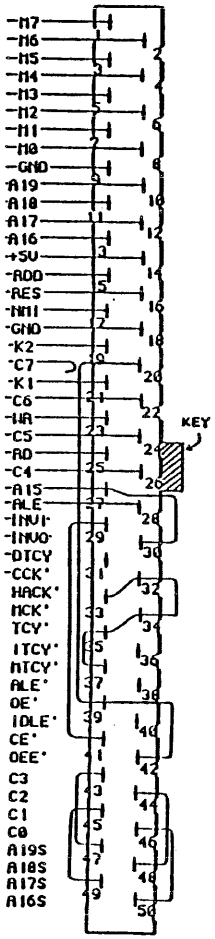
CABLE L

FOR 8086 max mode 8088 max mode



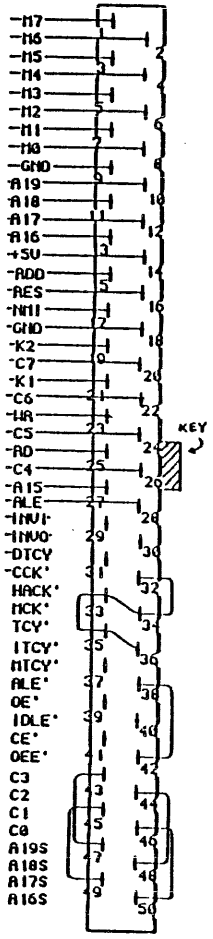
CABLE M

FOR 6805 P-Back (w/clk. ckt.)



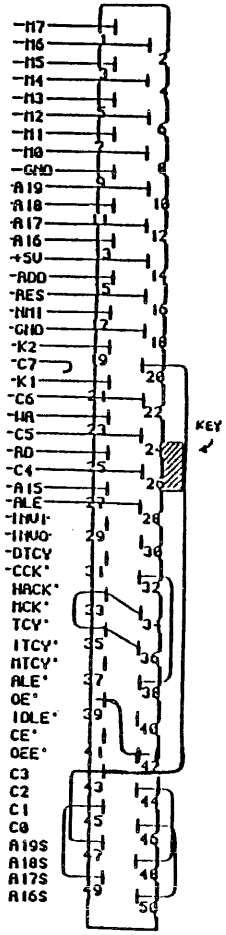
CABLE N

FOR 6801 P-Back 6301, 6805



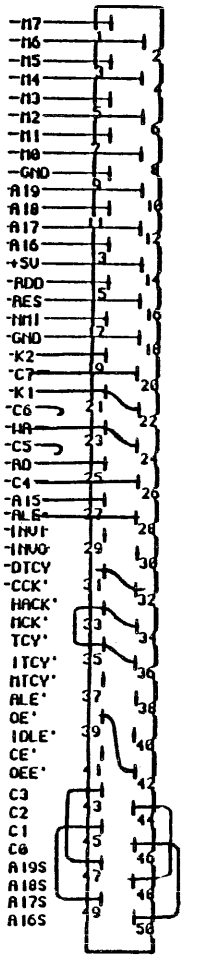
CABLE P

FOR 68000, 68009



CABLE Q

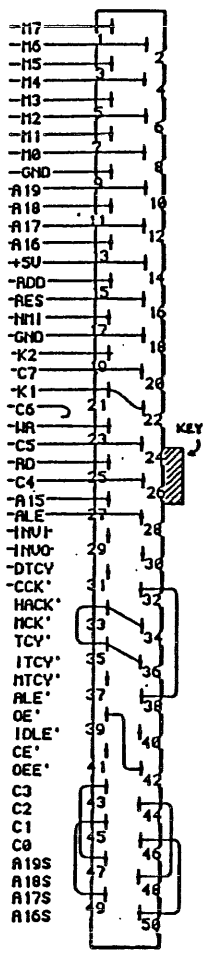
FOR NSC-800



CABLE R

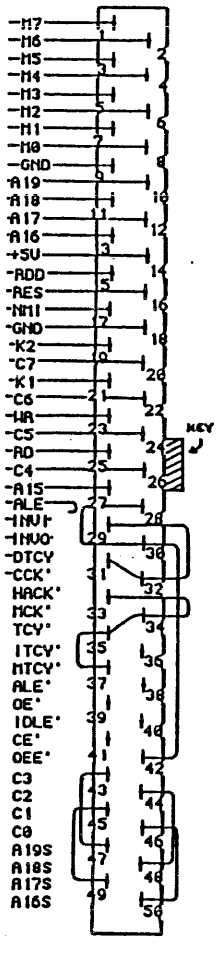
FOR 8096

(this page intentionally blank)



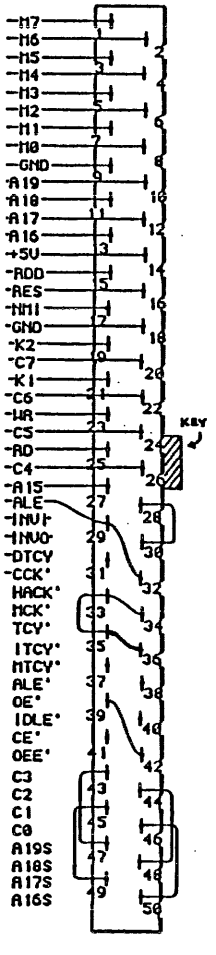
CABLE A

FOR 8085, 80186, 80188, 8086 min, 8088 min



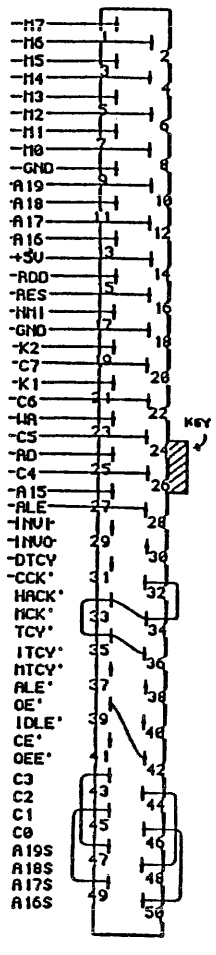
CABLE B

FOR R65/11Q, 6800, 6802, 6809, 6502



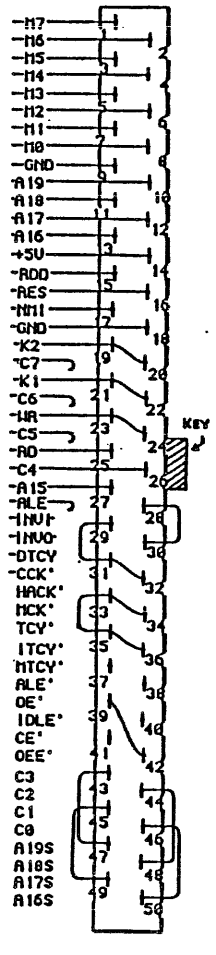
CABLE C

FOR Z-8001/2, 16032



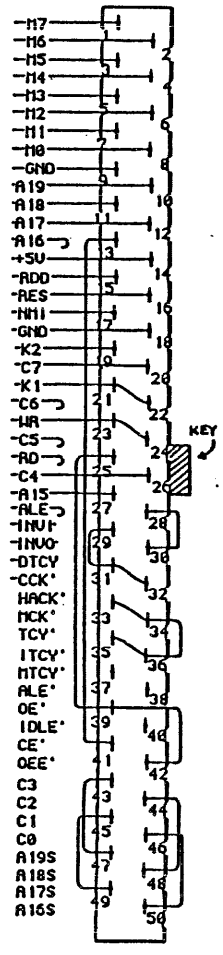
CABLE D

FOR 9900, 99000, Z-8+



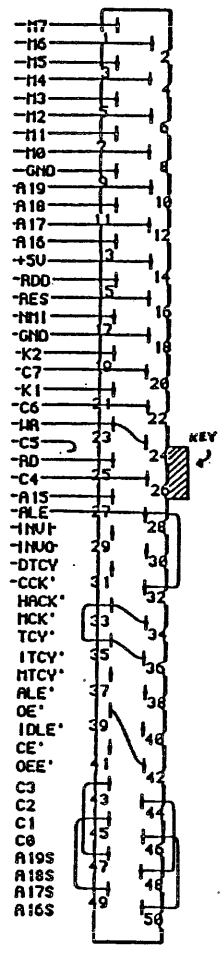
CABLE E

FOR Z80, 8051, 8031, 8039, Z-8 P.Back



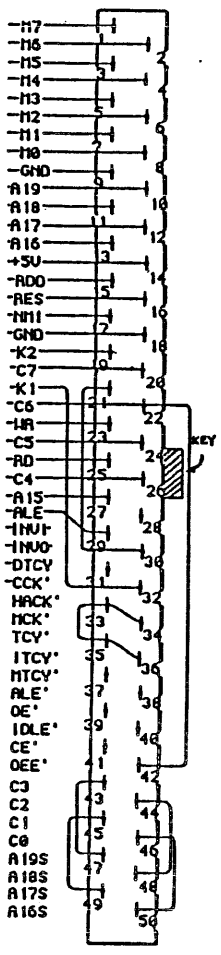
CABLE F

FOR 8048/35/39/40/49/50 (ROM Clocked)



CABLE G

FOR 1802



CABLE H

FOR 8080

Appendix E:
UniLab II Specifications

Host Computer Interface

RS-232C connector, 19,200 or 9,600 baud,
switch selectable.

Diskette Formats

IBM PC 5 1/4", MS-DOS

Emulator

Download time: 1 second for 2K bytes, including 16-bit block error check.
195 ns max access time ROM emulation. (145 ns optional)
ROM size: 32K x 8-bit or 16K x 16-bit standard. Programmable by cable, program option. Expandable to 128K bytes with optional plug-in board.
20-bit enable address decoding.
Individual 2K segments can be selected in any combination within 128K blocks.
Stand-alone operation possible as a ROM emulator.
16-bit Idle register loops target CPU, allows loading of emulation RAM and resumption of program execution.
Program loading software: from hex or binary disk files, hex serial download, memory image, ROM read.

Optional processor specific software gives full DEBUG capability including register and target memory display and change, breakpoints, and single-stepping.

Bus-State Analyzer

48-bit wide Trace Display and Memory.
48 data inputs. Two groups of 8 can be separately clocked.
6 clock signal inputs. Gated to form one bus clock:
Clock edge filter prevents re-trigger before 100 ns.
395 ns minimum bus cycle (10 MHz 68000).
297 ns with optional high-speed option.
Address demultiplexing latches included-- also used by emulator.
8 or 16 inputs available for custom connection to additional target board circuitry.

Analyzer Trigger

4-step sequential trigger.
RAM truth tables allow search for any function of 8 bits at each of six 8-bit groups, for each of four trigger steps.
8 truth tables x 4 steps = thirty-two 256-bit tables.
16-bit inside/outside range detection on address lines.
4-bit segment enable gives 20-bit address capability.
20-bit single address detection, 16-bit range detection in any 4-bit segment.
Pass Counter: wait up to 65,382 events or cycles before 4th step.
Before/After/At Pass count trigger enable.
Delay Counter: wait up to 65,382 events or cycles to stop trace.
Filter feature: Records only cycles that satisfy trigger.
Oscilloscope sync output: Sync on trigger.
Interrupt output: Interrupt target on trigger, if enabled.
LED indicates searching for trigger. Stand-alone operation possible while waiting for trigger.

Software Features

Menu or command driven with single context for all four instruments:
- 48-Channel Bus State Analyzer
- In-Circuit Emulator
- PROM Programmer
- Stimulus Generator

Extensible macro capability.
Cursor key control of text and trace display.
Pop-up mode switch panel.
Split screen displays, user-definable.
On-line glossary.
Menu-driven shell displays equivalent command lines.
40 user-definable soft-keys.
Line-by-line assembler.
Bonus features: Calculator, ASCII table, IC pinout library, memo message feature, direct DOS access, EGA/ECD support.

Software Options

Program Performance Analyzer Option.

EPROM/EEPROM Programmer

Smart programming algorithm for high speed.
28-pin Textool zero insertion force socket handles 24 and 28 pin devices.
Programs single supply EPROMs and EEPROMs.
See Appendix G. Programs 2716, TMS2516, 2532, 48016, 2732A, 2764/128, 27256/64A/128A, 27512.

Signal Inputs

TTL logic levels (74ALS inputs).
0.1 ma maximum loading includes emulator & analyzer.

Signal Outputs

TTL logic levels (74LS244 outputs).
100 ohms forward terminating resistors on Emulator data lines.
Reset output (**RES-**): open collector, 7406 thru 47 ohms.
Interrupt output (**NMI-**): open collector, 7406, low true.
9 Stimulus outputs (at EPROM socket): 8255 NMOS outputs.

Physical

Size: 2.1" hi x 13" wide x 7.8" deep.
Weight: 4 lbs. (1.8 kg.)
Shipping Weight: 11 lbs. (5 kg.)
Fits easily in a slim-line brief case.

Power

100 kHz switching supply built in.
110V $\pm$ 10% 50/60 Hz 15 watts (standard)
220V $\pm$ 10% 50/60 Hz 15 watts (optional)

Accessories Included

User's Guide.
Reference Manual.
40-pin IC clip.
16-pin IC clip.
Input stimulus cable.
Component clip adaptor probes (2).
Jumper wiring tool.

Accessory Options

Personality Paks for most popular microprocessors include:
- Connection hardware. See below.
- Disassembler/DEBUG Software (DDB-xxx).

Connection hardware included in the Personality Pak consists of either:

ROM Emulator cable. 8-bit, 24-pin version unless otherwise specified (C8-24). Analyzer cable pre-configured for your target processor.

or:

Emulation Module. Replaces processor on the target board.

Some Personality Paks also include a MicroTarget, an expandable target board.

Check with your Orion Sales Representative for current product availability.

ROM Cable options:

8-bit, 24-pin ROM emulator cable	C8-24
8-bit, 28-pin ROM emulator cable	C8-28
8-bit, direct connect emulator cable	C8-D
16-bit, 2 x 24-pin ROM emulator cable	C16-24
16-bit, 2 x 28-pin ROM emulator cable	C16-28
16-bit, direct connect emulator cable	C16-D

RAM Expansion options:

32K emulation RAM expansion board (64K total)	EB-32
96K emulation RAM expansion board (128K total)	EB-96

Appendix F:
Macros

Introduction

Macros allow you to combine several UniLab commands and give that combination of commands a new name. This is the simplest sort of macro you can make-- really more like a convenient abbreviation than like a real program.

The macro language included with the UniLab does have control structures that allow you to write more complicated macros. These structures are fully explained in several books (see page F-6). However, you don't need to understand and use the control structures to write useful macros.

For information on the UniLab program itself, order the UniLab Programmer's Guide from Orion.

Contents

How to Write a Macro

A macro definition begins with a colon (:) and ends with a semi-colon (;). These marks will not be recognized by the UniLab software until after you use the command **EXPERT.**

The first word after the colon is the name of the macro-- the new abbreviation. All the other words are the commands that the new abbreviation stands for. For example,

```
: D10       DUP 10 + MDUMP ;
```

creates a macro called **D10**, as in Dump **10** memory locations. This new word takes one argument, the starting address of the range that you want to dump. That address gets copied by **DUP** and then has **10** (hexadecimal) added to it. The macro then calls **MDUMP** on the address range.

Thus, if you were to type in:

```
342 D10
```

then you would get a dump of addresses 342 through 352.

Write Macros on FORTH Screens

The easiest way to test and alter macros is to write them in files with the screen editor, and then load the macro from the file.

The UniLab software normally holds open the file UniLab.SCR. Three screens of this file are available for you. Type **MEMO** to get into the first of those three screens.

After you type **MEMO**, press **CTRL-Z** to see the on-line prompts for the screen editor.

You exit from the screen editor by pressing the ESCape key twice in a row, or press ESCape followed by F to save any changes you made to the screen.

Don't use any UniLab.SCR screens besides those three, or you will overwrite help screens and error messages.

Load macros from a screen

You can enter your macro onto the screen, and then use ESCape followed by L to load the contents of the screen. If you keep your macros on screens, you can easily alter or update them as the need arises.

Create your own FORTH file for use with macros

After you work with macros a while, you will want to put them into a separate file. First, close the current file (**UniLab.SCR**) with the command:

CLOSE

Then create a new file with:

OPEN-NEW <file name>

and give it a size with:

<# of screens> **SCREENS**

1K is allocated per screen. Always specify at least 2 screens (numbered 0 and 1). NEVER put a macro into screen zero.

Only use **OPEN-NEW** when you want to create a new, blank file. After that, when you open the file, use the command:

OPEN <file name>.

Use the command:

<screen #> **EDIT**

to get into the file. This invokes the same editor used with the memo pad.

When you are done, use the command **UDL.SCR** to close your file and re-open the UniLab.SCR file. If you don't do this, then some of the on-line help facilities and error messages will not work.

Write Test Programs

You can use the macro capability of your UniLab system to write automatic test programs. In this section we will present some specific examples, which you can easily adapt to your specific needs.

A good starting test for a new system is to just let it execute no-op instructions. If the address bus has any shorts in it, will show as a departure from the normal address count sequence.

For example, let's assume a Z-80 or 8080 system with memory locations 0 to 7FF enabled. You can load the 00 no-op opcode into all enabled memory by just entering 0 7FF 0 MFILL. Now if you enter **STARTUP**, you should get a trace showing the first A6 addresses. By saving this trace with **TSAVE** and comparing it to the trace of an untested system, you can automate system checkout.

STARTUP and **S** are not suitable for use in automated test macros, since they cause the trace to be displayed till a key is pressed, another command must be used to start the analyzer without displaying the trace. That command is

<n> **SC** <file name>

will start the analyzer, wait n milliseconds, then compare the trace to a trace previously saved by **TSAVE**. It is very useful for automatic test sequences.

For instance, you enter

: TEST1 0 7FF 0 MFILL NORM A6 DCYCLES RESET 0 SC A:\Test.TRC ;

to define a startup test. This macro will fill the first 7FF memory locations with zeros-- no-op instructions-- then start the analyzer and compare the result to a trace that was saved as a file on drive A: using **TSAVE**.

A technician can then test the system by typing in **TEST1**. The UniLab system will reply with an OK message if the trace agrees with the one stored on the disk.

To define a test that will examine later addresses, you can enter

: TEST2 6FF ADR S 0 SC A:\TEST2.TRC ;

Of course, you will have to use **TSAVE** to save a reference trace.

If both tests work properly, we can combine them into a single test by entering:

```
: TEST  ." Test 1:"   TEST1   ." Test 2:"   TEST2 ;
```

This defines a new word TEST that will execute the 2 tests in sequence, identify the tests, and display an OK message if they pass. (Note that ." message" in a macro definition will print "message"). If either test fails, the program will automatically abort with the normal **TCOMP** fault display showing the faulty cycle and what it should have been.

Including messages in macros

If you want the operator to press a key, just put in a message to that effect by starting with ." and ending the message with " as we did above.

You can then use the command **KEY** to wait for a keystroke. This word also leaves the ASCII code for the key on the stack. You can either get rid of it with **DROP** or use it as you wish. For example, you could use it to determine the next step to be taken:

```
: SIMPLE-TEST
        ." This is a simple test " CR
        ." Do you wish to continue ?(y/n) " KEY
        ASCII y = IF REAL-TEST THEN
                  ." Bye  "   ;
```

This new word, **SIMPLE-TEST**, will execute the word **REAL-TEST** if the user enters in a "y." Otherwise, it will fall through and print out the closing " Bye."

Removing faulty definitions

If you define a test that doesn't work, you can erase the definition from memory by entering:

FORGET <macro name>

FORGET will also forget any words that have been defined since the macro that you want to forget. For instance, **FORGET TEST2** will forget **TEST2** and any other words you have defined since **TEST2.** (You thus forget a whole sequence of words. Enter **VLIST** for a list of words defined-- last word first.)

Control Structures

Your system's macro capability as described so far is really just the tip of the iceberg.

A complete FORTH system is resident within the UniLab software. This language includes constructs such as DO ... LOOP, IF .. THEN ...ELSE, and BEGIN ... UNTIL, so you can define macro words that are much more complex than the simple examples we have covered.

If you want to learn more about FORTH, the best book by far is Starting FORTH, by Leo Brodie.

If you want to contact other FORTH users, try going through:

The FORTH Interest Group
P.O. Box 1105
San Carlos, CA 94070
(415) 962-8653

They have monthly meetings in many locations and publish an excellent journal called FORTH Dimensions.

The UniLab software was all developed using the MVP-FORTH PADS (Professional Application Development System). This package, and many other FORTH books and programs, is available from

Mountain View Press Inc.
P.O. Box 4656
Mountain View, CA 94040
(415) 961-4103

The public-domain portion of that system (which is a modified 1979-Standard FORTH system) is included with your UniLab. Excellent documentation for that system is included in a book by Glen Hayden called All About FORTH, which is a complete glossary of the FORTH words.

If you plan to use the FORTH capabilities, you should buy the manual for the PADS FORTH system. It is available from

FORTHKIT
240 Prince Ave.
Los Gatos, CA 95030

The manual includes source screens and documentation for many nice utilities included with the system.

You should also request from Orion the Orion Programmer's Guide (available in July 1986).

For the Experienced

If you already know FORTH, the rest of this section will point out a few useful details for you. If not, you can skip the rest of this section. It is not necessary to learn FORTH to use the UniLab.

Redefinitions

Three standard FORTH words have been redefined in the UniLab system: **NOT** is used in trigger definitions so the synonym **0=** must be used: also **OUT** is redefined to **TOUT.** **CFA** has been redefined as **CFADR** to prevent conflicts with the hex number. Another difference to bear in mind is that the default number base in the UniLab system is Hex, while decimal is usually used in book examples.

Editor

There is a complete FORTH editor resident in your; system, which can be used for writing FORTH programs, then compiling them from the screens by using the **n LOAD** command (or escape L). See **MEMO** in the **Command Reference** chapter for a little more information on the use of the editor.

The editor is a very fast full-screen editor designed for use with FORTH screens. To use it you enter

<n> **EDIT**

(where n is a numbered 1K block) at any time when the UniLab program is running. A summary of all the editor commands will be displayed if you enter control Z.

Normally, UniLab help screens file is open so that is what you will see, but if you use

OPEN <file name>

you can edit or examine any file in 1K pieces.

Assembler

There is also a complete, reverse polish, assembler for your host processor. We include it because it doesn't take up much space and it is useful if you want to use your system to write FORTH programs. Refer to the books listed above for more details.

The assembler is a complete (FORTH) 8086 assembler. It is loaded automatically whenever you enter **CODE**. This version was adapted for MVP FORTH by Tom Wempe from Ray Duncan's version published in Dr. Dobb's Journal, Number 64, Feb. 1982.

Decompiler

Your UniLab software also includes a simple decompiler. This is useful if you want to understand the functioning of any of the UniLab or FORTH words or recall one of your own word definitions. To use it enter

' <word> **XX**

where <word> is any word in the system. The decompiler will print the address, contents, and name of each word in the definition in sequence each time you enter **XX**.

For numbers it will print "LIT" and then the number in the next line with garbage in the 3rd column. Messages defined with "."" will give garbage in the name column. Also headerless words will show junk in the name column. It's crude but amazingly effective for a definition that only occupies 12 bytes of object code!

Virtual files (.VIR)

The UniLab diskette includes three **.VIR** files: EDIT.VIR, ASM.VIR, and UTIL.VIR. These files are not needed to run the UniLab but you may find them useful for other work. The EDIT.VIR file is required to use the MEMO pad.

Some other virtual files are also used by the UniLab software. ULxxx.VIR contains trace routines and other material essential for the UniLab system to run.

MENxxx.VIR contains the menu overlay, the mode panels, and the symbol conversion menu. L1Bxxx.VIR and L2Bxxx.VIR contain the pinout libraries.

The names of these files will usually be coded with the version number, for example: UL32.VIR, MEN32.VIR, and L1B32.VIR for version 3.20. These version numbers **must** match the version number of the UniLab software or the system will crash.

The utilities are loaded from UTILxxx.VIR when needed by words like **start end TRIADS** or **from to copyto COPYSCRNS.** All of these commands are described in the documentation for the Mountain View Press PADS FORTH system.

Appendix G:
EPROMs and EEPROMs Supported

Part Number	PM	Vpp	Command To Read	To Program
EPROMs				
2716	16		RPROM	P2716
27C16	16		RPROM	PD2716
2532	16		RPROM	P2532 *
TMS2532	16		RPROM	P2532 *
2732	32	21	R2732	P2732A
2732A[b]	32	21	R2732	P2732A
27C32	32	21	R2732	P27C32
27C32	32	25	R2732	P27C32 **
2764	64	21	RPROM	PD2764
2764A	56	12.5	RPROM	P2764
TMS2764A	64	21	RPROM	P2764
27C64	64	21	RPROM	P2764
27C64	56	12.5	RPROM	P2764
27128	64	21	RPROM	PD2764
27128A	56	12.5	RPROM	P2764
27256	56	12.5	R27256	P27256 ***
27256	56-21[a]	21	R27256	P27256 ***
27C256[c]	56	12.5	R27256	P27256 ***
27512	512[a]	12.5	R27512	P27512 (8BIT mode needs 64K UniLab 16BIT mode nees 128K)
EEPROMs				
4816[b]	16		RPROM	P48016
48016[b]	16		RPROM	P48016

NOTES:

a Personality modules 56-21 and 512 are optional equipment.
b Limited support only for 4816, 48016, and 2732A.
c We do not support the Fujitsu MBM 27C256.

* The 2532 EPROM does not support the 16-bit mode of programming.
** Must cut pin 8 off of PM-32 for the 25-volt part.
*** You need a 64K UniLab to use 16BIT mode with the 27256 PROM.

Appendix H:
Microprocessor Support

The Orion UniLab can be used with almost every microprocessor on the market. Consult the price list or your Orion Sales Representative for the latest information.

Personality Paks are now available, which include all of the hardware and software you need to adapt the UniLab to a particular processor.

As the Orion product line grows, we will continue to support more processors with cabling and software. Contact your Orion Sales Representative for the latest information.

3.20 DEBUG Update:

All packages, except for Z8000, come with a line-by-line assembler.

All overlay and reserved areas are movable.

All software packages (except 1802) support the **NMI** features, some through the use of IRQ pin of the processor.

All software packages which support multiple processors provide you with a "patch menu." The is menu is presented to you when you call up the UniLab software, and is also available with the command **PATCH.**

Appendix I:
System Messages

Contents:

1. ERROR AND STATUS MESSAGES

<number> **?** - PROM programmer error. Usually this means an RS-232 error.

...enter INIT - The UniLab needs to be initialized with the host computer. You should enter **INIT**, then proceed.

Address entry error. Needs 'address-start address-end' command. - Insufficient parameters given to a command.

bad table file - Your assembler table file has been corrupted (the file called xxx.**TBL**). Try copying it from the distribution diskette again.

beyond blkmax - DOS tried to read a forth screen file beyond the limit set in the variable BLKMAX. This is mainly protection against accessing files on a hard disk as forth blocks.

beyond eof - DOS tried to read a file beyond its end of file.

boundaries for bins overlap - An error message from the histogram producer (**AHIST** or **THIST**). The program will not produce a histogram until this error is fixed. It occurs if any two ranges of addresses or times share a region. For example,

1000 - 2000 and 1500 - 2500

or even just

1000 - 2000 and 2000 - 3000.

bytes truncated, beyond 64K memory boundary. - A load was attempted that tried to load data beyond FFFF.

Can't Call DOS - Attempt to call DOS failed, probably because you are in the menu mode. If COMMAND.COM is not on your root directory, you will get this message.
Can also result from having the setting of **files=** in your CONFIG.SYS file too small. See the **Installation** chapter.

Can't find GLOSS.ORI - The file GLOSS.ORI has to be in the directory pointed to by the DOS environment string **GLOSSARY** in order to use LOOKUP and WORDS. See the **Installation** chapter.

Can't find GLOSS.TXT - The file GLOSS.TXT has to be in the directory pointed to by the DOS environment string **GLOSSARY** in order to use LOOKUP and WORDS. See the **Installation** chapter.

Can't find GNAMES.ORI - The file GNAMES.ORI has to be in the directory pointed to by the DOS environment string **GLOSSARY** in order to use LOOKUP and WORDS. See the **Installation** chapter.

Can't find <file name> - File could not be located on the disk or directory. Often caused by not having the proper values assigned to **ORION** and **GLOSSARY.** See the Installation chapter, Software Installation.
Can also be caused by renaming the UniLab software (with **SAVE-SYS**) and then trying to create a macro or operator system. The software will not be able to find a **.MCR** or **.OPR** file with the appropriate name. You will need to either rename the file from DOS, or save the system again to the old name.

Can't open overlay. - Your **.EXE** file and your **.OVL** file do not match up. Probably you have the executable file from one version of the UniLab software and the overlay file left over on your disk from an older version.

Can't R/W non-emulated address without working DEBUG control! - An attempt to access non-emulated memory failed, because the UniLab was unable to establish DEBUG control. You should establish DEBUG control manually, then try again to read and alter target system RAM as well as emulation ROM.

Can't use ALSO to add another ADR range. - The range trigger specs for the UniLab can get very complicated. The low- and high- order addresses are not intrinsically linked in the truth tables. Multiple ranges would probably yield a lot of triggers that would be combinations of wrong high and low address. For this reason, multiple ranges are not allowed.

compile only - See All About FORTH (see also Appendix F, page 5).

conditionals not paired - See All About FORTH (see page F-5).

Data is <data> **at addr** <adr> **..but is** <data> **at addr** <comp adr>- Displayed when MCOMP is used and the memory does not match up.

DEBUG control not established -
You have tried a command which requires DEBUG control, or an attempt to establish DEBUG control has just failed.
If this happens every time you try to get DEBUG control, you should turn to the TroubleShooting chapter.

definition not finished - See All About FORTH.

Disk full - The disk cannot hold any more files-- use a fresh diskette, or remove some files that are no longer needed.

diskerr # n - There is something wrong with a DOS disk operation. Consult your DOS manual for the error number's meaning.

divide overflow - Arithmetic error caused by dividing a number by zero.

eadr ng - End Address Error. An error in an internal command sent to the UniLab. Usually caused by an RS-232 error or a general system failure.

empty stack - The operating system is trying to use a number on the FORTH stack, and the FORTH stack is empty.

Emulator Memory Enable Status - A message displayed before the status of emulation ROM is displayed.

End of DOS file reached before formatted end-of-file. - Your Intel hex format file being loaded by **HEXLOAD** contains bad data, or lacks the checksum at the end.

end of text file - The end of the file being displayed by **TEXTFILE** has been reached.

End of Trace Buffer - The display has come to the end of the trace buffer. Use **TT** to start from the top, or <n> **TNT** to start from cycle number n.

Enter <parameter description> - A menu prompt, describing to you the number(s) required by that menu item. See Section 3 of this appendix.

Eprom programmed and verified, finished... - A status message from the EPROM programmer. Everything is fine.

Eprom VERIFY Error !!! - A status message indicating that there has been a problem while programming your EPROM.

file access error - The on-line assembler encountered a problem while trying to read or write or close a file. The most likely cause: you do not have the **.TBL** file in the correct directory. It should be in the directory pointed to by the DOS environment string **ORION** (SET ORION=????). See the **Installation** chapter.

File Name? --- Prompt requesting the name of the file to be opened or saved. This only appears if you do not include the file name in the same line as the command to open or save the file.

full-stack - Internal stack is overflowing. If this occurs, it might be that there is not enough room in the system. Possible cause: many user macros.

Hardware errors in Emulation Memory - While attempting to determine the memory size of your UniLab's emulation memory, the software has detected a hardware fault in the UniLab.

Hit any key to return to menu - If a menu operation uses the entire screen, the menu selections will be overwritten. This message will be displayed, letting the user return to the menu display.

in protected dictionary - You cannot FORGET words that are in the protected dictionary. See All About FORTH.

Initialization request refused - a check of the UniLab's PROM has found that it is not an Orion Instruments product.

Initializing UniLab - The host is sending an initializing command to the UniLab, and will wait for it to be acknowledged. If it does not get acknowledged, the host will wait forever. Hit **CTRL-BREAK** to get keyboard control again. See the **TroubleShooting** chapter.

Initializing UniLab Hardware errors in Emulation Memory - while attempting to determine the memory size of your UniLab's emulation memory, the software has detected a hardware fault in the UniLab.

input > 255 - see All About FORTH.

input stream exhausted - see All About FORTH.

Invalid number - an error message from the histogram producer (**AHIST** or **THIST**). You produce this error if you try to enter a value that is not a number in the base you are using. For example, FF is not a number in decimal. You will not be able to produce a histogram until you correct the mistake.

Invalid start and stop address for THIST - an error message from the histogram producer **THIST**. This error tells you that one of the two addresses that you gave to THIST is missing or is not a number in the base you are using.

isn't unique - the word used as a macro name is already used. This won't hurt anything except you can no longer use the previous word.

ladr ng - Load Address Error. An error in an internal command sent to the UniLab. Usually caused by an RS-232 error or a general system failure.

len ng - Length of data transmitted is bad. An error in an internal command sent to the UniLab. Usually caused by an RS-232 error or a general system failure.

loading only - see All About FORTH.

Lowbound is larger than highbound - an error message from the histogram producer (**AHIST** or **THIST**). This error occurs if a bin has a starting value that is higher than the ending value. You cannot make a histogram until you fix this error.

max 3 qualifiers - You cannot enter more than three qualifiers with the AFTER command.

MISC inputs cannot be used in qualifier or filter specs. - The MISC lines are excluded from being used in an AFTER or an ONLY trigger specification.

needs <number> **parameters** - The command needs more parameters than were given. Consult the glossary to see what commands the word needs.

No Analyzer Clock - The UniLab is not receiving a clock signal. All clocking is through the **RD-**, **WR-**, **K1-**, and **K2-** lines to the UniLab (except for the F cable, which gets clock from the piggyback rom socket).
The processor might be stopped, or these four lines might not be correctly connected. It is also possible that the UniLab software is not waiting long enough before checking for the analyzer clock. Use <value> **=WAIT** to increase the amount of time that the UniLab will let pass. The default value is 140.

no drive - The disk drive you tried to access does not exist. Usually received if you enter a command such as **BINLOAD B:myprog.bin** when you have no drive B.

No Good! (Above is correct.) Was: - Message produced by **TCOMP** when it detects a difference between the trace on the diskette (which it assumes is correct) and the trace that has just been made, which is sitting in host memory.

No Memory Enabled - No emulation memory is set up to be selected as ROM for the target board. This is not an error.

no room - Dictionary is full and won't hold any more macros. Use FORGET to forget unused macros before adding new ones.

no room for <heads or bodies> - Dictionary is full and will not hold any more macros.

no trigger! - Only shows on a compare trace with the SC command. This means that the trigger spec was not reached within the time you specified.

Not a DOS text file - TEXTFILE is trying to read in a file that is not a DOS text file.

Not available in menu mode - Some commands are restricted when using menus, such as calling up the HELP displays. These would overwrite the menu if they were permitted.

Not done till delay count = - The UniLab is waiting for enough events to fill up its trace buffer. If too few bus cycles occur, or if filtering is being used, this message will be generated. You can wait, or you can hit any key to break out of this state. You will then need to type **TD** (Trace buffer Dump) to see the filtered cycles or the cycles before the clock was stopped in the buffer. Note that the cycles will appear at the end of the trace buffer, while there will still be left-over garbage toward the beginning of the trace.

Not emulation memory - same as "not enb."

not enb. - Emulation memory is not enabled for this data transfer-- not an error message, but a reminder that the address you are referencing is not enabled (for example, target system RAM).

Not enough bins available - An error message from the histogram producer (**AHIST** or **THIST**). This error occurs if you try to allot more than 15 bins using the Subdivide key (**F3**). This can occur if you already have several bins alloted and then try to subdivide one of them among all the bins.

not enough memory - Either the **MACRO** command can't get enough RAM to create a macro system, or the on-line assembler (**ASM** or **ASM-FILE**) can't allocate enough RAM to read in the table file. Use **?FREE** to check on the amount of free RAM available, and then use either **=HISTORY** or **=SYMBOLS** to reduce the amount of RAM dedicated to those two space hogs. You will have to **SAVE-SYS** and then restart the UniLab software before the new settings will take effect. Look up the appropriate entries in the **Command Reference** chapter.

not found - Word used in macro definition does not exist. Check spelling. Note that macros may use other macros, but each word in a macro must be already defined.
Also used to let you know that the file was not found by **SYMLOAD**, or that the file was not the right type.

not in dictionary - **HELP** cannot find the word in the UniLab glossary.

not recognized - Command or word not available. This is usually a misspelled word. If it is a known word that should exist, type BYE to exit the system.

ODD or EVEN ? - A prompt to tell the user that either the address range must be given (with an implicit odd or even start address) or the command ODD or EVEN must be used when reading or programming a 27512 in the 16-bit mode.

ok - Word returned by the UniLab system software every time it accepts a command.

OK - returned by the software when in a macro system.

out of bounds - Stack underflow-- a catastrophic error. If it occurs, exit system and re-boot. Do <u>not</u> **SAVE-SYS**.

parse error - The on-line assembler does not recognize your assembly language command.

rcv sum ng!! - An error occurred when a HEXFILE was being received. This is usually an indication of an RS-232 problem.

Reading.... - Message displayed when reading a 27512 prom, since it takes a few minutes.

reading text...please wait - TEXTFILE reads in a file to the UniLab program and analyzes it for the number of lines. While it is creating a line index, this message is printed out.

Requires <parameter description> - A parameter entry error message. Indicates that the wrong number of parameters was given to a command. See Section 2 of this appendix.

resetting - The analyzer has been started, and the reset line toggled low then high again, to reset the target processor. Look at, chapter 6, Section 4.5.

RS232 err # n ...enter INIT - The RS-232 communication between the host and the UniLab is not functioning properly. Check the cable hookup. If you have changed baud rate, make sure that the software setting corresponds to the switch setting in the UniLab.

STANDALONE mode.... Use PROMMSG later to communicate with UniLab after EPROM is programmed. - Status message after issuing a prom programming command in **STANDALONE** mode. Do not re-initialize the UniLab after a standalone PROM programming, or you will lose the status message.

Symbol table full - the memory allocated for symbol tables has been filled. Use **?FREE** to find out how much memory has been allocated to symbols and to screen history.
Use <# Kbytes> **=SYMB** to change the size of memory allocated for symbols. You will need to **SAVE-SYS**, exit to DOS and then come start up the UniLab software again.

target adr (not EMENABLED) - This is usually not an error message. It is a notice that you are addressing a memory location that you have not enabled. Either you have made a mistake, or you are purposely addressing RAM on your target board rather than emulation ROM. A read or write to target memory is only possible after you have established debug control.
This message is always printed out as a reminder that the DEBUG is performing the memory read or write, and that this is not a simple transfer of data between the host and UniLab.

The command is: <command> - This is a menu display to show the user what the associated command would be if entered as a command rather than as a menu choice.

This Eprom does not seem to be erased!!! - An error message when you try to program an EPROM that is not blank.

TO address is smaller than FROM address. - A command was given which has mismatched parameters-- the second number is too small.

Too many files - Not enough files have been allocated, so you cannot open another. This can be changed by changing the **CONFIG.SYS** file in your root directory to contain the line **FILES=16.** See the **Installation** chapter.

unloadable - See All About FORTH.

Unsupported record type. Types 0, 2, & 3 only are supported. - The Intel hex format file that you are trying to load with **HEXLOAD** is of the wrong record type. Bytes 7 and 8 of each line of the file tell what record type the line uses.

Wrong UNILAB.SCR file - The file named UNILAB.SCR on the disk is not the correct version for the ULxxx.EXE file. Make sure that you copied everything from the master diskette.

Wrong <file name> - The **.MCR** or **.OPR** file does not match the current **.EXE file.**

2. PARAMETER ENTRY ERRORS

These messages result when you call a command without giving it the proper number of parameters. You can enter any command name without the parameters, and get a prompt that tells you what the command requires.

Requires a Goto-Address, and the next Breakpoint-Address

Requires the From-Address, and the To-Address

Requires a Number-of-Lines

Requires the From-Address, the To-Address, and a Value

Requires Source-From, Source-To, and Destination addresses(1)

Requires the Start-Address and the #-of-lines

Requires an Address

Requires a Word (16 bit) Value

Requires a Cycle#

Requires a Byte (8 bit) Value

Requires an Address or Range (Address1 TO Address2)

Requires the Cycle#

Requires a Block-Number

Requires a Value and a Destination-Address

Requires an Address and a Block-Number

Requires a Nibble (4 bit) Value

Requires 'host-address target-address #bytes'

3. MENU MESSAGES

These messages are generated by entries in the menu, to let you know what values that menu item needs. When you get one of these messages, enter the value requested.

```
Enter the Starting target address:
Enter the Ending target address:

Enter the Target address:
Enter the First value:
Enter the Last value:
Enter the Value:

Enter the Bit# (0-7):
Enter the desired hex output code (0-FF)
Enter the Trigger address:
Enter the Source address:
Enter the Destination address:
Enter the number of lines to disassemble (default=5):
Enter the starting address of the first block to compare:
Enter the starting address of the second block:


Enter the breakpoint address in emulation memory:
Enter address to continue execution:
Enter next breakpoint address:
Hit key for next debug function (from above):
```

Appendix J:
.BIN Files and .TRC Files

Overview

Your distribution diskette includes one or more **.BIN** files, the binary image of the simple target program for your microprocessor. That file is described in Chapter Three: **The Guided Demo.**

The chart in this appendix tells you which file to load into memory, and where to load it.

With some packages, we distribute a **.TRC** file to test your cable connection to your processor, as described in section 4 of the **Installation** instructions of Chapter Two. This appendix includes the name of the **.TRC** file, for those processor packages that include demonstration traces

Patches and 3.20 Software Update:

All software packages which support multiple processors now provide you with a "patch menu." The is menu is presented to you when you call up the UniLab software, and is also available through the command **PATCH.** Thus, the "patch word" is always **PATCH.**

.BIN files

You can load the sample program into memory with the command **LTARG,** which also takes care of enabling memory and any other needed details.

However, loading the **.BIN** file from disk gives you familiarity with the procedure you will have to follow when you load your own program from disk.

Before using BINLOAD

With most processors, the only preparation you need is enabling the 2K segment into which you will load the binary file. For example, to load the binary file **test65** into emulation memory for the 6502:

```
FF00 EMENABLE
FF00 FFFF BINLOAD test65
```

With some processors, you might have to enable more than 2K

of memory, or set the value of some variables. These exceptions are noted in the following chart.

The <to address> argument of BINLOAD

BINLOAD needs two arguments in addition to the file name: the <from address> and the <to address>. These addresses tell **BINLOAD** where to start loading the file, and when to stop.

Of course, if **BINLOAD** reaches the end of the binary file before it reaches the <to address>, then it will stop loading.

You should always give as the <to address> the highest address in emulation memory. Though the sample programs are small, the reset vector is often at the top of memory, pointing at an address closer to the bottom of memory.

.TRC files

In general, you use the trace file to verify that you have the proper connections between your UniLab and target board.

Comparing your trace to the one on diskette

Load the sample program, either from the **.BIN** file or with **LTARG**, and use **STARTUP** to generate a trace. Then use:

AA TCOMP <trace file name>

to compare your entire trace to the trace on diskette.

You can also use:

TSHOW <trace file name>

to look at the trace that is stored on diskette.

Comparing partial traces

Sometimes the difference between the standard trace file and your trace will be trivial. For example, some simple target programs contain instructions that read RAM locations which have not been initialized-- and so the value stored in that location will vary.

The UniLab software stops checking the traces after it finds the first difference. If you want to compare your trace starting after the trivial difference, then you will use the **TCOMP** command with a different parameter:

<number of cycles to compare> **TCOMP** <trace file name>

Note that the number of cycles is a count starting from the end of the trace buffer.

You can also use **TMASK** to specify that only certain columns of the trace should be compared. Consult the TMASK entry in the **Command Reference** chapter.

N/A

Trace files are not available (n/a) for all processors.

Chart of .BIN and .TRC files

	Processor	.BIN file	Load in starting at address	.TRC file
DDB-48	8048 family members in expanded mode	TEST48	00	DEMO48
DDB-51	8051 family members in expanded mode **NOTE:** You must also enable F800 to FFFF for the overlay area.	TEST51	00	DEMO51
DDB-51P	8051P, 80C51P **NOTE:** You must also enable 800 to FFF for the overlay area.	TEST51P	00	DEMO51P
DDB-611	68HC11	TEST611	FF00	DEMO611
DDB-63	6303R	TEST63	FF00	DEMO63R
	6303X	TEST63	FF00	DEMO63X
(obsolete)	63P01	TEST63	FF00	DEMO63P
DDB-65	6502 or 65C02	TEST65	FF00	DEMO65
DDB-65P	R65/11EB	TEST65P	FF00	DEMO65EB
	R65/41	TEST65P	FF00	DEMO6541
	R6511Q	TEST65P	FF00	DEMOR65Q
DDB-68	6800 or 6808	TEST68	F800	DEMO68
DDB-681	6801	TEST681	F800	DEMO681
	6803	TEST681	F800	DEMO681
	piggyback	TEST681	F800	n/a
DDB-682	6802	TEST682	F800	DEMO682
DDB-685	6805E2	TEST685	1F00	n/a
	HD6305	TEST685	1F00	n/a
	piggyback chips	TEST685P	F00	DEMO685P
	6805E3	TES685E3	FF00	n/a

	Processor	.BIN file	Load in starting at address	.TRC file
DDB-688	68008	TEST688	00	DEMO688
DDB-689	6809E	TEST689	FF00	DEMO689
DDB-68K	68000	TEST68K	00	DEMO68K
DDB-85	8085	TEST85	00	DEMO85
	8080	TEST85	00	n/a
DDB-86	8086 min	TEST86MI	F800	DEMO86MI
	8086+	TEST86MA	F800	n/a
	80186	TEST186	F800	n/a
	80286	TEST286	F800	n/a
	NOTE: With all members of the 8086 family, you should use **SEG'** before loading into emulation memory with **BINLOAD.** After loading, you can turn back on the interpretation of addresses as offsets from segments, with **SEG.**			
DDB-88	8088 min	TEST88MI	F800	DEMO88MI
	8088+	TEST88MA	F800	n/a
	80188	TEST188	F800	DEMO188
	NOTE: With all members of the 8088 family, you should use **SEG'** before loading into emulation memory with **BINLOAD.** After loading, you can again turn on interpretation of addresses as offsets from segments, with **SEG.**			
DDB-96	8094, 8095, 8096, 8097	TEST96	2080	DEMO96
	NOTE: You must enable the entire area 0 to 2FFF, for the overlay area.			
DDB-HD64	HD64180	TESTHD64	00	DEMOHD64
DDB-SC8	NSC-800	TESTSC8	00	DEMOSC8
DDB-S8	ROMless members of the super 8 family	TESTS8	20	DEMOS8

	Processor	.BIN file	Load in starting at address	.TRC file
DDB-Z8	Z8	TESTZ8	0C	n/a
	piggyback	TESTZ8P	0C	DEMOZ8P
	NOTE: You must enable 00 through FFF.			
	NOTE #2: Assumes you have pulldown resistors on the upper address lines. If you have pullup resistors, then **ALSO FF0C EMENABLE** and then **0C FF FF0C MMOVE.**			
DDB-Z80	Z80	TESTZ80	00	DEMOZ80
	NSC-800	TESTZ80	00	DEMONSC8
	HD64180	TESTZ80	00	DEMOHD64
DDB-Z8K	Z8001, Z8003			
These two	processors need two values initialized:			
	0 =ROM.SEGMENT			
	0 =RAM.SEGMENT			
		TESTZ81	00	n/a
	Z8002, Z8004	TESTZ82	00	n/a
DIS-18	1802 family	TEST18	00	DEMO18

FULL INDEX

This index covers all chapters in both volumes. The index at the end of Volume I covers only that first volume.

Neither index covers the appendices.